CSC309

Contents

1 - HTML and CSS

The Web

How do computers communicate with each other? Once both computers are joined on the same network address:

1. They communicate through their identities, **Internet Protocols** (IP).
2. The operating system connects through **port numbers** (i.e., socket), where the client communicates with one process to the process of the server (or vice-versa).
    - A client connects to the server using its IP and port number (e.g., `192.168.1.102:5000`).
3. Then, a two-way connection is established. They can then send **packets** (small pieces of information) to each other.
    - Each packet should send its IP, port, and the information it would like to send. This two-way connection is called TCP-IP (Transmission Control Protocol).

    [!NOTE] Netcat Netcat, or `nc` is a software that allows connections between a server and client(s). It is quite secure, since it has end-to-end encryption. Server: `nc -l 8000` sets up a server with port `8000`; `-l` is the parameter for "listen". Every computer has its own IP address `127.0.0.1`, or `localhost` and port `8000`. - You can check a device's IP using `ifconfig` (`ipconfig` for Windows) and looking at the `inet addr`.

Domains

Domains - Mappings to IP addresses to make them more human-readable. (e.g., `www.google.com` → `142.251.41.78`).

- Domains are stored in **Domain Name Servers (DNS)**. There is one big DNS server, 8888, hosted by Google.

Clients first **resolve** the domain, then connect to the IP address.

- A server's DNS has a list of IPs that it can connect to; typically there is only one.

Stateful and Stateless Connections

Stateful - Two-way open connection; what the server responds to depending on **previous messages**.

- The server should keep track of thousands of open connections. If the connection breaks, all of the **state** is lost. **Stateless -** The client will send one protocol, and the server will send one protocol back; no other protocols are shared.
- This protocol is called **HyperText Transfer Protocol** (HTTP).

HTTP Messages HTTP messages are strings that have a special format:

- They request specific targets:
    - **Path:** `./signup`, `/account/index.html`
- They have a header and body.
- Their default port is `80`.

Response Codes are the messages that the client will send to the server.

- **Success:** $200 - 299$
    - 200 "OK", 201 "created".
- **Redirection:** $300 - 399$

- 301 “Moved permanently”.
- **Client Errors**: 400 – 499
  - 404 Not Found, 400 bad request, 403 permission denied.
- **Server Errors:** 500 – 599
  - 500 Internal server error, 502 bad gateway.

Web Browser

The web browser has two parts, the **client** and the **rendering application**.

- The client enters the **Uniform Resource Locator** (URL), then connects, sends requests to the server, and receives responses.
- The rendering then responds with the HTML, images, PDFs, or other files.

  [!NOTE] Curl `curl` is a CLI command that gives you the request header from a web address (e.g., `curl https://www.web.cs.utoronto.ca`). - It can print unreadable text, but the rendering application will turn it into human-readable content back to the client.

HTML and CSS

HTML - Text format with `<body>` and `<head>` tags and elements (attributes). **Tags:**

- Headings `<h1>` to `<h6>`
- Paragraphs `<p>`
- Links `<a>`; stands for anchor.
- Images `<img />`
- Lists `<ol>` or `<ul>`
- Tables `<table>`
- Navigation Bar `<nav>`
- New Line `<br>`
- Inline organization `<span>`
- Block-level organization `<div>` **Attributes:**
- **Style** attribute
- Identifiers (id, class)
- `src` or `href`

  [!NOTE] Forms Forms can be created in HTML using various input tags: - Text field `<input type="text" />` - Passwords/emails `<input type="password" />` - Radio button `<input type="radio" />` - Checkbox `<input type="checkbox" />` - Text area `<textarea />` - Submit `<button type="submit" />` The **action** attribute defines the URL/path of the HTTP request, and the **method** attribute is its parameter. Inputs have **name** and **value** attributes.

  GET and POST GET - Queries and retrievals (e.g., Google searches); query parameters get **appended** to the end of the URL. **POST** - Private user data (e.g., names, passwords, etc.)

CSS

Style attributes - How the elements look. Usage: `<h1 style="color: red">Hello</h1>` **Selectors -** The same styles can be applied to an arbitrary set of elements (e.g., of a certain tag, specific element (id), or a set of elements (class)).

- Example: `p { color:red; }` makes all paragraphs red.
- Precedence: Use `!important` to bypass selectors.
  - Inline CSS → Style Tag → CSS File
  - ID → Class → Tag **Stylesheets** are separate CSS files with the content of style tags (reusable selectors for styles). They are imported via `<link rel="stylesheet" href="style.css" />`.

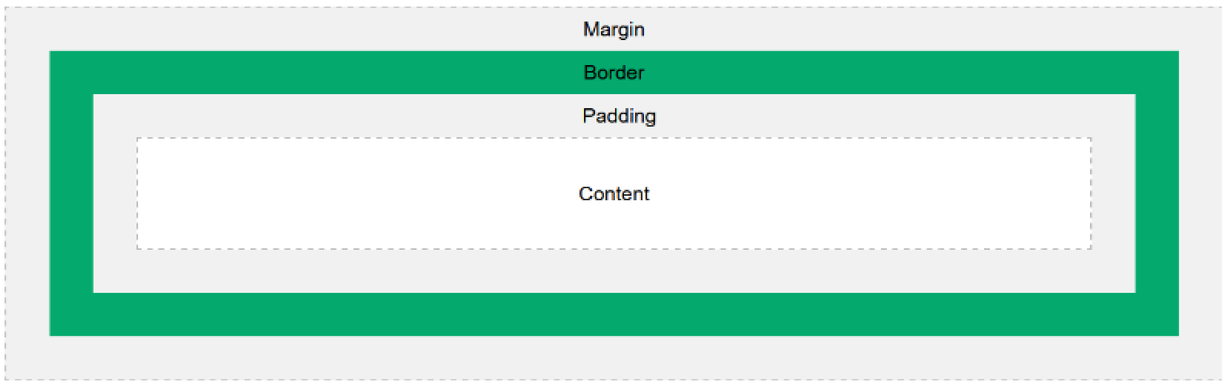
Basic CSS Properties:

- Color
- Background-color
- Font-size
- Text-align
- Width
- Height
- Z-index
- Font-family
- List-style-type
- Opacity

Units - Used for width, margin, font-size, etc.

- **Absolute:** cm, in, px
- **Relative:** %, vw, em (element's font-size), rem (root element's font-size), fr

Box Model:



Borders:

- Shorthand: `border: 10px solid red;`
- Style: solid, dotted, etc.
- Width: 10px/thin, etc.
- Color
- Radius: 5px, etc. Specific edges can be specifically styled (e.g., `border-top-width`, or `border-width: 1px 2px 3px 5px`).
- Order: top, right, bottom, left; Top and bottom, left and right.

2 - JavaScript

JavaScript

JavaScript - A scripting language interpreted at runtime with no JAR or EXE file.

- All browsers have a JS interpreter, but is not excluded to client-side web applications (e.g., NodeJS, Next.js, etc.)

History

- Originated in the 1990's by Brandon Eich based on a deal between Netscape and Sun.
  - Netscape's browser will support Java apps, and Eich's language will be called JavaScript.
  - JavaScript was created in just 10 days! Other than this, there is no further relevance between Java and JavaScript. Now, JavaScript is one of the most popular languages used by programmers.

Syntax

JavaScript's syntax is somewhat similar to Java and C-style syntax, since it was very popular then. JavaScript does not have a preference towards adding semicolons or not.

Variables

```
var x = 5;
var y = "hello";
console.log(x + y); // returns '5hello'

typeof(x)  // returns 'number'
```

- Note that all numbers (floats, integers, etc.) are still type "number".
- JS is dynamically-typed (like Python): A type will be assigned automatically to a variable, and you can change its type. JS has syntax similar to Python's `fstrings`:

```
var x = "yes";
console.log("They answered ${x}");
```

Objects

```
var obj = {
    name: 'abcde',
    age: 2,
    address: "1 Lane"
}

obj.address  // prints '1 Lane'
```

Arrays are also considered objects:

```
var arr = [1, "frfr", {}, [], false]
arr[0]  // 1

// Add new item to the array
arr.push(0)  // returns 6 (new size of array)

// Get the length of the array
arr.length
```

JSONs (JavaScript Object Notation) is the most popular way of serializing objects, now has its own file type as well. Objects are the only **mutable** types in JS.

Null and Undefined If `var x = null`, `x` has type 'object'. We also have an undefined type.

```
var a = { name: null }
a.name  // prints null

// Using previous example:
obj.y  // Returns 'undefined', only has one value
```

Undefined exists to avoid runtime errors or crashes, so it will return for the variable/function instead.

- Example: If we access index 1000 on an array size 6, it will just return `undefined`.

Functions

```
function name(par1, par2, par3) {
    // code to be executed
}
```

Functions have type 'function', and therefore can be passed as a parameter or property (method):

```
var obj = {
    f: function(x) {
        return x + 2;
    }
}

cars.clear = function(){
    this.length = 0;
}
```

Classes Templates for creating objects, heavily based on Java syntax. They are not really object-oriented classes, they are just special functions. Classes additionally support inheritance.

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    // Getter
    get area() {
        return this.calcArea();
    }

    // Method
    calcArea() {
        return this.height * this.width;
    }
}

const square = new Rectangle(10,10);
square.area  // returns 100

typeof(square)  // prints 'object'
```

Conditions If Statements:

```
if (typeof(cars[0]) === "number" && cars[0] < 0) {
    cars[0] *= -1
} else {
    console.log("Bad");
}
```

> [!NOTE] Note on Equivalence == - Compares values of two variables w/ type conversion. === - Compares the values w/o type conversion; strict equality. === prevents transitivity (e.g., 1 === 1, '1' === '1', but 1 !== '1')

Loops:

```
while (cars.length > 0) {
    cars.pop()
}
```

For loops can be iterative or use integers:

```
for (var i=0; i<10; i++) {
    console.log(i * i)
}

for (item of arr) {
    console.log(item)
}

// Array Specific
names.forEach(function(name, index) {
    console.log(name + " at index " + index)
})
```

Switch Statement:

```
switch(cars[0]) {
    case 1:
        console.log("int")
        break
    case "name":
        console.log("str")
        break
    case x:
        console.log("var " + x)
        break
    default:
        console.log("none")
}
```

Scope There are three types of a scope (the current context of your code; where you can access certain variables and functions). Calling the variable/function outside of its scope will cause a syntax error or recall error.

Global Scope Access is given outside of any function; variables are accessed from anywhere in the program.

Function Scope Variables defined **anywhere** in a function are local to that function (i.e., it can be used anywhere inside its function, but not outside).

```
function myF() {
    var car = "Volvo";
    // code can use car
}

// code cannot use car
```

Block Scope Limits a variable to its **block** inside the function; use `let`:

```
function f(n) {
    if (n > 10) {
        let tmp = 2;
    }
    // tmp cannot be accessed here
}
```

Note that `let` does not support declaration, but does support re-assignment.

- `let` is more similar to regular variables in other languages, and is preferred over `var` because of it.

Arrow Functions Arrow functions are a convenient way of defining functions, though they are almost equivalent to regular functions.

```
function regular(a,b) {
    return a + b;
}
// equivalent to
const arrow = (a,b) => {
    return a + b;
}
// can be reduced further!
const conciseArrow = (a,b) => a + b;
```

Arrow functions are great for simplifying our code and making it easier to look at!

Code Simplifications Instead of a for loop, we can use `forEach` or `map`:

```
var names = ["ali", "hassan"]
names.forEach(
    (item, index) => console.log(item + " at " + index)
    )
// Equivalently:
upper = names.map(item => item.toUpperCase())
```

We can simplify this even further using a `filter`!

```
let students = [ {name: "John", id: 1} ]
let john = students.filter(item => item.name === "John")
```

`reduce` also allows us to accumulate values of an array:

```
let maxCredit = employees.reduce(
    (acc, cur) => Math.max(cur.credit, acc), Number.NEGATIVE_INFINITY
)
```

Destructuring

```
const hero = {
    name: 'Batman',
    realName: 'Bruce Wayne'
};
// equivalent to
const { name, realName } = hero;
name;  // prints 'Batman'
realName;  // prints 'Bruce Wayne'
```

Subtlety Regular functions use the `this` value to reference the object that called the function.

- Methods and event listeners are the actual object/element.
- Global functions are the global object (window). Arrow functions do not have a `this` value; do not use arrow functions as an object method or listener!
- They can, however, **bind** (capture) `this` like other closure values.

```
const person = {
    name: 'Aaron',
    sayHi() {
        setTimeout(() =>
            console.log(this.name + " says hi!"), 500);
    }
}
```

Client-side JavaScript

JavaScript can be written alongside HTML files in a `<script>` tag.

- **Inline** - `<script> console.log(1 + 2) </script>`
- **JS File** - `<script src="region_dropdown.js"></script>`

The browser will create the **DOM Tree** (Document Object Model) of the page.

- Each **element** is a node.
- Child elements are children of the parent node.
- Scripts access DOM through the **document** variable.

Getting Elements:

```
document.getElementById("st-2")
document.getElementsByClassName("share-buttons")
document.getElementsByTagName("ul")

document.querySelector("#submit-btn")  // first element matching the selector
document.querySelectorAll(".col-mid-12")
```

Relevant nodes are accessed through priorities: parent, first child, last child, child nodes, next sibling.

Manipulating Elements: You can change properties of elements using `innerHTML`, `style`, and `getAttribute()`.

```
let body = document.body
body.innerHTML = "<h3>hello!</h3>"

h3 = document.getElementsByTagName("h3")
h3.style.color = "green"
h3.setAttibute("class", "title")
```

Events: The browser also monitors events:

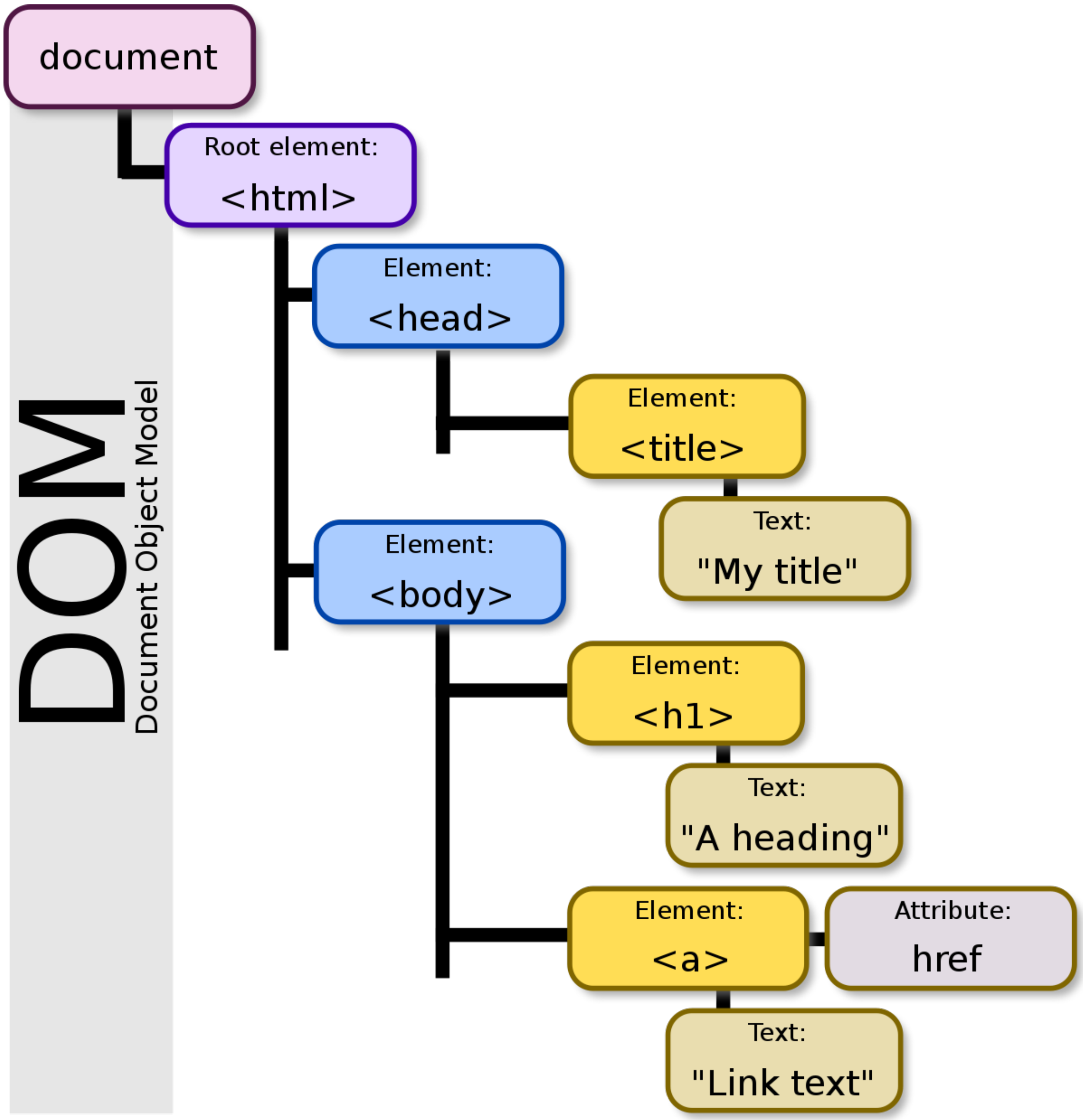


Figure 1: 1_b9i7n4XOt1L2AvWzGEY36Q.png

- **Document** events: `onload`, `onkeydown`, `onkey`, …
- **Element** events: `onclick`, `onmouseover`, `ondrag`, `oncopy`, `onfocus`, `onselect`, … Functions can be assigned to these events:

```
h3.onclick = function() {
    this.innerHTML = "you clicked on me"
}
```

3 - Node.js, Next.js, APIs

Web Development

Frontend Development - What the user sees and how they interact with the website.

- User interface (UI) and user experience (UX).
- Client-side code; HTML/CSS renderings and JS codes running on browser. **Backend Development -** How the application works in its server (e.g., Server → API and database).
- All logic and processes that happen behind the scene.

Backend Frameworks

Web Servers - Listens on specified port(s) and handles incoming connections to generate response, fetch files, and forward them to corresponding applications.

- Also handles balancing, security, file serving, etc.
- **Examples:** Apache, Nginx

Developing the web servers for an application from scratch is very pedantic and time-consuming. Now many **backend frameworks** exist that have pre-implemented components for HTTP requests.

Types of Backend Frameworks

- **PHP** - No longer common; e.g., Laravel, CodeIgniter
- **Python -** Django (useful in larger projects), Flask, FastAPI
- **JavaScript -** Express, Next.js
- **Ruby -** Ruby on Rails

  [!NOTE] JavaScript and Web Dev JavaScript is the **only** language that can be used in **both** front-end and back-end, since all websites run JS natively. For the simplicity of this course, Next.js and Node.js will be the only frameworks used. - Using only JS also helps with code consistency, unity, type sharing, library sharing, etc. - Node.js specifically allows JS projects to include backend, frontend, or both, at the same time!

Node.js

Node.js is a **runtime environment** that will run JavaScript server side (so JS does not have to run on the browser).

- Also includes a package manager, console, build tools, etc.

Basics of Using Node

Console - Write `node` in your shell to initialize the console. All JS code can run normally, but note that the code runs directly on your system rather than the browser. Files can also be run using `node <FILENAME>`.

Installing Libraries - npm (Node Package Manager) to install packages/libraries. Use `npm install <PACKAGE>`, and they will be stored in the `node_modules` directory.

- Packages are automatically generated and maintained in the `package.json` file.

Import and Export - Variables, classes, or functions can be externally used from a JS file (i.e., module).

```
const var1 = 3, var2 = (x) => x + 1
export { var1, var2 }  // Executable from anywhere in project now!

// Can also be reduced to one statement
export const var1 = 3, var2 = (x) => x + 1

// Importing modules
import { var1 } from './App'
```

Each module can have **one** default export (e.g., `export default App`), then importing can be done using `import App from './App'`.

- Then, the names do not have to match and can be imported under any arbitrary name.

Using Node for Backend Projects

Use **Node Package Execute** (npx) to create a boilerplate/easy starting point for your project:

1. Create the app using `npx create-next-app@latest`
   1. Answer "No" to all prompts (for now).
2. Looking through the project:
   1. `package.json` will have Next.js and React pre-installed!
   2. `/pages/api` has all of the backend code we need.
   3. The `pages` directory allows every JS file to correspond to a URL path, except for underscored files (e.g., `/_app`).
3. Start a project/server using `npm run dev`.

APIs **APIs (URL Handlers) -** Application Programming Interface; how an application can be talked to.

The default export is the **handler** function, which executes when a request arrives for a corresponding URL path.

- Inputs **request** and **response** objects; does not have a return!

```
// Example
export default function handler(req, res) {
    ...
}
```

Status/Responses:

```
    // text
res.status(200).send("Hello")
    // JSON
res.status(200).json({message: "Hello"})
    // Redirect
res.status(302).redirect("/api/users/login")
    // Error
```

```
res.status(400).send("Bad request")
res.status(500).send("Internal error")
    // HTML Response
res.status(200).send(`
    <html>
        <head>
            <title>Success!</title>
        </head>
        <body>
            <h1>Login successful.</h1>
        </body>
    </html>
    `)
```

Request Objects

- **Method -** `if (req.method === 'POST') { ... }`
- **Headers -** `if (req.header['type'] === 'application/json')`
- **Queries -** `const search = req.query.search || null`
  - Will have a ? in the URL.
- **Request Body -** `const { username, password } = req.body`
  - Assumes JSON by default.

How do Web Apps Work?

Many applications used to have a combined backend/frontend and had the following process:

1. Browser requests a URL.
2. API handler returns HTML with the appropriate CSS and client-side JS.
   1. Can be read from separate files, and dynamically filled based on request.
3. Users click on links and/or fill out forms.
4. API handlers process GET/POST requests, returning another HTML response/redirect. This process has worked for over 10 years, but has some drawbacks:

- Too backend-oriented.
  - Frontend logic is served as part of backend handlers.
  - Code gets messy and hard to understand.
- Limits frontends to web browsers (e.g., to mobile).
- Frontend cannot be sophisticated (e.g., single-pages only).

As a result, we separate our frontend and backend!

- **Modularity -** Changes in frontend will **not** affect backend, vice-versa.
  - Different services/apps will talk to each other via protocols.
  - Web applications will have a set of HTTP requests.
- **Consolidation -** We can have one backend and **multiple** frontends (e.g., between web, Android, and iOS). Backend views only have data retrieval and manipulation, and does not care about how data is shown, UI, or UX.

  [!NOTE] JSONs as Response Formats JavaScript Object Notations, or JSONs are a popular response format, derived from JS syntax for designing objects. They are human-readable and fast; many languages have built-in parsers and support! - Primitive types: Number, string, boolean, null - Array: Ordered collections - Object: Key-value pairs Array elements/objects can be any type.

API Architecture

How the API components interact and communicate vary based on its architecture. **Representational State Transfer** (REST) has a set of URL endpoints doing data management tasks. All request and response data is formatted as a JSON. Next.js natively supports RESTful APIs!

- The request body is parsed from a JSON into a JS object, accessible in `req.body`.
- There is native support for returning JSON responses via `req.status(...).json(...)`.
  - Appropriate headers will also be set.

Example of Backend Code

```
// pages/api/users/signup.js
import { capitalize } from "@/util";

export default function handler(req, res) {
  const { username, email, password } = req.body;

  console.log(typeof username);

  if (!username || !email) {
    return res.status(400).send("Missing username or email");
  }

  if ((password?.length ?? 0) < 6) {
    res.status(400).send("Password must be at least 6 characters long");
    return;
  }

  // actually store the user into the database
  res.status(200).send(`Received ${username} ${email} ${password}`);
}
```

```
// pages/api/users/index.js
export default function handler(req, res) {
  const { name = "someone", age } = req.query;

  res
    .status(200)
    .send(`you are ${name}, ${age || 20} years old, at /api/users`);
}
```

`pages/api/users/signup-form.js` will have the default export function handler that sends to the frontend: `export default function handler(req, res) { ... }`.

4 - Async, Models, and ORM

Asynchronous Programming

Event Loops

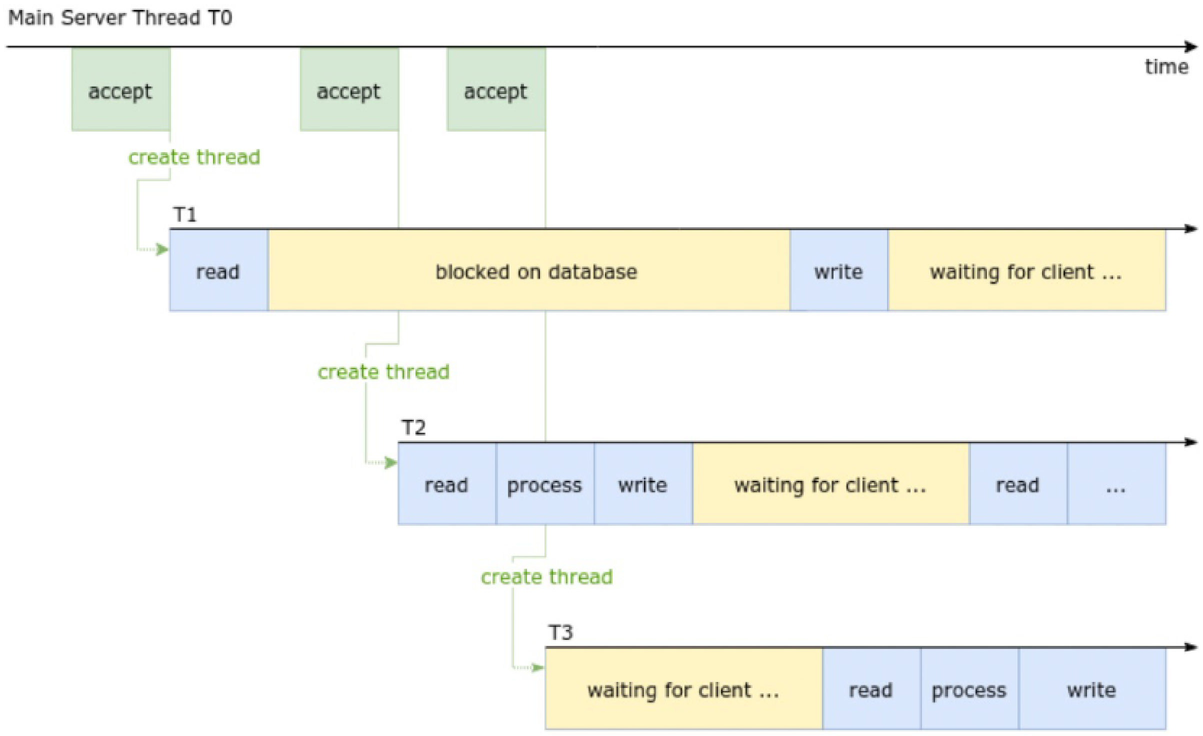
API handlers are able to do more sophisticated work, including reading/writing from databases, file operations, and making requests to other servers/APIs. These can be slow compared to the rest of the handler's job (which is just object manipulation).

- Determine what the bottleneck is (i.e., complex CPU processing vs. waiting for I/O).

  [!NOTE] I/O and CPU Bounds I/O and CPU bounds are tasks used in computer science. **I/O Bound:** Small bursts of CPU activity with I/O user interaction; these have highest priority (e.g., word processor). > More threads $\implies$ More idling $\implies$ Waste of resources.

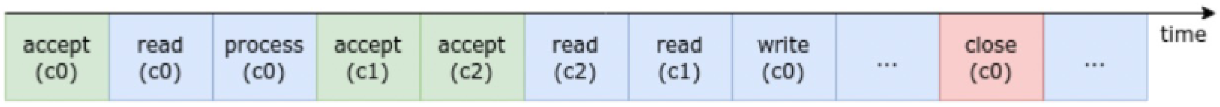
  CPU Bound: Mostly CPU activity (e.g., gcc, 3D rendering). Has long CPU bursts/processes, and lower priority. CPU tasks can be sped up with **multi-threading**: > More CPU power $\implies$ process finishing sooner.

APIs are I/O bound! When a server has multiple client connections, we don't want subsequent clients to wait for the first thread to completely finish. In the image below, there is too much idling time between the three threads (more CPU power that isn't being used).



Note this is not a HTTP server, since it waits for a client rather than having open-ended communication between client and server.

We can "compress" the events using only one CPU in an **Event Loop**:



This way, we can do more work with the same CPU power as above (and in ==one thread==). The CPU takes control from the idling task and gives it to another task that needs it.

All the events (or tasks/callbacks/etc.) get queued (**event queue**), then popped into the **event loop** with a heap and stack. Tasks are executed in the **thread pool**, then are pushed back to the end of the queue.

```
function event_loop():
    while (true):
        # Get first task in queue
        current_task = task_queue.pop(0)

        # Execute task in the same thread
        result = execute(current_task)

        if (result != complete):
            # If blocked by I/O or a different I/O
            # task, push it to end of queue.
            task_queue.append(current_task)
```

This technique allowing responsive tasks across multiple threads is called **asynchronous programming**.

Asynchronous Programming

Async programming is not naturally available in many languages, and traditionally, **callbacks** and **promises** are used as workarounds.

- **Callbacks -** Not very clean; many nested loops and if-statements.
- **Promises -** Similar to callbacks, but does not use nested logic. However, the subsequent logic is in `then` clauses, making idling still required (and also still unclean code).

Async Functions are available in JavaScript, Python (3.4+), Go, etc. They have the exact same flow as sync programming where I/O blocking tasks wait (`await`), but the rest is handled by the interpreter, event loop, etc.

```
async function handler(req, res) {
    try {
        // Await propagates throughout entire function (all other vars must have await)
        const apiResponse = await callAPI(...)
        const readResponse = await readDB(...)
    } catch (error) {
        // failure callback
    }
}
```

Suppose `apiResponse` does not have `await`: Then, `readResponse` will not wait for `apiResponse` to finish when executing.

Model View Controller (MVC)

Controller The API handler logic, Next.js frameworks, etc.

- Request handling, interaction with client, querying database, etc. **View -** Frontend components.
- HTML, CSS, client-side JS.

Every web application needs a **persistent storage** (i.e., data persistence). There are many databases:

- **Relational:** Postgres, MySQL
- **Non-Relational:** Cassandra, MongoDB Node.js supports multiple database backends.

Object Relational Mapper (ORM)

ORM allows us to convert data between a relational database and the heap, providing an abstraction over the underlying database queries.

- Method/attribute accesses are translated to queries.
- Results are wrapped by objects/attributes.

ORMs have many advantages!

1. **Simple -** Does not need SQL syntax.
2. **Consistent -** Everything is in JS.
3. Can switch between databases easily.
4. Enables OOP.
5. Runs secure and efficient queries (e.g., SQL injection). Raw queries might still be preferred if you need very efficient databases.

SQLite A lightweight database in standard SQL format, storing everything into one file!

```
sqlite3  # Will install SQLite

## Opens a new SQL shell sqlite>
```

It follows standard SQL syntax and is good for personal development, but is not very ideal (not good for production).

Prisma This courses uses **Prisma** for ORMs, which is simple and very popular!

```
## Install
npm i prisma @prisma/client @prisma/studio


## Run
npx prisma init

## Creates folder prisma and file schema.prisma

## Note prisma files are not JS nor SQL files
```

Prisma generates JS classes from the schema, and **syncs** it with the database schema.

```
## Turn prisma code into JS classes
npx prisma generate


## Sync schema with database
npx prisma migrate dev

## Will add migrations and db folders
```

To view the database, run `npx prisma studio` and access `localhost:5555`; creates a visual tool to browse table(s) and modify data.

Models

The data **management** logic; how data is defined, what fields there are, how it is stored in DB. Typically defined as classes; the ORM maps it to a table in the database.

Model Design - Must be done **before** coding starts, and is independent of programming languages and frameworks. User data is the most sensitive part of your application; make sure you design secure models! **Example:** A news application has the models 'user', 'reporter', 'comment', 'report', etc.

You can use either **class** or **ER** diagrams.

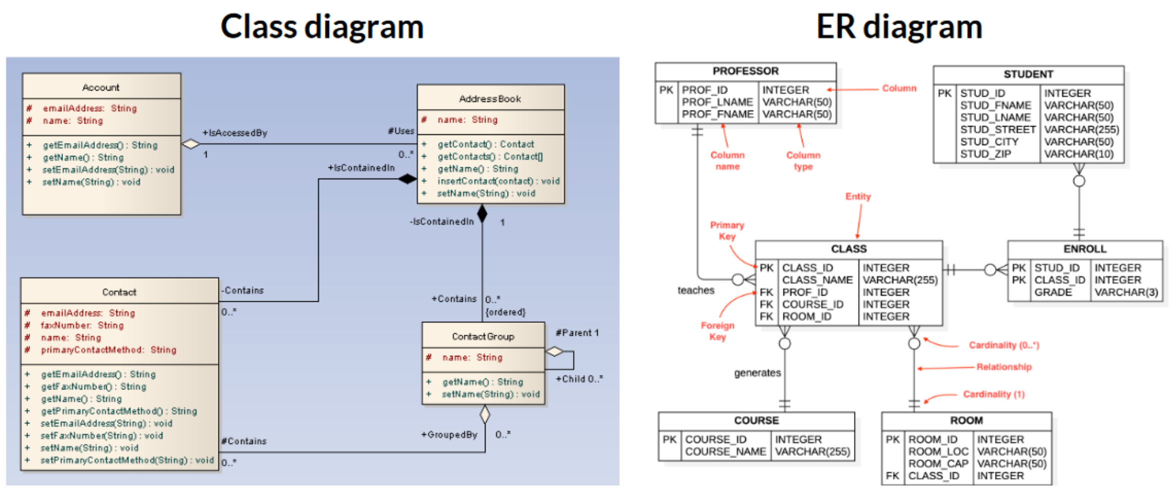


Figure 2: Pasted image 20240928211713.png

Using Schemas (in Prisma) **Data Source -** Type and address of the database.

- Provider: `sqlite`, `mysql`, `postgresql`, etc.
- URL: File/server address with credentials. Define one `model` for each model in the ER/class diagram. Then, add fields from the diagrams and map it to the "database" column by the ORM.

Fields - Attributes of a model.

```
-- Field attributes
@id, @default, @unique, @map, @index, ...
```

```
-- Model attributes
@@unique, @@map, ...

-- Scalar Types
String, Booolean, Int, BigInt, Float, Decimal, DateTime, Json, Bytes, Unsupported
```

Nullable fields have the `?` symbol. Default values are preferred over null values (since it introduces issues like type complexity). Null has a dependent use case, e.g., when the default case is very different from null (0 vs. null).

ID (Primary Key) - Used for unique constraints (in unique data), where a value will be independent from other data in a row. It is encouraged to use a separate, automatic field for id (e.g., auto-incrementing integer or Universally Unique Identifier (UUID)). Changing the primary key is almost impossible!

Relations (`@relation`) - Describes the relationships between entries/tables in a database; considered a **foreign key**. Typically uses one-to-one or many-to-many relations. Relations also define a column storing the id of the referenced model.

```
-- Example:
categoryId Int
category   Category @relation("CategoryProduct", fields: [categoryId], references: [id])
```

Reverse traversal can be done by a field in the **original** model:

```
product Product[] @relation ("CategoryProduct")
```

- **One-to-One:** Similar to one-to-many relations; marks the foreign key column with `@unique`.
- **Many-to-Many:** Defines an array at each end, turning into a **separate table** by the ORM.

5 - CRUD

Recap: Prisma

1. Install via `npm i prisma @prisma/client @prisma/studio`.
2. Create a file named `schema.prisma`.
3. Prisma generates JS classes from the schema file, syncing it with the database schema.
    1. Schema doesn't automatically impact anything. Generate the relevant JS classes: `npx prisma generate`. Sync schema with database: `npx prisma migrate dev`.

CRUD

CRUD - Create, Read, Update, Delete. Runtime database operations are either **data access level** or **data manipulation level** queries.

Prisma with Next.js

Define a client **instance** (in a separate file):

```
import { PrismaClient } from '@prisma/client'
export const prisma = new PrismaClient()
```

Then, **import** the instance in the API handlers.

```
// Asynchronous Functions
prisma.<MODEL_NAME>.<OPERATION_NAME>

// Examples:
prisma.product.findMany(...)
prisma.user.create(...)
prisma.category.update(...)
```

Translated into SQL/NoSQL queries by the ORM; it pours the results into JS objects.

Queries All queries have definable conditions on related objects within the `where` clause. `select` and `include` customize which fields should be present in the returned object(s).

- `select`: Only selects specified columns and/or related objects.
- `include`: Includes specified related objects on top of existing columns. Select:
- `where`: Filters
- `include`: Related objects to fetch; translates to `JOIN`.
- Alternate methods: `findFirst`, `findUnique`, `findUniqueOrThrow`.

```
const products = await prisma.product.findMany({
    where: {
        name: {
            contains: req.query.productName
        },
        price: {
            lt: 3000,
        },
        isAvailable: true,
    },
    include: {
        store: true,
    }
})
```

Insert:

- `data`: Column values
- Related fields: `create`, `connect`, or `conectorCreate`.
- Returns the created object; can be used in API responses.

```
await prisma.store.create({
    data: {
        name: req.body.name,
        description: req.body.description,
        address,
        owner: {
            create: {
                username: generateRandomString(10),
                firstName: 'first name',
                lastName: 'surname'
            },
        },
        sector: {
            connect: {
                id: sectorId,
            },
        },
    },
})
```

Update/Delete:

```
await prisma.user.update({
    where: { id: 'test-user' },
    data: { password: hashPassword(newPassword) },
})

await prisma.product.delete({
    where: {
        id: productId,
        storeId: req.store.id,
    },
})
```

- `update` and `delete` throw an **exception** if the `where` clause doesn't match to **exactly** one record.

Validation

Request data should be thoroughly validated:

- Required fields must exist.
- Field types (e.g., number/string) must match.
- Specific formats (e.g., email, phone number) must be checked. Never trust the user or frontend, since there could be malicious clients. JS cannot do any type enforcement or validation.
- Run validations before executing the database queries. Return a 400 response for malicious/invalid data. Running queries with unclean data is dangerous:
- Security vulnerabilities may be stored.
- Can cause crashed and a 500 response with unhelpful errors.

6 - Auth and Migration

Auth refers to both authentication and authorization.

Authentication - Obtains user information from username and password, session, API key, fingerprints, etc.

- Who is calling? Is it really the person they say they are?

Authorization - Checks user's properties and permissions.

- Does this person have enough access and permissions (aka authorized) to make this request?

Common Auths

Basic Auths `signup.js` is often used instead of `index.js` since it is a unique feature and should be treated in a special manner.

When we use the `POST` API and we don't want to return everything back to the user (e.g., the password), use `select`:

```
await prisma.user.create({
    data: {
        username,
        password,
        email
    },
    select: {
        username: true,
        email: true
    }
})
```

We don't want to store password in the database (security issue), we can use a package like `bcrypt`. `@/utils/auth.js`:

```
import bcrypt from "bcrypt";

const BCRYPT_SALT_ROUNDS = 10;

export async function hashPassword(password) {
    return await bcrypt.hash(password, BCRYPT_SALT_ROUNDS);
}
```

```
export async function comparePassword(password, hash) {
    return await bcrypt.compare(password, hash);
}
```

Then, we can change our object creation:

```
data: {
    password: await hashPassword(password)
}
```

The **basic auth** process involves sending credentials at **every request**; it has no concept of login and logout (insecure! transfers raw sensitive data many times). As a result, it is not used in modern applications.

Session Auth Instead of continuously sending requests, the client only sends their username/password at **login**. If successful, the server creates and stores a **session id**, mapped to the user. The session id is returned in the response (where the browser saves it in **cookies**). Then, the browser sends the same session id at future requests.

- Incognito/private tabs: Browser never sends the same session id.

Token Auth Instead of a **random** session id, the token contains information about the user. It can be a JSON string e.g., `{ "user": "123", "expires": 20240120 }`. This must be signed by the server to avoid attacks, which is a key that is turned into an encrypted random string. Commonly called a JWT (JSON web token).

- You can install a package implementing this two-way hash, `jsonwebtoken`. JWTs are generally safe (can't construct a counterfeit token), but tokens can be compromised.

```
import jwt from "jsonwebtoken";

const JWT_SECRET = "randomsecuretoken";
const JWT_EXPIRES_IN = "1h";

export function generateToken(obj) {
    return jwt.sign(obj, JWT_SECRET, {
    expiresIn: JWT_EXPIRES_IN});
}

export function verifyToken(token) {
    if (!token?.startswith("Bearer ")) { return null; }

    token = token.split(" ")[1];
    try {
        return jwt.verify(token, JWT_SECRET);
    } catch (error) {
        return null;
    }
}
```

Tokens typically have prefixes in their keys to add minor protection from secret scanners.

Session vs. Token **Session Auth**

- Less scalable
  - Server stores *all* sessions.
  - Has an additional database query per request.

- More control
  - Server can revoke sessions.

Token Auth

- Simpler
  - No database interactions.
- More scalable
  - Client is in charge of storing the token.
- Less control
  - Not revokable; does not have a true logout.

Creating a Login Page

In our `@/pages/api/users/` directory, we can create a `login.js` file:

```
import { comparePassword, generateToken }

export default async function handler(req, res) {
    const {  } = req.body;

    const user = await prisma.user.findUnique({
    where:{ username }});

    if (!user || !await comparePassword(password, user.password)) {
        return res.status(401).json({
        message: "Invalid credentials"});
    }  // 401 = unauthorized

    const token = generateToken({ userId: user.id, username: user.username });

    return res.status(200).json({
    token,
    });
}

export default function handler(req, res) {
    const user = verifyToken(req.header.authorization);
    ...
}
```

Obviously, if anyone has access to your code, they can see your secret and codes to these encryptions. As a result, we send the sensitive information (e.g., JWT secrets) to a separate file `.env`, which is in shell syntax. Then, instead of the variables in our `auth.js` file, we write `const JWT_SECRET = parseInt(process.env.JWT_SECRET);` and similar variables.

Token Types **Short-Lived Tokens -** Access tokens should expire within 15 minutes to an hour.

- This is bad UX...

Refresh Tokens - Signs using a different secret token, and can only be used to generate new access tokens regularly.

- Lasts much longer (hours to weeks).
- User only needs to reauthenticate when the token expires.
- When receiving a 401 unauthorized response: **refresh** the token and **resend** with a new access token.

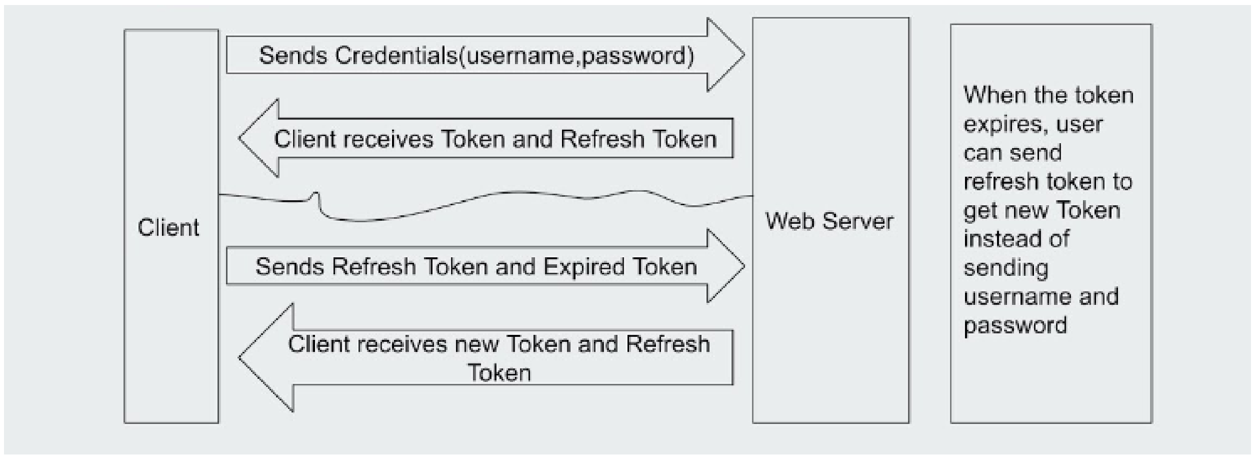

Client
Sends Credentials(username,password)
Client receives Token and Refresh Token
Sends Refresh Token and Expired Token
Client receives new Token and Refresh Token
Web Server
When the token expires, user can send refresh token to get new Token instead of sending username and password

Figure 3: Pasted image 20241016124020.png

Authorization Check several conditions before executing any API handler logic:

- Is the user authenticated?
- Does the user have enough **access**? If not, return a **403 Unauthorized** code. Authorization includes reusable logic, ideally separate from the handler logic (i.e., in middlewares).

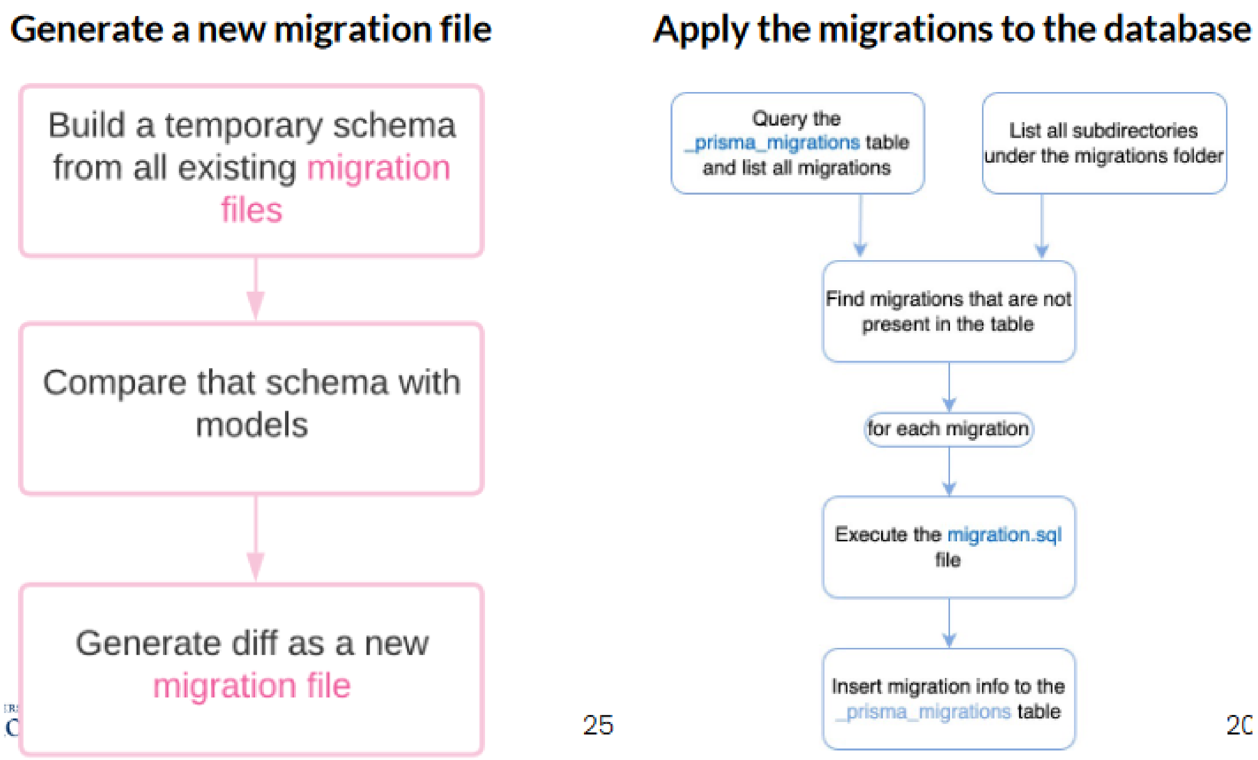
Migrations

The state of database tables is the same as what is defined in the model schema, but they are **totally independent**! The **ORM** applies the application's schema to the database via **DDL** queries.

A schema's state changes with:

- Creation/removal of a table or model.
- Creation/removal of a column or field.
- Modification of field options or attributes. Whenever a state changes, the database *should* migrate it to the new state (Prisma does not do this automatically). If the ORM and database schema do not match, there will be a database exception.

Migration Workflow


Generate a new migration file
Build a temporary schema from all existing migration files
Compare that schema with models
Generate diff as a new migration file
25
Apply the migrations to the database
Query the _prisma_migrations table and list all migrations
List all subdirectories under the migrations folder
Find migrations that are not present in the table
for each migration
Execute the migration.sql file
Insert migration info to the _prisma_migrations table
20

Any changes since the last migration are stored in the `migrations` folder as SQL files.

New Migration Creates a new folder inside the `migrations` dir with the DDL queries.

- Logs will show what has changed since the last commit (migration).
- Builds the old database state from previous migrations; does not contact the database.
- Iterates over all models to find differences. Migrations are created and applied via `npx prisma migrate dev` and are stored in the database. Migrations should not be applied twice (i.e., do not use `CREATE TABLE` again).

Migrations Table Migrations are stored in the `_prisma_migrations` table with its **metadata**. The content is only stored in the migration file.

```
PRAGMA table_info(_prisma_migrations);
0|id|TEXT|1||1
1|checksum|TEXT|1|0
...
```

`checksum` ensures that migrations are not edited.

Migration Assumptions

- Never directly change the database tables (e.g., `ALTER TABLE`).
- Never edit/delete a migration file. Migration files must be the same everywhere; always push these files to `git`.

```
## Generates JS code of schema
npx prisma generate
```

```
## Identifies changes since last migration, generating a new one. Also applies unapplied migrations.
npx prisma migrate dev

## Applies unapplied migrations w/o creating new ones. Used in production
npx prisma migrate deploy
```

Migration Errors Same/conflicting migrations, manual updates to database tables, failing migrations, or migration file edits can all cause errors. Potential for data loss is very high with migration errors!

Resolving Migrations:

1. Update the migrations table without executing the queries.

```
npx prisma migrate resolve --applied "migration_name"
npx prisma migrate resolve --rolled-back "migration_name"
```

2. Manually sync the database schema with migrations.
3. Reset the entire database using `npx prisma db reset`. This deletes all of the table's data and applies migrations on an empty database. Do **not** use this in production!
4. Delete the entire database `dev.db` file, then the migrations directory. Restart with a fresh schema and new migrations. Do not use this in production!

7 - React

Traditionally, backend servers listen for HTTP requests, which come from the browser:

- GET requests by entering a URL or clicking on a link.
- POST requests by filling out forms.
- Typically request a specific page. Then, the server returns a HTTP response with HTML body, and the browser renders the HTML page.

In **modern web applications**, the requests come from the browser, but also mobile apps, postman, etc.:

- Typically request a specific CRUD operation.
- GET requests for queries, POST for data manipulation. Then, the server returns a HTTP response with a JSON body. The client will process the response accordingly.

In this course, we focus on **web clients** (sending requests through web browsers).

Single Page Applications

Single-Page Applications - When JavaScript is used to make changes to the webpage.

- Seamless user experience
  - No reloads, no refreshes.
  - Everything does not get reset every time.
  - More control over the UX design.
- Efficient
  - The whole page does not get updated.
- Faster load time
  - The initial load (when nothing is there) takes less time.

Single page applications use a technology called **Asynchronous JavaScript and XML (Ajax)**. The browser sends the request in the background. It does not block the main thread, and further changes are made to the document.

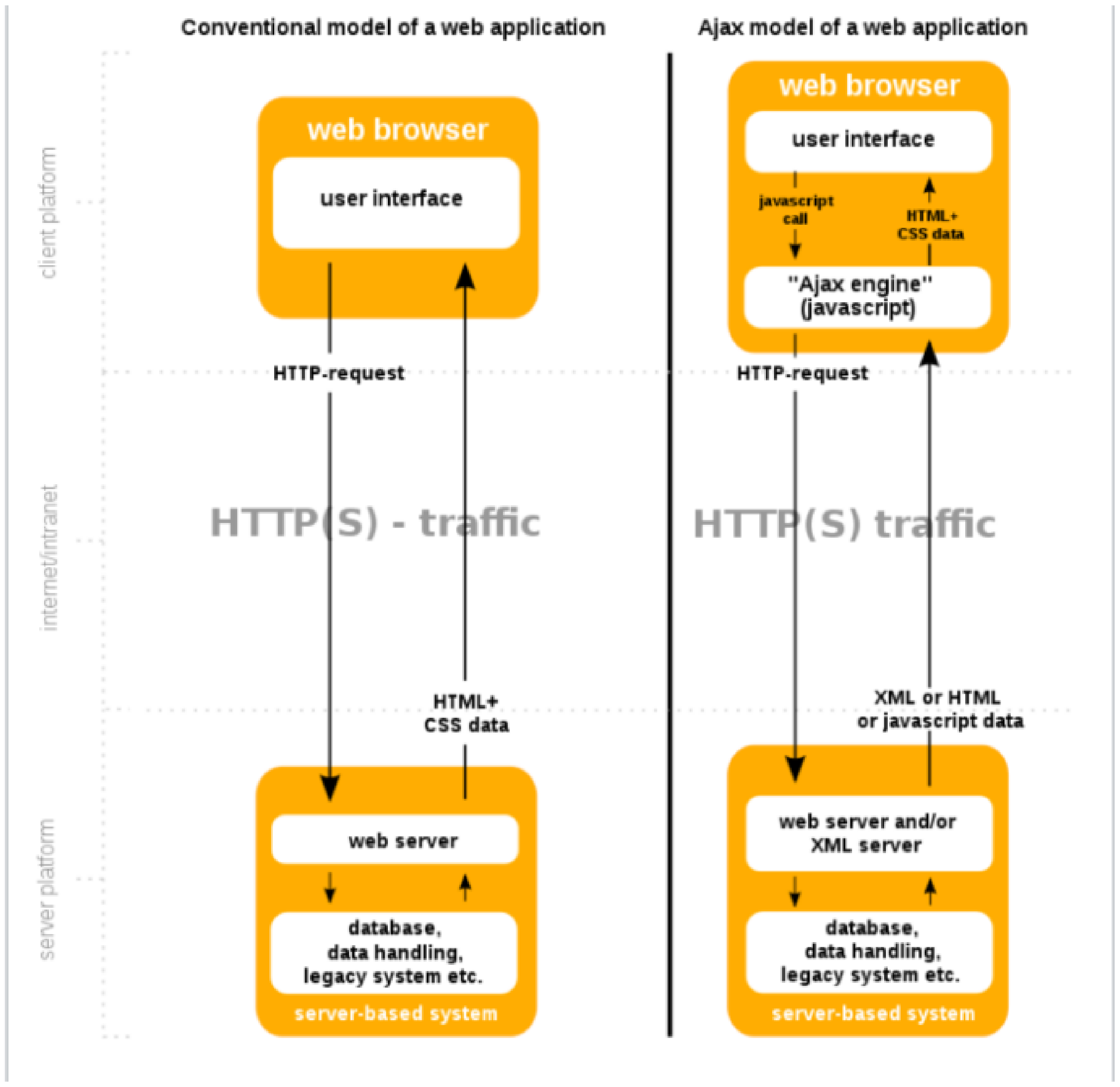


Figure 4: Pasted image 20241105223813.png

Rather than using pure Ajax, we use many frameworks. This also allows for us to separate backend and frontend! **Examples:** React, Angular, Vue

React

React is a JS library for building interactive user interfaces, taking charge of **re-rendering** when something changes. Therefore, you no longer need to manipulate elements manually.

- Creates a virtual DOM in memory.
- When something changes, React re-renders its own DOM.
- Compares the new and old DOMs and finds out what has been updated.
- Updates the specific elements of the browser's DOM. Updating and re-rendering the actual DOM is expensive, so React only changes what really needs to be changed.

JSX React uses a special variation of JS that allows for merging HTML and JS together, e.g., `const element = <h1>Hello, world!</h1>`. Browsers do not understand this syntax, so they should be translated before execution.

Example:

```
const element = <span className="red">Hello, world!</span>
const name = "Hello world";
const id = "div-1"

const element2 = (
    <p>
        <div id={id}>
            Hi, there is a {name} here!
        </div>
    </p>
)
```

as opposed to

```
const element = /*#__PURE__*/React.createElement("span", {
    className: "red"
}, "Hello, world!");
const name = "Hello  world";
const id = "div-1";
const element2 = /*#__PURE__*/React.createElement("p", null, /*__PURE__*/React.createElement("div", {
    id: id
}, "Hi, there is a ", name, " here!"));
// Note that React.createElement does NOT create a JS element, it is its own React element.
```

We can import React and Babel (JSX) scripts into your HTML:

```
<script src="https://unpkg.com/react/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-dom.production.min.js"></script>
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

We can then render elements in an actual JS element:

```
<script type="text/babel">
    const element = <h1>Hello World!</h1>;
    ReactDOM.render(element, document.body)
</script>
```

Components Components are functions or classes that return a React element.

- Can be re-used like a known tag; makes elements easily reusable.

```
function sayHello() {
    return <h1>Hello world!</h1>;
}

ReactDOM.render(<SayHello />, document.getElementById("root"));
```

- You can put any JS statement inside the `{}` in JSX.
- Singular tags must *always* end with `/>`.
- Components' names should always be capitalized.
  - Lowercase names are reserved for built-in elements (e.g., `h1`).
- JSX elements must be wrapped in one enclosing tag; if more than one, wrap them in a **React fragment**:

```
<>
    <p>
        <div id={id}>
            Hi, there is a {name} here!
        </div>
    </p>
    <img src="http://test.com/img.jpg" />
</>
```

Props Props are **read-only** data coming from the parent element.

- React uses this to mimic JS attributes.

```
function Text(props) {
    return <h4>{props.value}</h4>
}

<Text value="John" />
```

Note that styles and classes are handled a bit differently in JSX:

```
function Text(props) {
    return(
        <h4 className="text" style={{fontSize: props.size}}>
            {props.value}
        </h4>
    )
}
// Can be simplified!
function Text({ size, value }) {
    return (
        <h4 className="text" style={{fontSize: size}}>
            {value}
        </h4>
    )
}

<Text value="Cars" size={30} />
```

We can have more sophisticated props:

```
function List({ title, values }) {
    return (
        <>
            <Text value={title} size={40} />
            <ul>
                {values.map((item, index) => (
                    <li key={index}>
                        {item}
                    </li>
                ))}
            </ul>
        </>
    )
}
```

- Elements created in a loop must have a **unique key prop**.
  - This identifies which item has changed, is added, or is removed. Otherwise, React will have to re-render the whole list if something changes.

Paired Tags Components can be used as a **paired tag**. What is put inside tags will be passed on to child props.

```
function Wrapper({ children }) {
    return <div className="col">
             { children }
           </div>;
}

const wrapped = (
    <Wrapper>
        <List values={[1, 2, 3, "my cat"]} </>
    </Wrapper>
)
```

Class Components Extends `React.Component` and should implement the `render` method. Props are passed to the constructor.

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}</h1>;
    }
}
```

State **State -** An object initialized in the constructor. All components have a built-in state. Once the state changes, the component re-renders.

```
// Initialize state object in constructor
class Counter extends React.Component {
    constructor(props) {
    super(props)
    this.state = { counter: 0 }
    }
}

// Access state values via this.state
```

```
render() {
    return <h3>{this.state.counter}</h3>
}
```

React states should never be mutated (breaks the underlying assumptions of React). So, to update the state, call the `setState` method. Other approaches will *not* trigger the re-render.

- Never assign the state other than in the constructor!

```
this.setState({
    counter: this.state.counter + 1
})

// Avoid the code below!!!
this.state = {
    counter: this.state.counter + 1
}
```

> [!NOTE] Lifting the State To pass a shared state between components, move it to their **common ancestor**. - Define the state in the common ancestor; - Pass it as props to the original components; then - Pass a setter function as the change handler.

Events React uses the same set of events as plain JS.

- use camelCase for React events, e.g., `onClick`.
- The action must be a function, not a statement.
  - `onClick = {() => alert()}`

Each JS function has its own `this` which is the caller object. The object that *calls* the event handler is **not** the component object! Arrow functions are the easiest solution to this, since they capture `this` from the outer scope while the class body contains the proper `this`.

```
increment = () => {
    this.setState({count: this.state.count + 1});
}

render() {
return (
    <div>
        <button onClick={this.increment}>+</button>
    </div>
)
}
```

We can store and use an input's value:

- Add it to the state.
- Read it from the state as well. Then, we read the new value from `event.target.value`.

```
<input
    type="text"
    value={this.state.celcius}
    onChange={event => this.setState({
                        ...this.state,
                        celsius: event.target.value
})}/>
```

8 - Monorepo and Hooks

Rather than using React in HTML files, we can use it as the front-end code from a Node server.

- When the browser requests a URL, a series of HTML, CSS, and JS files are returned, containing compiled JS components. Then, a pre-compiled and bundled build for production.

Monorepo

Next.js is both a frontend and backend network, aka a monorepo. **Monorepo -** The practice of having all code in **one** repository.

- Different node projects (apps and shared packages) in one parent Node project. Giant codebases like Google or Facebook follow this practice.

Benefits:

- Does not deal with repo versions.
- Can share types, utils, and libraries across the repo.
- Project is self-contained and easy to navigate.

Next.js is actually a **monolith**, where all the code is in one Node project (rather than multiple).

- Better for small projects; migrate to monorepo if the project gets bigger.

React in Next.js

Within the `pages` directory, every path outside of `/api` is a **front-end path** (unless it starts with an `_`).

- Define a React component in each file and make it the **default export**. This will render when the path is accessed from the browser. React will create the HTML and compile the JSX for you. Additionally, you can import styles and assets to your JS modules to be handled and served by the server.

File Structure

- Separate pages from components.
  - You don't want files to be associated with paths.
- Create a `components` directory.
  - Create subdirectories for each component as well.
  - Contains the main components `index.js` and any child components.
- Always have some re-usable base components (e.g., inputs, forms, headings).
- Importing CSS files: `import "@/styles/globals.css"`.
- Images and other static files gather under the `public` directory.
- Components shouldn't be too big!
  - Have nested, child components.

Hooks

Hooks are "syntax sugars" in React that write verbose classes, constructors, and `setState` for you.

- As a result, you can move back to function components.

useState

- The state does not have to be one object with hooks.
- Define separate state variables using the `useState` hook.
- Returns the variable and update function.
- Component gets re-rendered when the value changes.

```
import React, { useState } from 'react';

const Status = (props) => {
    const [status, setStatus] = useState("good");
    const toggleStatus = () => {
        setStatus(status === "good" ? "bad" : "good")
    }

    return (
        <>
            <h3>Situation is {status}</h3>
            <button onClick={toggleStatus}>Toggle</button>
    )
        </>
}
```

Benefits:

- Is one function component rather than verbose class components.
- Enables multiple state variables.
- Does not use `this` and does not use method binding.
- Easy to share state with child elements; each state variable has its own setter.

Lifecycle

Rather than only running code when `render()` is called, we can add custom **lifecycles.**

- Examples: Sending a request upon load, accessing state values. In class components, we have lifecycles like `componentWillMount()`, `componentDidMount()`, `componentWillUnmount()`, etc.

Benefits: #TODO

useEffect `useEffect` is a powerful hook that replaces lifecycle functions.

- Is called when a component **mounts**, or when something changes.

```
import React, { useState, useEffect } from 'react';

useEffect(() => {
    console.log("Called when a component mounts.")
}, []);

// When any elements of the array change, the effect is invoked.
useEffect(() => {
    console.log("Props size or status has changed")
}, [status, props.length]);
```

- It is recommended to have a separate `useEffect` for different concerns.

- Do not leave out the second argument `[]`.
  - The effect would run at every re-render, making it inefficient.
- The array should include all variables that are used in the effect, or it may use stale values at re-renders.

API Calls and Hooks

Fetch API - The interface for browsers to send HTTP requests. Returns a promise.

```
let request = await fetch('/account/login/', {
    method: 'POST',
    data: {username: 'Bob', password: '123'}
})
const data = await response.json()
console.log(data)
```

```
// Example: Fetching data on page load and adding it to the state.
const [holidays, setHolidays] = useState([]);

const fetchHolidays = async () => {
    const response = await fetch("https://canada-holidays.com/api/v1/holidays");
    const data = await response.json();
    setHolidays(data.holidays);
}

useEffect(() => {
    fetchHolidays()
}, []);
```

> [!NOTE] Pagination Most times, GET APIs do not return all response at once. Instead, they send results in pages!

Authentication

API Auth First Party Authentication:

- Stores access/refresh tokens in a client's persistent storage.
- Should not be deleted when the tab/browser/computer is closed. Web browsers use `localStorage`:

```
localStorage.setItem('access_token', access_token);
localStorage.getItem('access_token');
```

Third-Party Authentication:

- Used when contacting external APIs (e.g., maps, weather).
- Authentication is different; the entire app is a client to that third-party system.
  - End-user cannot login to that system. Solution: API keys! They are either permanent or are very long lived (months/years).

CORS Having the front-end contacting third-party APIs is very dangerous.

- Clients will have access to the API key.
- Can cause significant security or financial issues. **Cross-Origin Resource Sharing (CORS) -** Standard that allows web pages to access resources from a different domain.
- A client should only request to URLS with the same domain.

- Browser blocks you from fetching a different domain.
- API servers also block requests coming from a browser.

Solution: Backend Proxies!

- Implement a backend API that requests the third-party service and returns the response. Advantages:
- API key is not exposed.
- More control over what data is transferred.
- More control over who accesses the data.
- Better logging and monitoring.

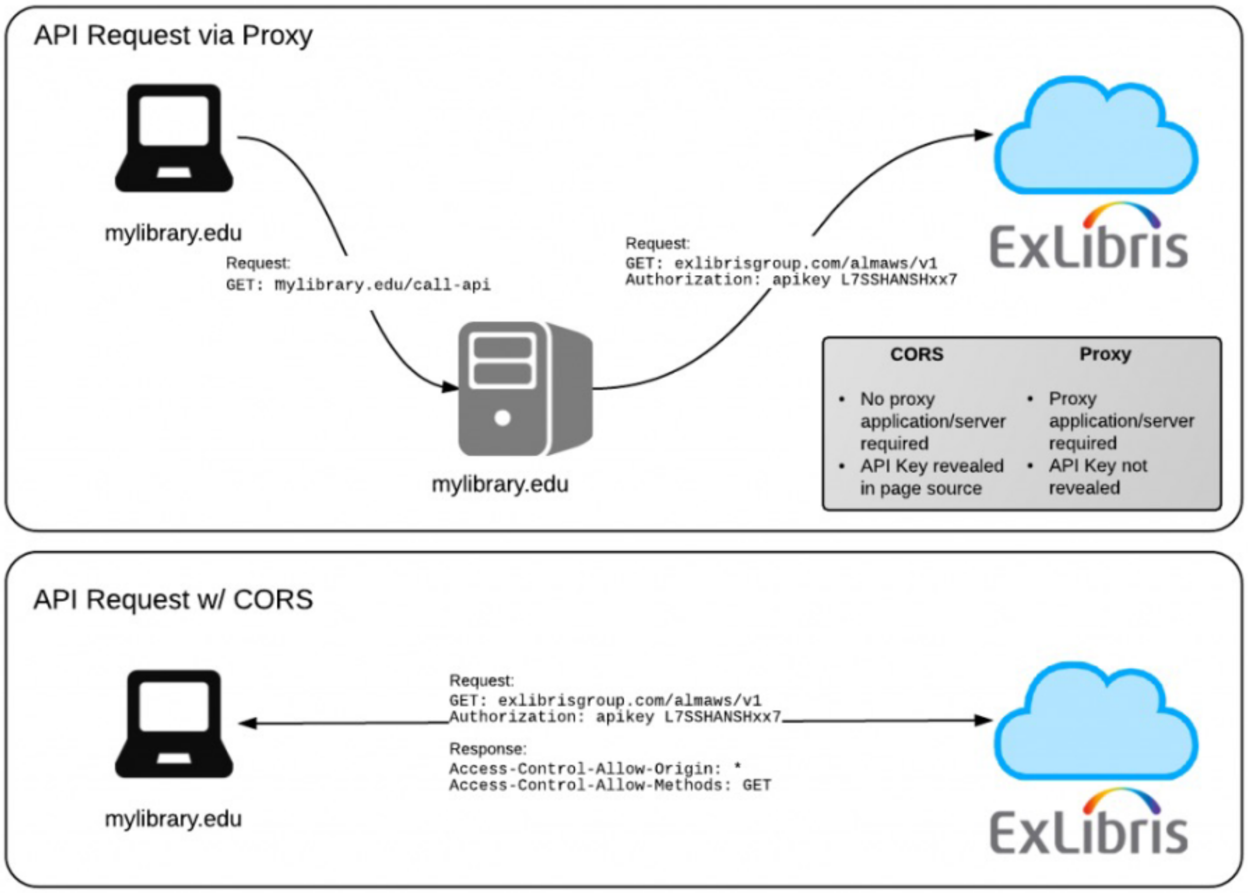

API Request via Proxy
mylibrary.edu
Request:
GET: mylibrary.edu/call-api
mylibrary.edu
Request:
GET: exlibrisgroup.com/almaws/v1
Authorization: apikey L7SSHANSHxx7
ExLibris
CORS
• No proxy application/server required
• API Key revealed in page source
Proxy
• Proxy application/server required
• API Key not revealed
API Request w/ CORS
mylibrary.edu
Request:
GET: exlibrisgroup.com/almaws/v1
Authorization: apikey L7SSHANSHxx7
Response:
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET
ExLibris

Figure 5: Pasted image 20241106152007.png

OAuth Being able to sign in via Google, Facebook, etc.

- Redirects the client to external login. If successful, redirects back with the auth code.
- Server contacts Google with the auth code and API key to receive user info.
- Server creates an account and generates an access token for the client.

9 - TypeScript and Advanced CSS

Navigation

Sometimes you may need a URL change via code.

- Example: For a 401 response, redirect to the login page. Is similar to `window.location.replace()` in regular JS.

We can instead use the Next router:

```
let router = useRouter();
router.push("/login");
```

Defining Navigation

Arguments: Parameters can be both defined as URL args (part of the path) or query params (key-value pairs added after ? in the URL).

- URL args defined in the file name (e.g., `[storeId].jsx`).
- Can be accessed via `router.query`: `const { storeId } = router.query;`.

Links: Similar to `<a>` tag, but does not reload the browser.

- Usage: `<Link href="/watch">watch</Link>`.
- Import Link and Router from Next (not React!):
  - `import Link from 'next/link;`, and `import { useRouter } from 'next/router`.

Global States and Context **Global State:** Set of local states concurrent with each other (no cause-and-effect relationship).

- Alternative to **prop drilling** (passing down the state and its setter functions to its children).
- Accessible everywhere (does not pass things all the way down).
- Do not use them for everything; makes code dirty and harder to understand; makes component re-use harder.

Context: How to handle **global** states with React.

- Create state variables and/or setters in a context. Everything inside the context is accessible within its **provider**.

```
export const testContext = createContext({
    var1: null, var2: null,
});
```

- Put a default initial value for every variable that you will include in your context.

Provider:

1. Create an object `const myObject = { var1: 1, var2: 2};`
2. Put a provider around the parent component and pass the object: `<TestContext.Provider value={myObject} />`
3. At any descendent, you can access the context object: `const {var1, var2} = useContext(testContext)`

Benefits of Context:

- Useful for storing data used by many components, or is set and read in different components (e.g., account info, profile data).

  - Create one context for each set of relevant variables and their setters.

```
%% Example %%
export function useAPIContext() {
    const [deployment, setDeployment] = useState([]);
    const [servers, setServers] = useState([]);

    return {
        deployment, setDeployment,
        servers, setServers,
    };
}

%% Then... %%
ReactDOM.render(
    <React.StrictMode>
        <APIContext.Provider value={useAPIContext()}>
            <ControlPanel />
        </APIContext.Provider>
    </React.StrictMode>,
    document.getElementById('root')
);
```

Type System

Static vs. Dynamic Typing:

- **Static -** Types are checked at compile-time (e.g., C, Java).
- **Dynamic -** Types are checked at **runtime**; you can change a variable's type (e.g., Python).

Strong vs. Weak Typing:

- **Strong -** Enforces strict rules about how types are used and combined (e.g., Java, Python).
- **Weak -** Flexible about type conversions, leading to **implicit type coercion**.

JS Types (recap): number, string, boolean, object, function, undefined. JS is **dynamically** and **weakly** typed.

- Can reassign variables to different types.
- Automatically **converts** types in unexpected ways to avoid crashing.
  - `1 + "2"` is `"12"`
  - `5 == "5"` is true
  - `"0" == false` is true
  - `[] + []` is `""`
  - `[] + {}` is `[object Object]` JS is very error prone! Having no typing makes code **unreadable** for other developers (and sometimes yourself). It also becomes a mess and attracts bugs.

TypeScript

- Invented by Microsoft in 2012.
- A **superset** of JavaScript - adds typing to the language.
- Has no runtime effect! Will compile to JS.

Adding TypeScript to Next.js Projects

1. Create an empty file named `tsconfig.json` and restart the server.
2. Next.js will automatically fill the file with existing configs from `jsconfig.json`.
3. Rename files from `js/jsx` to `ts/tsx`. You can add TS to all Node projects (not just Next.js projects)!

TS Characteristics

TypeScript has many benefits!

- Improved code quality.
- Catch errors during development, not at runtime.
- Better collaboration in large teams with clear types.
- Better tooling and autocompletion in IDEs.

Statically Typed:

```
let name = 'Alice'
name = 42  // Error: Type 'number' is not assignable to type 'string'
```

Strongly Typed:

```
const num = 10
const str = '20'
const result = num + str
// Error: Operator '+' cannot be applied to types 'number' and 'string'
```

Syntax:

- Type declaration:

  ```
  let message: string = "Hello, Typescript!"
  function greet(name: string): string {
      return `Hello, ${name}`
  }
  ```

- Type inference also works: `let count = 42`

System:

- Primitive Types:
  - JS primitive types
  - PLUS additional types: `any, unknown, void, never`.
  - Array and tuple: `number[], [string, number]`
  - Enums
  - Optionals: `function greet(name?: string)`
- Type Aliases: `type ID = string | number`
- Interfaces and Generic Types:

  ```
  interface ModalProps {
      text: string
      image?: string
      autoHide: boolean
  }

  const Modal: React.FC<ModalProps> = (props) => {
      const [loading, setLoading] = useState<boolean>(false)
  }
  ```

Other Notes

- TypeScript works alongside JS; all files do not have to be converted to TypeScript.
- **Suppression:**
  - The `any` type is a wildcard that suppresses type checking.
  - Use `@ts-ignore` to disable TS for a line; discouraged! Only use if necessary.
- All checks are done at compile time; there is no impact on runtime!
  - Runtime is again in plain JS.
- To check types at **runtime**, use `typeof` (JS types) and `instanceof` (JS classes).

Advanced CSS

Traditional CSS has many issues:

- CSS bloat - Files grow very big with many unused styles.
- Specificity war - Overly complex rules for CSS precedence.
- CSS frameworks (e.g., Bootstrap, Material) - Leads to websites looking similar.
- Context switching between JS and CSS files.

Tailwind CSS

Tailwind replaces CSS with utility classes! Each class corresponds to a CSS style for an element.

```
<button className="bg-blue-500 text-white py-2 px-4 rounded"> Click Me </button>
```

Installation:

1. Install via `npm install tailwindcss postcss autoprefixer`.
2. Run `npx tailwindcss init -p` to generate config files.
   - Generates `tailwind.config.js` and `postcss.config.js`.
3. Add all JS/TS file **globs** (literal/wildcard filepaths) to `tailwind.config.js`..
4. Add the following lines to `globals.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Power of Tailwind:

- Arbitrary values (e.g., `w-[50%]`).
- Dark and light modes (`bg-white dark:bg-black`).
- Responsive styles (`p-4 sm:p-6 md:p-8`).
- CSS interoperability (`h1 { @apply text-2xl; }`).
- Custom themes: Define theme colours, font, sizes, etc., in `tailwind.config.js`.

  [!NOTE] Responsive Design Your web app should render well in different devices (e.g., laptops, tablets, phones). General tip: Avoid absolute lengths! - The most responsive unit is `rem`. - Tailwind units automatically translate into rem (e.g., `pt-4` becomes `padding-top: 1rem;`).

Flex Container:

- Add `class="flex"` (with `flex-col` for vertical) to the parent container.
- To wrap items on overflow, add `flex-wrap`.
- To handle spacing between items, use class names like `justify-center`, `justify-between`, `justify-around`.

Items:

- Control how much space each item takes if there's extra (or too little).
- Use `flex-1` to take whatever space is left!
  - Example use case: A navbar with links on one end, and something on the other.
- Divide items between multiple elements with the same class.

Grid Container:

- `class="grid"` with `grid-cols-3` or `grid-rows-3` for rows/columns.
- Use `gap-4` for gap between items. For responsiveness, we often need to change the number of columns:

```
class="grid sm:grid-cols-2 md:grid-cols-3"
```

Items: To specify space an item needs, use `class="col-span-2"`.

Example!

```
<div className="grid bg-blue-500 grid-cols-4 gap-2 p-2 text-black text-center">
    <div className="bg-blue-300 col-span-4">Header</div>
    <div className="bg-blue-300 row-span-2">Menu</div>
    <div className="bg-blue-300 col-span-2 h[100px]">Main</div>
    <div className="bg-blue-300 col-span-3">Footer</div>
</div>
```

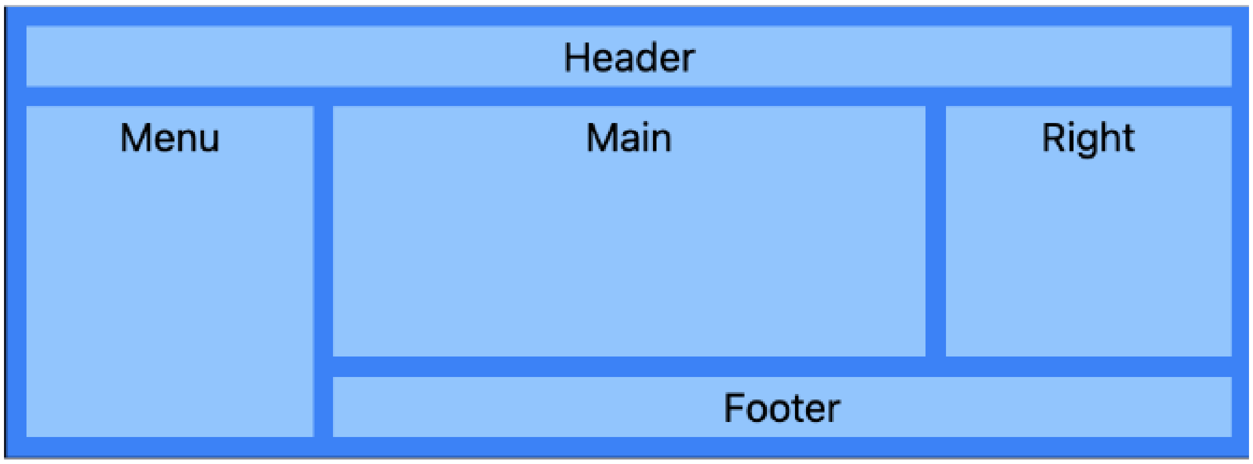


Figure 6: Pasted image 20241217133427.png

Tailwind Notes While a great tool, keep in mind `className` bloat!

- There are often long, repetitive classes. We can either:
  - Create a new component for re-used classes.
  - Use `@apply` to move classes to CSS.

10 - Docker

Isolation

Each machine has:

- A different OS
- Different softwares installed (or different versions of the same software).
- Different softwares running at the same time.
  - Could impact the file system, dependencies, etc.

Traditionally, to fix any issues, we would use:

- Virtual machines - Full isolation, but heavy and slow.
- Sandboxing - Limit a process' access to resources (e.g., RAM, CPU).
- `chroot` jail - Restrict a process to a specific directory. It cannot access outside that directory. There is a trade-off between efficiency and true isolation; managing multiple instances will be difficult. It's also not very portable! We need to redo much of the work on a new machine.

Docker

A platform for developing, shipping, and running applications.

- Allows you to package an application with all its dependencies into a standardized unit called a **container**.
- Makes your app **portable**; can be stopped, restarted, and copied easily!
  - No need to worry about different machines, dependencies, etc.

Benefits:

- Consistency across environments.
- Simplified dependency management (don't need to install manually).
- Containers will run in isolation.
- Easier continuous integration and deployment (CI/CD).

  [!NOTE] Docker History **2008 - Linux Containers (LXC)** - Used Linux kernel features like cgroups and namespaces. - Ran multiple isolated Linux systems on a single host.

  2010 - dotCloud - Founded by Solomon Hykes. - Led the exploration of containerization as a core technology.

  2013 - dotCloud open-sourced their container technology, naming it **Docker**. Now, Docker is the industry standard for deployment.

Containers vs. VMs

Virtual Machine - Runs a full OS with its own kernel and a virtualized set of hardware resources (CPU, memory, storage) on a physical machine.

Docker Container - Uses the host OS's kernel.

- Has process-level isolation.
- Shared kernel space, isolated user space.

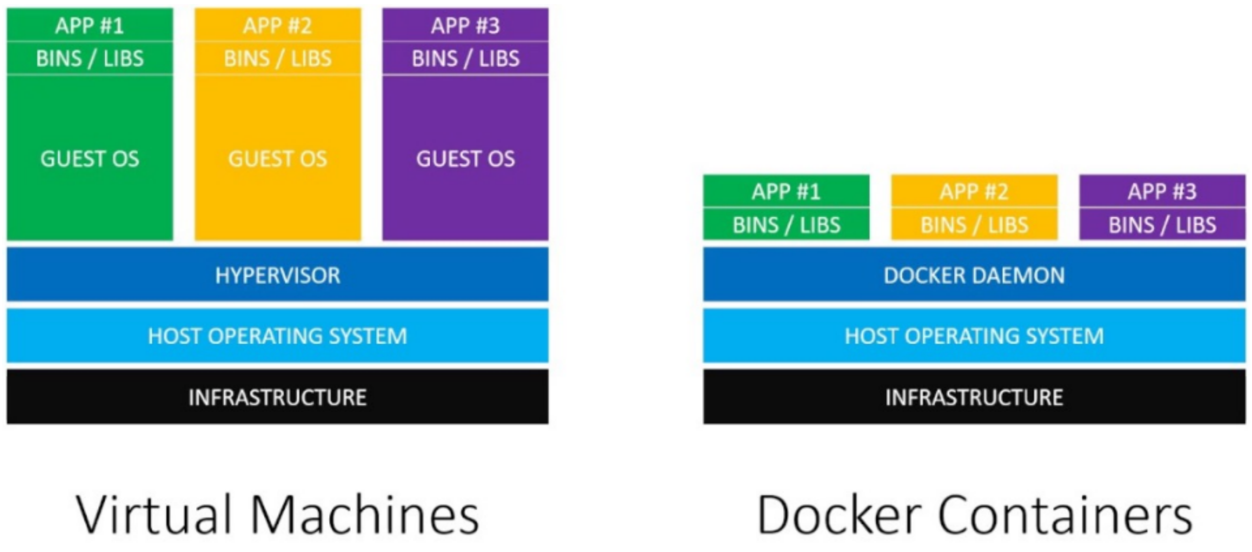


Figure 7: Pasted image 20241217135658.png

Containers vs. VMs:

Containers	Virtual Machines
Consistent across environments with the same OS	Consistent across all environments, regardless of OS
Lightweight	Very heavy
Fast start-up	Slow start-up
Low performance overhead	High performance overhead

Docker Concepts

Dockerfile Contains instructions on how to build a Docker image (e.g., where dependencies are installed).
Instructions:

- `FROM`: Base **image** to start with
- `RUN`: Executes commands (e.g., `apt install`).
- `COPY`: Copies files from the host to the image.
- `CMD`: Command to run when the container starts.
  - The container exists as soon as the CMD finishes (so only run CMD once!).

Example Dockerfile

1. Create a file named `Dockerfile`.
2. Build command: `docker build -t hello`.
3. Run command: `docker run hello`.

```
FROM python:3.11
RUN echo 'print("Hello, World!")' > /app.py
CMD python /app.py
```

Next.js Dockerfiles

- Runs the Next.js application in **development** mode.

- Copies files to the image, installs the dependencies, and exposes the port.
- Run command: `docker run -p 3000:3000 nextjs-app`.

```
FROM node:20-alpine
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD npm run dev
```

Docker Images A lightweight, standalone, and executable package.

- Includes everything needed to run a software (e.g., code, dependencies, `env` variables, and system tools).
- Images are a read-only template used to create Docker containers.
- A build will start from a base image (e.g., `alpine, python:3.12`).
- Built in **layers:** Each layer represents a step in a Dockerfile.
  - Layers are cached for efficiency and reusability.
- Images are immutable and portable.
- Can be versioned using tags; default tag is `latest`.

Docker Containers

- Instantiated from Docker images.
- Run command: `docker run -d -p 8080:80 <image_name>:<image_tag>`.
- List running containers: `docker ps`.
- Stop a container: `Docker stop <container_name>`.
- View logs: `docker logs <container_name>`.

Docker Volumes

- Persistent data storage for docker containers.
- Also allows for sharing data between containers.
- Use cases: database, user uploads, HTTPS certificates. Example: PostgreSQL

```
%% Commands %%
docker volume create pgdata
docker run -d \
    --name my-postgres \
    -e POSTGRES_PASSWORD=password \
    -v pgdata:/var/lib/postgresql/data \
    -p 5432:5432 \
    postgres
```

Docker Hub Visit https://hub.docker.com.

- Global repository of docker images.
- You can search, explore, and use millions of images; Dockerfile's Base images are downloaded from Docker hub.

- You can push/publish your own images:
  - `docker login` then `docker push <username>/<image_name>:<image_tag>`.

Docker Compose

- Applications often have multiple containers (e.g., for backend, web server, database, static files, etc.).
- We can orchestrate all containers into one place, specifying an order of containers to be run on startup. **Steps:**

1. Create a file named `docker-compose.yml`.
2. Run the setup with `docker compose up`.
3. Stop with `docker compose down`.

Deploying Docker Images

Deploying Docker images are easy (see also [[11 - Deployment|Week 12]]).

Cloud Providers

- Every major cloud provider has services to directly deploy a Dockerfile or image (e.g., AWS App Runner, GCP Cloud Run).
- Great way to quickly deploy your image to the internet.
  - The cloud will manage the domain, permissions, load balancing, etc.

Serverless Functions

- Write the functions; the Cloud will containerize and deploy it (e.g., AWS Lambda, Vercel).
- Perfect for deploying a simple service as quickly as possible.

Cloud-Based Applications

- These days, applications are broken into **microservices**.
- Some services are directly provided by the Cloud (e.g., AWS RDS and AWS S3).
- Services are either managed in a Docker Compose (for smaller applications) or in an K8s orchestration.

  [!NOTE] Kubernetes (K8s) - Open source container orchestration platform. - Designed for large-scale setups. - Has scaling, load balancing, and clustering features. - Supports automated deployment and rollbacks.

11 - Deployment

Deploying a Web Application

Server - A computer/virtual machine with a **static public** IP address.

- Can also have a domain bound to the IP.
  - They need to unique, and typically requires to be purchased on a yearly basis. The IP only determines how the application can be accessed.
- You can access your local website within the same network using `npm run dev -H 0.0.0.0`. However, you may need to run your application differently between deployment and production!

Development vs. Production

Development:

- Uses a test database (e.g., SQLite).
- Upon errors, the **stack trace** is shown.
- Run with a deployment server.

Production:

- Uses a real database with user data (e.g., Postgres, MySQL).
- Shows generic errors (e.g., 500, 404).
- Run with a real **webserver**.

Deploying Next.js Projects

General Notes:

1. Make sure to use the **latest version** of node/Docker to avoid any code being out-of-date.
2. Clone your code onto the server (e.g., Replit, VM), install any dependencies and deploy migrations. (e.g., `npm install; npx prisma generate; npx prisma migrate deploy`).
   - Note: There is a difference between `npx prisma migrate dev` and `npx prisma migrate deploy`!
     - We don't want to generate new migrations when deploying the code, since we are not developing it anymore.
     - A new migration is a new file; you **do not want to generate files when deploying**, i.e., touch the code base once it's deployed.
     - Ensures that the migration files will be in sync and avoids multiple migrations in deployment.
3. The app is started using `npm run dev` and accessed using the server's IP address, but this is still the **development mode**!

These steps aren't technically a true deployment. It is missing something to go from development to deployment!

- Performance is very slow!
  - **Has hot-reload:** Does not restart the server when a file changes!
    - Consequence: Compiles every page and needs to reload every time something changes.
  - TypeScript and JSX need to get compiled to JS, since they are only there for development.
- Environment variables should differ from development to production.
  - **Production env:** They need to be overridden to keep any secret keys/tokens!

Production Settings and Build Next.js has built-in support for different environment variables: Create a base `.env` file, then use `.env.production` and `.env.development` for any overrides.

- Frontend `env` variables should start with `NEXT_PUBLIC_` since they will be shared with the browser. Remember to not push the `.env` files or the list of keys (e.g., `env.sample`) to git!

1. Build your app for production using `npm run build`.
   - Builds, transforms, and compiles all files and dependencies into production, builds them, and bundles them into HTML/CSS/JS files.
   - Static pages don't need any code to be run on the server.
2. The `.next` file will be generated, which self-contains all of the files required for the build.
   - Contains `server/`, `static/`, `cache/`, etc.
   - This will minify all JS files, and performs several optimizations!
     - Pre-generates and caches everything it can.
3. Run the **production** application using `npm start`.

To dockerize your application:

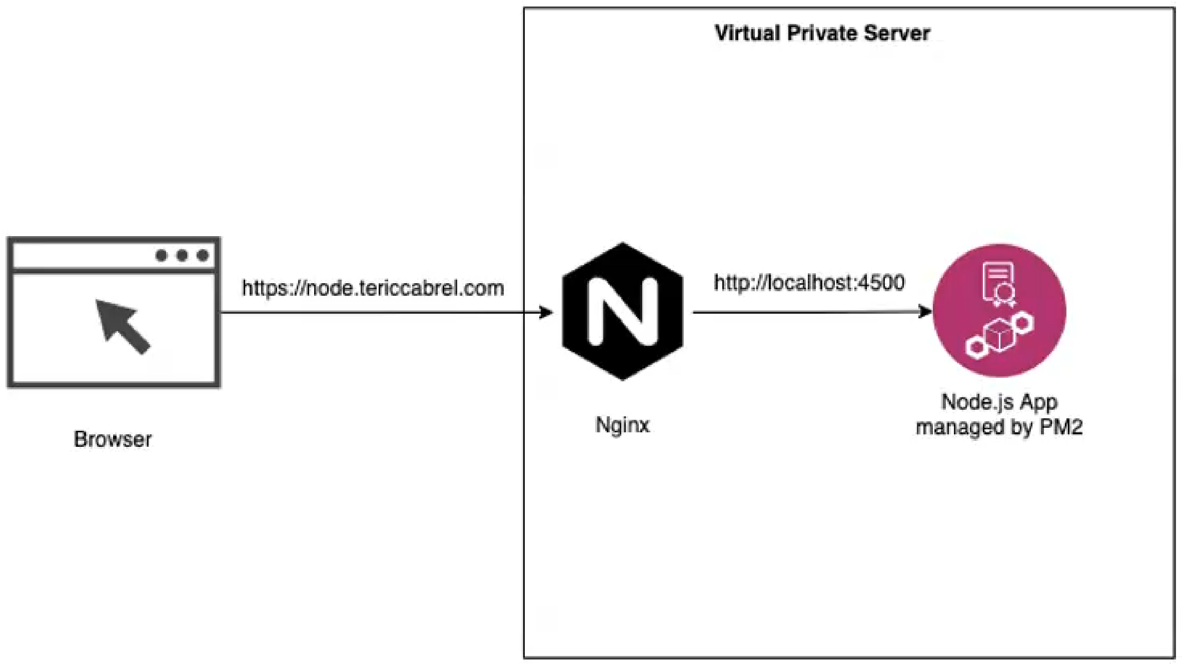
```
FROM node:20-alpine
WORKDIR /app
COPY . .
RUN npm install
RUN npx prisma generate
RUN npm run build
EXPOSE 3000
ENV NODE_ENV=production
CMD npm start
```

Be sure to use `.dockerignore` for any unnecessary files (e.g., node_modules). Notice that `npx prisma migrate` isn't included in the docker file:

- The dockerfile is run during the build time of the image containing the code (we do not have access to the database at this point).
  - This is only to ensure that the commands work without any environment variables. We do not want to contact the database yet. just things that are run in the container!
  - There is a separation with the build, run, and container phases of a web application. We shouldn't migrate or deal with the database here! We do need `npx prisma generate` though, since we need the schema itself (Prisma generates JS code). to understand any fields of a model/class in the database.
- It is not a part of the project's source code!

Tools Used in Deployment

Process Manager 2 Code is not directly run through the Node server. Instead, it runs via Process Manager 2 (PM2).



Advantages:

- Managing lifecycle:

    - App will run continuously.
    - Restarts the application after crash.
    - Can start the app on reboot.
- Scaling the application:
    - Can configure the number of workers and cores.
- Monitoring:
    - Access to logs and metrics.

Installation:

1. Install: `sudo npm install -g pm2`
2. Start the app: `pm2 start "npm start" --name nextapp`
3. **Daemonize**: `pm2 startup`, then `pm2 save`

Commands:

- Status of apps: `pm2 list`
- View logs: `pm2 logs nextjs-app`
- Monitor: `pm2 monit`
- Restart: `pm2 restart nextjs-app`

PM2 and Docker's usage is interchangeable, and depending on what you need, you can use one or both for deploying your web application.

New Dockerfile (Using PM2):

```
## Build
FROM node:20-alpine AS builder
WORKDIR /app
RUN npm install -g pm2
COPY package*.json ./
RUN npm install
COPY . .
RUN npx prisma generate
RUN npm run build


## Run

## Only copy the necessary files to the production image
FROM node:20-alpine AS runner
WORKDIR /app


## Copy the built Next.js application from the builder stage
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/public ./public
COPY --from=builder /app/package.json ./


## Do not install devDependencies
RUN npm install --only=production
EXPOSE 3000
```

```
ENV NODE_ENV=production
CMD pm2-runtime start npm -- start
```

Here, the first image is temporary (a provisional image). We will take only some of these files out, then copy them into the final production image.

- Note we can only have one CMD.

This two-step process optimizes our deployment a lot!

- We no longer need our code to run (since we already built it). This will decrease the size of our image.
- You don't need dependencies not used in production (e.g., TypeScript).

Also note that we use PM2 for runtime!

- Docker does not have a background process; if the container stops, then all of the space is released.
- However, with PM2, it will run in the background!

Webserver

Webserver - A process that listens in on specific ports (e.g., 80, 443).

- Will serve multiple web applications on the same system.
- Examples: Nginx, Apache

  [!NOTE] Deploying with Nginx Nginx has many features, especially for security (e.g., DDOS prevention) or configuration with rate limiting or hosts/domains. These are not provided by PM2 or NPM. 1. Create a config file at `/etc/nginx/sites-available/<project_name>`. 2. Make a **symbolic link** to that file at `/etc/nginx/sites-enabled/`. 3. Restart Nginx. "' server { listen 80; server_name csc309.xyz www.csc309.xyz;

  ```
  location /_next/static/ {
      alias /home/ubuntu/csc309_proj/./next/static/;
  }
  location /public/ {
      alias /home/ubuntu/csc309_proj/public/;
  }
  location / {
      include proxy_params;
      proxy_pass http://localhost:3000/;
  }
  ```

  } "'

Static Files Webservers are also important for **serving static files** (e.g., JS/CSS/HTML, images, fonts).

- Webservers will increase the performance and security of these files (since it doesn't have to run through the node server). Static files can also be uploaded to CDNs for even faster retrieval.

Be sure to check for permissions of your static files:

- The webserver needs to read static files and iterate through directories.

The Final Next.js Docker Compose

This is the culmination of all our work to create a portable and self-contained web application!

Volumes

- Database

- Static files storage **Services (Containers)**
- Database
- Migration runner
- Backend code (build and run via PM2)
- Webserver

The above procedure takes a long time, so Vercel is often used to deploy web applications instead.

- Developed by the creator of Next.js
- Very straightforward deployment system
- Will build the application and host it on its own servers.
  - Has automatic deployment per push to the main branch.
  - Provides HTTPS, scaling, and various optimizations. However, Vercel is a **serverless** platform, and only works for Next.js projects (e.g., cannot host databases and persistent storage with the application).

DevOps

In a company, code will change constantly. Whenever changes need to be made, it will go through a delivery pipeline/process so that it can be automatically deployed on its own!

DevOps - The set of principles and practices that automate build and deployment of software (for continuous delivery).

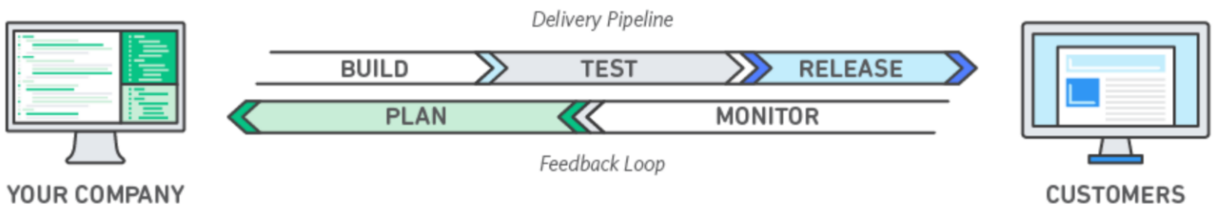


Figure 8: Pasted image 20241215200653.png

Auto Dev-Ops: Some repositories (e.g., Github/Gitlab) support auto DevOps! Once a push is made, a pipeline starts:

1. Test
2. Build
3. Deploy This is also called CI/CD (continuous integration and continuous delivery).

Github can run all CI/CD processes through a **docker container**. If everything is fine, the docker images will be your application! This can be pushed to a registry and pulled in the server.

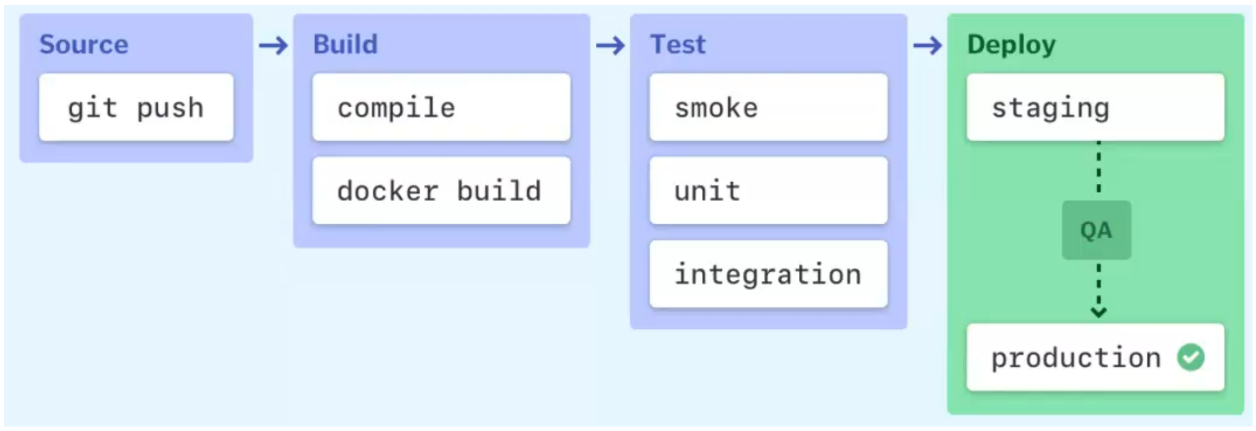


Figure 9: Pasted image 20241215201210.png