

Week 1

How to setup the assignments repo locally and run the test cases?

Steps

1.Check node is available in your machine.using terminal!

node -v -->to check the node version

```
praveenkumars@PRAVEENKUMARs-MacBook-Pro ~ % node -v  
v20.9.0  
praveenkumars@PRAVEENKUMARs-MacBook-Pro ~ % npm -v  
10.2.3
```

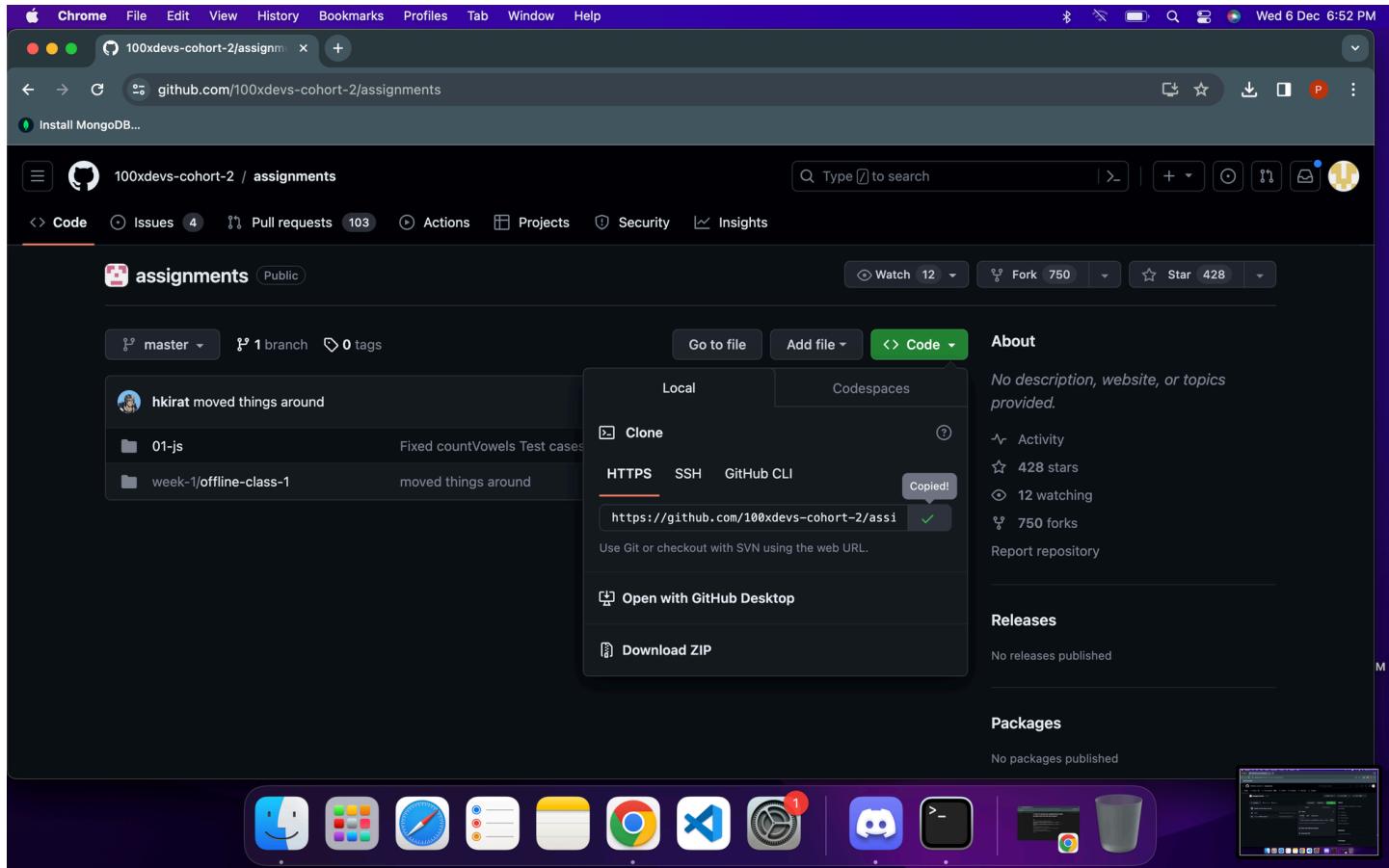
2.If it is already installed go to step 4.if not.step 3

3.install nodejs from [download nodejs](#) and setup it.refer youtube video if required.

4.install the vs code and complete the setup .

5.install and setup the git locally.refer youtube video.

6.Go to [assignments repo](#) and copy the link.or download the .zip file and open it.



1. open the terminal and use the command

```
git clone githuburl
```

```
[praveenkumars@PRAVEENKUMARs-MacBook-Pro assignments % git clone https://github.com/100xdevs-cohort-2/assignments ]
```

8 . do the following commands the file will open in the vs code or manually open it.

```
[praveenkumars@PRAVEENKUMARs-MacBook-Pro assignments % ls
assignments
[praveenkumars@PRAVEENKUMARs-MacBook-Pro assignments % cd assignments
[praveenkumars@PRAVEENKUMARs-MacBook-Pro assignments % ls
01-js    week-1
[praveenkumars@PRAVEENKUMARs-MacBook-Pro assignments % cd 01-js
praveenkumars@PRAVEENKUMARs-MacBook-Pro 01-js % code .]
```

9 .Now vs code will be open 01-js file.Go to easy folder choose any problem .I choose anagram.js and solve it.

```

1 /*
2  Write a function 'isAnagram' which takes 2 parameters and returns true/false if those are anagrams
3  What's Anagram?
4  - A word, phrase, or name formed by rearranging the letters of another, such as spar, formed from
5 */
6
7 function isAnagram(str1, str2) {
8     const cleanStr = (str) => str.replace(/\s/g, '').toLowerCase();
9     const sortedStr = (str) => cleanStr(str).split('').sort().join('');
10
11     return sortedStr(str1) === sortedStr(str2);
12 }
13
14
15
16
17 module.exports = isAnagram;
18

```

1. For running the test for particular problem. Open the terminal use the following command with your file name.

```
npx jest ./tests/anagram.test.js
```

```

1 praveenkumars@PRAVEENKUMARS-MacBook-Pro 01-js % npx jest ./tests/anagram.test.js
2 PASS tests/anagram.test.js
3   isAnagram
4     ✓ returns true for anagrams (3 ms)
5     ✓ returns false for non-anagrams (1 ms)
6     ✓ returns true for anagrams with different casing (1 ms)
7     ✓ returns true for anagrams with special characters (1 ms)
8     ✓ returns false for non-anagrams with special characters (1 ms)
9
10 Test Suites: 1 passed, 1 total
11 Tests:       5 passed, 5 total
12 Snapshots:   0 total
13 Time:        0.257 s, estimated 1 s
14 Ran all test suites matching ./tests\anagram.test.js/i.
15
16 praveenkumars@PRAVEENKUMARS-MacBook-Pro 01-js %

```

1. For running all the test case use the following command

```
npx jest ./tests/
```

SUCCESSFULLY COMPLETED

***Note**

To run the test case should be in correct directory for this week it is 01-js

IF ERROR OCCURS

RUN THE FOLLOWING COMMANDS

```
npm i -g  
npm i npm -g
```

CHECK THE CURRENT DIRECTORY IT SHOULD BE 01-js



Week 3.1

Middlewares, Global Catches & Zod

In this lecture, Harkirat dives deep into **Middlewares**: behind-the-scenes helpers that tidy up things before your main code does its thing. **Global catches**: safety nets for your code, they catch unexpected issues before they cause chaos. And finally, **Zod**: a library that ensures efficient input validation on your behalf.

Middlewares, Global Catches & Zod

Understanding Middlewares:

Middlewares in JS Context & Problem Statement:

Solution: Middlewares

Some Associated Concepts:

1. next() Keyword:
2. Difference between res.send and res.json:
3. Importance of app.use(express.json()):
4. Middleware and req.body:

3 Ways of Sending Inputs to a Response:

1. Query Parameter:
2. Body:
3. Headers:

Bottom Line:

Global Catches:

Importance of Global Error Handling:

Input Validation:

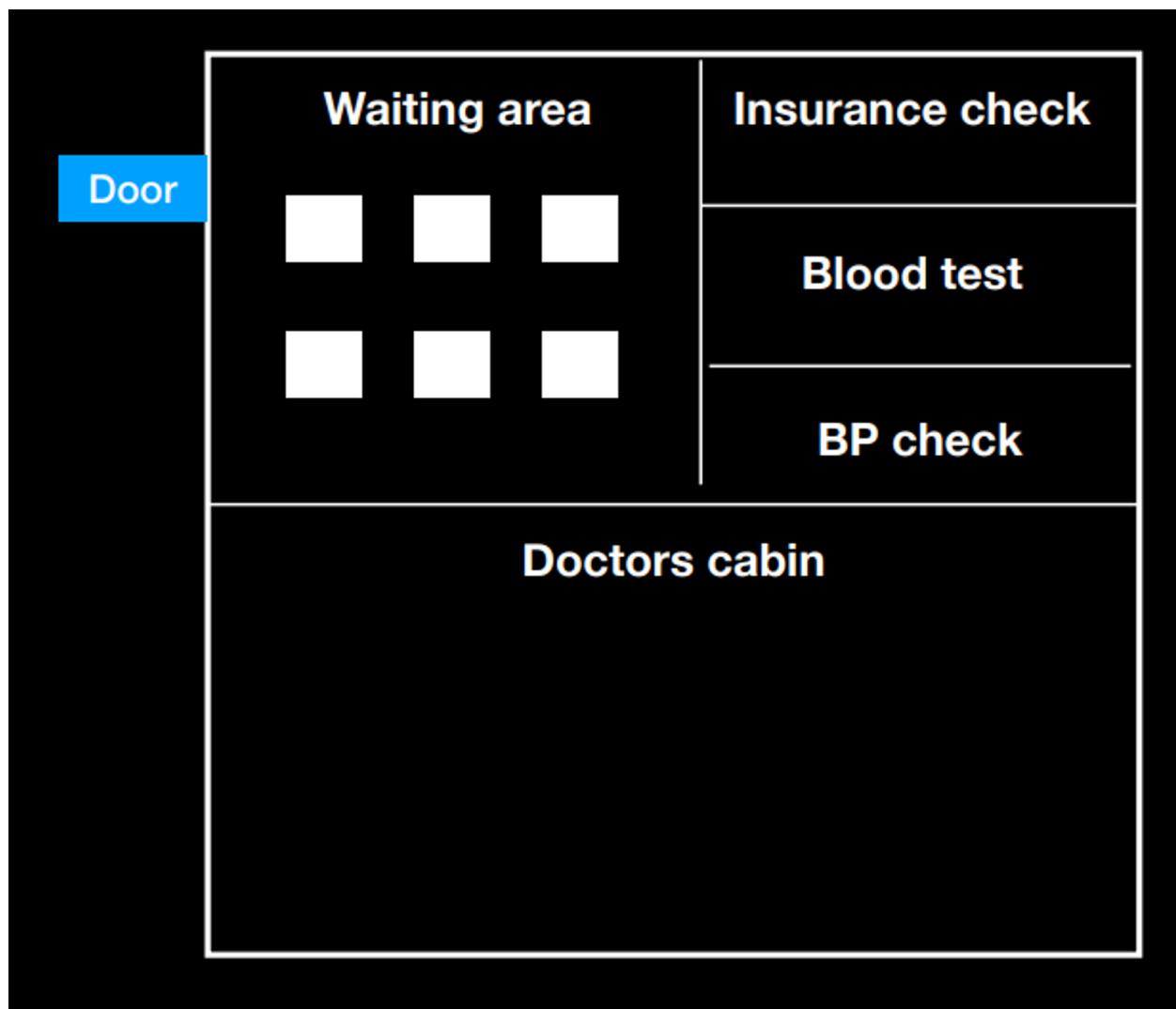
1. Naive Way - Multiple If-Else Statements:

2. Using zod Library for Schema Validation:

Zod:

Zod Syntax Overview:

Why Zod:



Understanding Middlewares:

Imagine a Busy Hospital:

Think of a hospital where there's a doctor, patients waiting in line, and a few helpful assistants making sure everything runs smoothly.

1. Doctor's Cabin (**Application Logic**) :

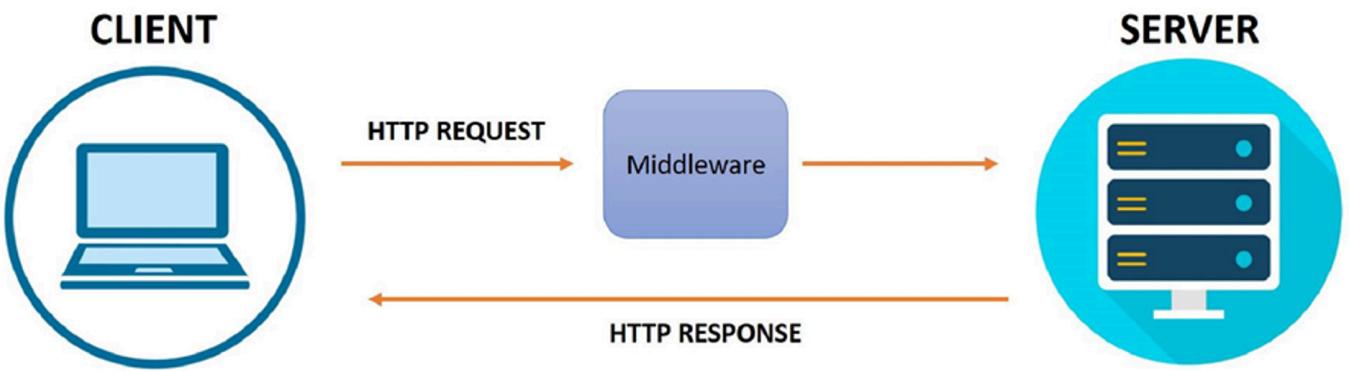
- The doctor is like the main brain of our hospital – ready to help patients with their problems.

2. Waiting Room (**Callback Queue**) :

- The waiting room is where patients hang out before seeing the doctor. Each patient has a unique situation.

3. Intermediates (**Middlewares**) :

- Before a patient sees the doctor, there are some helpers doing important tasks.
- One helper checks if patients have the right paperwork . This is like ensuring everyone is who they say they are (**Authentication**)
- Another helper does quick health checks – like making sure patients' blood pressure is okay. This is similar to checking if the information coming to the doctor is healthy and makes sense (**Input Validation**)



Middlewares in JS Context & Problem Statement:

Earlier we used to organize all our prechecks followed by the application logic all in one route.

Middlewares emerged as a solution to enhance code organization by extracting prechecks from the core application logic. The motivation behind their introduction lies in our commitment to the "**"Don't Repeat Yourself" (DRY) principle**".

By isolating these preliminary checks into distinct functions or code blocks known as middlewares, we achieve a more modular and maintainable codebase. This separation not only streamlines the primary application logic but also promotes code reuse, making it easier to manage, understand, and scale our software architecture.

```

6 v app.get("/health-checkup", function (req, res) {
7   // do health checks here
8   const kidneyId = req.query.kidneyId;
9   const username = req.headers.username;
10  const password = req.headers.password;
11
12  if (username != "harkirat" && password != "pass") {
13    res.status(403).json({
14      msg: "User doesn't exist",
15    });
16    return;
17  }
18
19  if (kidneyId != 1 && kidneyId != 2) {
20    res.status(411).json({
21      msg: "wrong inputs",
22    });
23    return;
24  }
25  // do something with kidney here
26
27  res.send("Your heart is healthy");
28 });
29

```

Username checks → [lines 12-17]

Input validation → [lines 19-24]

Solution: Middlewares

Middlewares

Defining middleware (just another fn)

```

index.js > f app.get("/heart-check") callback
4  const app = express();
5
6  function userMiddleware(req, res, next) {
7    if (username != "harkirat" && password != "pass") {
8      res.status(403).json({
9        msg: "Incorrect inputs",
10       });
11    } else {
12      next();
13    }
14  };
15
16  function kidneyMiddleware(req, res, next) {
17    if (kidneyId != 1 && kidneyId != 2) {
18      res.status(403).json({
19        msg: "Incorrect inputs",
20       });
21    } else {
22      next();
23    }
24  };
25
26  app.get("/health-checkup", userMiddleware, kidneyMiddleware, function (req, res) {
27    // do something with kidney here
28
29    res.send("Your heart is healthy");
30  });
31
32

```

Using the middleware

Furthermore, with middleware, we can easily include as many precheck functions as needed. This means we have the freedom to add various checks or operations to our application without making the main code complex. It's like having building blocks that we can mix and match to create a customized

process for our application, making it more adaptable and easier to manage.

Here `userMiddleware` and `kidneyMiddleware`

Some Associated Concepts:

1. `next()` Keyword:

In middleware functions in Express, `next` is a callback function that is used to pass control to the next middleware function in the stack. When you call `next()`, it tells Express to move to the next middleware in line. If `next()` is not called within a middleware function, the request-response cycle stops, and the client receives no response.

Example:

```
app.use((req, res, next) => {
  console.log('This middleware runs first.');
  next(); // Move to the next middleware
});

app.use((req, res) => {
  console.log('This middleware runs second.');
  res.send('Response sent from the second middleware.');
});
```

2. Difference between `res.send` and `res.json`:

- `res.send` : Sends a response of various types (string, Buffer, object, etc.). Express tries to guess the content type based on the data provided.

```
res.send('Hello, World!'); // Sends a plain text response
```

- `res.json` : Sends a JSON response. It automatically sets the `Content-Type` header to `application/json`.

```
res.json({ message: 'Hello, JSON!' }); // Sends a JSON response
```

3. Importance of `app.use(express.json())`:

`app.use(express.json())` is middleware that parses incoming JSON payloads in the request body. It is crucial when dealing with JSON data sent in the request body, typically in POST or PUT requests. Without this middleware, you might receive the JSON data as a raw string, and you'd need to manually parse it.

Example:

```
const express = require('express');
const app = express();

app.use(express.json()); // Middleware to parse JSON in the request body

app.post('/api/data', (req, res) => {
  const jsonData = req.body; // Now req.body contains the parsed JSON data
  // Process the data...
  res.json({ success: true });
});
```

4. Middleware and `req.body` :

- `req.query` and `req.headers` don't require middleware because they represent the query parameters and headers of the incoming request, respectively. Express automatically parses them.
- `req.body` requires middleware like `express.json()` to parse the request body, especially when the body contains JSON data. Other middleware, like `express.urlencoded()`, is used for parsing form data in the request body.

Middleware helps in processing the request at different stages and is essential for tasks like parsing, logging, authentication, and more in a modular and organized way.

3 Ways of Sending Inputs to a Response:

1. Query Parameter:

- **What it is:** Like giving specific instructions in the web address.
- **Example:** In `www.example.com/search?topic=animals`, the query parameter is `topic` with the value `animals`.
- **Use Case:** Good for simple stuff you want everyone to see, like search terms in a URL.

2. Body:

- **What it is:** Imagine it as the hidden part of a request, carrying more detailed information.
- **Example:** When you fill out a form on a website, the details you enter (name, email) go in the body of the request.
- **Use Case:** Great for sending lots of information, especially when you're submitting something like a form.

3. Headers:

- **What it is:** Extra information attached to the request, kind of like details about a letter.
- **Example:** Headers could include things like your identity or the type of data you're sending.
- **Use Case:** Perfect for passing along special information that doesn't fit neatly in the URL or body, like who you are or how to handle the data.

Bottom Line:

- **Query Parameters:** Simple instructions visible in the web address.
- **Body:** Hidden part of the request for more detailed info, great for forms.
- **Headers:** Extra details about the request, useful for special information.

Global Catches:

It essentially help us the developers give a better error message to the user.

Global Catch or Error-Handling Middleware is a special type of middleware function in Express that has four arguments instead of three (`(err, req, res, next)`). Express recognizes it as an error-handling middleware because of these four arguments.

```
// Error Handling Middleware
const errorHandler = (err, req, res, next) => {
  console.error('Error:', err);

  // Customize the error response based on your requirements
  res.status(500).json({ error: 'Something went wrong!' });
};

};
```

Importance of Global Error Handling:

1. Centralized Handling:

- Global catch blocks allow you to centrally manage and handle errors that occur anywhere in your application. Instead of handling errors at each specific location, you can capture and process them in a centralized location.

2. Consistent Error Handling:

- Using a global catch mechanism ensures a consistent approach to error handling throughout the application. You can define how errors are logged, reported, or displayed in one place, making it easier to maintain a uniform user experience.

3. Fallback Mechanism:

- Global catches often serve as a fallback mechanism. If an unexpected error occurs and is not handled locally, the global catch can capture it, preventing the application from crashing and providing an opportunity to log the error for further analysis.

Input Validation:

Input validation is a crucial aspect of securing your application. It helps ensure that the data received by your server is in the expected format and meets certain criteria.

Take for instance a login schema, now instead of passing a username and password in the body, the user can pass in any gibberish and may try to crash the server. Thus, it is our responsibility to ensure that our application logic handles all these input vulnerabilities.

Let's explore two approaches to input validation: the naive way with multiple `if-else` statements and using the `zod` library for schema validation.

1. Naive Way - Multiple If-Else Statements:

In the naive approach, you manually check each input parameter to ensure it meets your criteria. Here's an example using Express.js:

```
const express = require('express');
const app = express();

app.use(express.json());

app.post('/login', (req, res) => {
  const { username, password } = req.body;

  if (!username || typeof username !== 'string' || username.length < 3 ||
    !password || typeof password !== 'string' || password.length < 6) {
    return res.status(400).json({ error: 'Invalid input.' });
  }

  // Proceed with authentication logic
  // ...
})
```

```

res.json({ success: true });

});

const PORT = 3000;
app.listen(PORT, () => console.log(`Server is running on http://localhost:${PORT}`));

```

In this example, we manually check the `username` and `password` fields for their existence, data type, and minimum length. This approach can become cumbersome as the number of input parameters increases, and it may lead to code duplication.

2. Using `zod` Library for Schema Validation:

`zod` is a TypeScript-first schema declaration and validation library. It provides a concise way to define schemas and validate input data. Here's an example using `zod` for the same login scenario:

```

const express = require('express');
const { z } = require('zod');
const app = express();

app.use(express.json());

const loginSchema = z.object({
  username: z.string().min(3),
  password: z.string().min(6),
});

app.post('/login', (req, res) => {
  const { username, password } = req.body;

  try {
    loginSchema.parse({ username, password });
    // Proceed with authentication logic
    // ...
    res.json({ success: true });
  } catch (error) {
    res.status(400).json({ error: 'Invalid input.', details: error.errors });
  }
});

const PORT = 3000;
app.listen(PORT, () => console.log(`Server is running on http://localhost:${PORT}`));

```

In this example, we define a `loginSchema` using `zod` that specifies the expected structure and constraints for the input data. The `parse` method is then used to validate the input against the

schema. If the input is invalid, `zod` throws an error, and we can handle it appropriately. This approach is more concise and less error-prone compared to the manual if-else checks.

Zod:

Zod is a TypeScript-first schema declaration and validation library. It provides a simple and expressive way to define the structure and constraints of your data, allowing you to easily validate and parse input against those specifications. Here's a brief explanation of Zod and its syntax:

Zod Syntax Overview:

1. Basic Types:

- Zod provides basic types such as `string`, `number`, `boolean`, `null`, `undefined`, etc.

```
const schema = z.string();
```

2. Object Schema:

- You can define the structure of an object using the `object` method and specify the shape of its properties.

```
const userSchema = z.object({
  username: z.string(),
  age: z.number(),
});
```

3. Nested Schemas:

- You can nest schemas within each other to create more complex structures.

```
const addressSchema = z.object({
  street: z.string(),
  city: z.string(),
});
```

```
const userSchema = z.object({
  username: z.string(),
  address: addressSchema,
});
```

4. Array Schema:

- You can define the schema for arrays using the `array` method.

```
const numbersSchema = z.array(z.number());
```

5. Union and Intersection Types:

- Zod supports union and intersection types for more flexibility.

```
const numberOrStringSchema = z.union([z.number(), z.string()]);
const combinedSchema = z.intersection([userSchema, addressSchema]);
```

6. Optional and Nullable:

- You can make properties optional or nullable using `optional` and `nullable` methods.

```
const userSchema = z.object({
  username: z.string(),
  age: z.optional(z.number()),
  email: z.nullable(z.string()),
});
```

7. Custom Validators:

- Zod allows you to define custom validation logic using the `refine` method.

```
const positiveNumberSchema = z.number().refine((num) => num > 0, {
  message: 'Number must be positive',
});
```

8. Parsing and Validation:

- To validate and parse data, use the `parse` method. If the data is invalid, it throws an error with details about the validation failure.

```
try {
  const userData = userSchema.parse({
    username: 'john_doe',
    age: 25,
    address: {
      street: '123 Main St',
      city: 'Exampleville',
    },
  });
  console.log('Parsed data:', userData);
} catch (error) {
  console.error('Validation error:', error.errors);
}
```

Why Zod:

- **TypeScript-First Approach:** Zod is designed with TypeScript in mind, providing strong type-checking and autocompletion for your schemas.
- **Concise and Expressive Syntax:** Zod's syntax is concise and expressive, making it easy to define complex data structures with minimal code.
- **Validation and Parsing:** Zod not only validates data but also automatically parses it into the expected TypeScript types.
- **Rich Set of Features:** Zod includes a variety of features, such as custom validation, optional and nullable types, union and intersection types, making it a powerful tool for data validation in your applications.

Overall, Zod simplifies the process of declaring and validating data structures, reducing the likelihood of runtime errors and improving the overall robustness of your code.



Week 3.5

Document Object Model (Laisha)

In this session, Laisha delves into the [Document Object Model \(DOM\)](#), demystifying the core structure that transforms plain HTML pages into dynamic, interactive web encounters through JavaScript. She also sheds light on NodeList and HTMLCollection, the essential tools for effectively handling clusters of elements.

[Document Object Model \(Laisha\)](#)

[DOM](#)

[What is DOM?](#)

[Communication with the Browser](#)

[Accessing the DOM](#)

[Possibilities of DOM](#)

[Independence and Consistency](#)

[DOM Tree](#)

[Implementing innerHTML](#)

[Difference Between HTMLCollection and NodeList:](#)

[HTMLCollection:](#)

[NodeList:](#)

[Practical Considerations:](#)

[HTMLCollection](#)

Finding HTML Elements

By ID:

By Tag Name:

By Class Name:

By CSS Selector:

By HTML Object Collections:

Changing HTML Elements

Example - using setAttribute to change an input field to a button:

Adding HTML Elements:

Deleting HTML Elements:

Query Selectors

DOM Node & Methods

Key Points:

Types Of Nodes

DOM Events

Key Concepts:

Reacting to Events:

Common Events:

Example:

The onload and onunload functions:

onload Event:

onunload Event:

DOM Event Listeners

Using addEventListener:

Syntax:

Example of Multiple Event Listeners:

Event Bubbling & Event Capturing

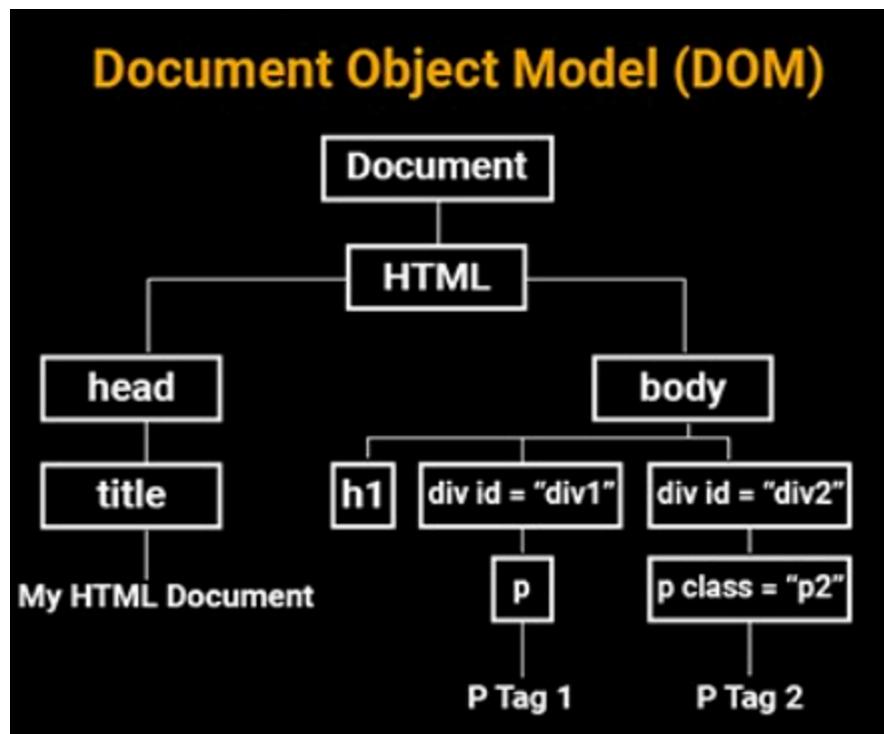
Example of Event Capturing:

Example of Event Bubbling:

DOM

You know how a remote control brings a television to life by letting you change channels and do cool things? Well, in the web world, JavaScript is like that remote control for your HTML page, making it

active and dynamic. And the secret sauce behind this magic is the DOM – the Document Object Model.



What is DOM?

So, what's this DOM thing? DOM stands for Document Object Model. It's like the behind-the-scenes framework that JavaScript uses to talk to your browser. Imagine it as the language that JavaScript speaks with your web browser to make things happen on your HTML page.

Communication with the Browser

JavaScript and the browser communicate through a set of tools in this magical interface known as the DOM. These tools include properties, methods, and events. It's like having a language to tell your browser what to do and when to do it.

Accessing the DOM

Okay, how do we get our hands on this DOM magic? Well, accessing the DOM is like reaching for that remote control. In JavaScript, you use commands to grab elements from your HTML page, change their content, or even create new elements. It's like giving instructions to your browser using JavaScript.

Possibilities of DOM

Now, let's talk about the possibilities the DOM opens up. With JavaScript and the DOM, you can:

- Change the content of your webpage dynamically.
- Update styles and layout on the fly.
- Respond to user interactions, like clicks or keyboard inputs.
- Add or remove elements, making your page super interactive.

Independence and Consistency

DOM doesn't pick sides. It's independent of any particular programming language. This means whether you're using JavaScript, Python, or any other language, the DOM provides a consistent way to interact with your document. It's like a universal remote that works with any TV.

In a Nutshell:

So, the DOM is your backstage pass to making HTML pages come alive with JavaScript. It's a set of rules and tools that allow you to control, change, and interact with your webpage dynamically. It's like giving your webpage a personality and making it respond to your JavaScript commands.

DOM Tree

The DOM tree, or The Document Object Model tree, is a hierarchical representation of the structure of a web document in the context of web development. It's essentially a way to organize and navigate the elements of an HTML or XML document. Here's a breakdown:

- **Document Object:** At the top of the tree is the Document Object, representing the entire web document.
- **HTML Element:** The HTML element comes next, serving as the container for the entire document.
- **Head and Body Elements:** Within the HTML element, there are two main sections: the Head and the Body. The Head typically contains meta-information, styles, and links to external resources, while the Body holds the primary content visible on the webpage.
- **Further Nesting:** Each of these main sections may contain further nested elements. For instance, the Head could include elements like title, meta, or link, while the Body could include paragraphs, images, buttons, and other content-related elements.

The DOM tree essentially forms a family tree-like structure, where elements are organized in a hierarchy based on their relationships with each other. Understanding the DOM tree is crucial for

web developers because it provides a structured way to interact with and manipulate the content of a webpage using programming languages like JavaScript.

Implementing innerHTML

The below implementation consists of an input field that allows users to enter their name. Accompanying this, a button is present, equipped with an `onclick` attribute that invokes the `displayGreeting` function. Within this function, the entered name is acquired by accessing the value of the input field through `document.getElementById("nameInput").value`. Subsequently, the function utilizes `innerHTML` to dynamically alter the content of the `<p>` element identified by the id "greetingMessage," facilitating the display of a personalized greeting message.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Simple Name Greeting</title>
</head>
<body>

    <label for="nameInput">Enter Your Name:</label>
    <input type="text" id="nameInput">
    <button onclick="displayGreeting()">Submit</button>

    <p id="greetingMessage"></p>

    <script>
        function displayGreeting() {
            // Get the entered name from the input field
            var enteredName = document.getElementById("nameInput").value;

            // Render the greeting message on the DOM using innerHTML
            document.getElementById("greetingMessage").innerHTML = "Hello, " + enteredName +
        }
    </script>

</body>
</html>
```

Difference Between HTMLCollection and NodeList:

Both HTMLCollections and NodeLists are collections of nodes in the Document Object Model (DOM) provided by JavaScript, but they have some key differences:

HTMLCollection:

1. Live Collection:

- **Live:** An HTMLCollection is live, meaning it is automatically updated when the underlying document changes. If elements are added or removed, the HTMLCollection is automatically updated to reflect these changes.

2. Accessing Elements:

- **By Index:** Elements in an HTMLCollection can be accessed using numerical indices, similar to an array.

3. Methods:

- **Limited Methods:** HTMLCollections have a more limited set of methods compared to NodeLists.

4. Specific to Elements:

- **Element-Specific:** HTMLCollections are typically used for collections of HTML elements, such as those returned by `getElementsByName` or `getElementsByClassName`.

NodeList:

1. Live or Static:

- **Live or Static:** A NodeList can be live or static. If it's obtained using `querySelectorAll`, it's static and won't automatically update. If it's obtained by other means, like `childNodes`, it might be live.

2. Accessing Elements:

- **By Index or forEach:** Like HTMLCollection, you can access elements by index. Additionally, NodeList supports the `forEach` method for iteration.

3. Methods:

- **Richer Set of Methods:** NodeLists typically have a broader set of methods compared to HTMLCollections.

4. Not Limited to Elements:

- **Node-Oriented:** NodeLists can include various types of nodes, not just HTML elements. They might include text nodes, comment nodes, etc.

Practical Considerations:

- **Common Methods:**
 - For general purpose, when using methods like `querySelectorAll`, you will get a NodeList.
- **Live vs. Static:**
 - If you need a live collection that automatically updates, an HTMLCollection might be suitable.
 - If you want a static collection that won't change, or if you need a broader range of methods, a NodeList might be preferable.
- **Usage:**
 - HTMLCollections are often associated with specific methods like `getElementsByClassName` or `getElementsByTagName`.
 - NodeLists are often the result of more generic methods like `querySelectorAll` or properties like `childNodes`.

In summary, the choice between HTMLCollection and NodeList depends on your specific needs, especially regarding the liveliness of the collection and the methods you require for manipulation.

HTMLCollection

Finding HTML Elements

By ID:

To find an HTML element by its ID, you can use the `getElementById` method.

```
var elementById = document.getElementById("yourElementId");
```

By Tag Name:

To find HTML elements by their tag name, you can use the `getElementsByName` method.

```
var elementsByName = document.getElementsByName("yourTagName");
```

By Class Name:

To find HTML elements by their class name, you can use the `getElementsByClassName` method.

```
var elementsByClassName = document.getElementsByClassName("yourClassName");
```

By CSS Selector:

To find HTML elements using CSS selectors, you can use the `querySelector` or `querySelectorAll` methods.

```
var elementBySelector = document.querySelector("yourCSSSelector");
var elementsBySelectorAll = document.querySelectorAll("yourCSSSelector");
```

By HTML Object Collections:

To find HTML elements using HTML collections, you can use methods like `getElementsByName` or `getElementsByName` in specific cases.

```
var elementsByName = document.getElementsByName("yourElementName");
```

These methods provide different ways to locate and interact with HTML elements in a document using JavaScript. Choose the appropriate method based on your specific needs and the structure of your HTML document.

Changing HTML Elements

Changing HTML elements dynamically is a fundamental aspect of web development, and JavaScript provides several methods to achieve this. Here are some commonly used methods for changing HTML elements:

1. `innerHTML` :

- **Purpose:** Changes the HTML content (including tags) of an element.
- **Example:**

```
document.getElementById("myElement").innerHTML = "New content";
```

2. `textContent` :

- **Purpose:** Changes the text content of an element, excluding HTML tags.

- Example:

```
document.getElementById("myElement").textContent = "New text content";
```

3. **setAttribute** :

- Purpose: Sets the value of an attribute on an element.
- Example:

```
document.getElementById("myElement").setAttribute("class", "newClass");
```

4. **style** :

- Purpose: Modifies the inline styles of an element.
- Example:

```
document.getElementById("myElement").style.color = "blue";
```

5. **classList** :

- Purpose: Provides methods to add, remove, or toggle CSS classes on an element.
- Examples:

```
document.getElementById("myElement").classList.add("newClass");
document.getElementById("myElement").classList.remove("oldClass");
```

6. **appendChild** :

- Purpose: Adds a new child element to an existing element.
- Example:

```
var newElement = document.createElement("p");
newElement.textContent = "New paragraph";
document.getElementById("parentElement").appendChild(newElement);
```

7. **removeChild** :

- Purpose: Removes a child element from its parent.
- Example:

```
var childToRemove = document.getElementById("childElement");
document.getElementById("parentElement").removeChild(childToRemove);
```

8. **setAttribute :**

- **Purpose:** Sets or changes the value of an attribute on an HTML element.
- **Example:**

```
document.getElementById("myElement").setAttribute("src", "new-image.jpg");
```

These methods provide a diverse set of tools for us —developers to manipulate HTML elements dynamically, whether it's updating content, changing styles, or modifying attributes. The choice of method depends on the specific requirement and the nature of the change you want to apply.

Example - using **setAttribute** to change an input field to a button:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Change Input to Button Example</title>
</head>
<body>

<input type="text" id="myInput" value="Type Something">
<button onclick="changeToButton()">Change to Button</button>

<script>
  function changeToButton() {
    // Get the input element
    var inputElement = document.getElementById("myInput");

    // Create a new button element
    var buttonElement = document.createElement("button");

    // Set attributes for the new button
    buttonElement.setAttribute("type", "button");
    buttonElement.setAttribute("onclick", "alert('Button Clicked!')");
    buttonElement.innerHTML = "Click Me";

    // Replace the input with the new button
    inputElement.parentNode.replaceChild(buttonElement, inputElement);
  }
</script>
```

```
</body>
</html>
```

In this example, an input field with the id "myInput" is initially present, alongside a button labeled "Change to Button." Clicking this button triggers the `changeToButton` function, wherein a new button is dynamically created using `createElement`. Key attributes (type and onclick) are set via `setAttribute`, and the input field is promptly replaced by this newly fashioned button using `replaceChild`. The outcome is a dynamic transformation, demonstrating the capability to swap an input field for a button upon clicking "Change to Button," complete with an onclick attribute for interactive functionality.

Adding HTML Elements:

1. `createElement` Method:

- **Purpose:** Creates a new HTML element.
- **Example:**

```
var newElement = document.createElement("div");
```

2. `appendChild` Method:

- **Purpose:** Appends a new child element to an existing element.
- **Example:**

```
var parentElement = document.getElementById("parent");
parentElement.appendChild(newElement);
```

3. `insertBefore` Method:

- **Purpose:** Inserts a new element before a specified existing element.
- **Example:**

```
var existingElement = document.getElementById("existing");
parentElement.insertBefore(newElement, existingElement);
```

4. `innerHTML` Property:

- **Purpose:** Sets or gets the HTML content inside an element.

- Example:

```
parentElement.innerHTML = "<p>New content</p>";
```

5. **insertAdjacentHTML** Method:

- Purpose: Inserts HTML into a specified position relative to the element.
- Example:

```
parentElement.insertAdjacentHTML("beforeend", "<p>New content</p>");
```

Deleting HTML Elements:

1. **removeChild** Method:

- Purpose: Removes a child element from its parent.
- Example:

```
var childElement = document.getElementById("child");
parentElement.removeChild(childElement);
```

2. **remove** Method (Modern Browsers):

- Purpose: Removes the element itself.
- Example:

```
var elementToRemove = document.getElementById("toRemove");
elementToRemove.remove();
```

3. **replaceChild** Method:

- Purpose: Replaces a child element with a new element.
- Example:

```
var newChildElement = document.createElement("span");
parentElement.replaceChild(newChildElement, oldChildElement);
```

4. **innerHTML** Property (Setting to an Empty String):

- Purpose: Sets the HTML content inside an element to an empty string, effectively

removing its content.

- **Example:**

```
parentElement.innerHTML = "";
```

5. **outerHTML** Property:

- **Purpose:** Replaces an element with its HTML content.
- **Example:**

```
var newHTML = "<p>New content</p>";  
parentElement.outerHTML = newHTML;
```

Query Selectors

Query Selectors allows developers to select and manipulate HTML elements in a document using CSS-like syntax. They provide a powerful and flexible way to target specific elements based on various criteria, such as element type, class, ID, or attribute.

Here are some common examples of using Query Selectors:

- **Selecting by Element Type:**

```
var paragraphs = document.querySelectorAll('p');
```

- **Selecting by Class Name:**

```
var elementsWithClass = document.querySelectorAll('.className');
```

- **Selecting by ID:**

```
var elementWithId = document.querySelector('#elementId');
```

- **Selecting by Attribute:**

```
var elementsWithAttribute = document.querySelectorAll('[data-custom]');
```

- **Combining Selectors:**

```
var complexSelection = document.querySelectorAll('ul li.active');
```

Query Selectors return either a NodeList (for `querySelectorAll`) or a single element (for `querySelector`). NodeList is a collection of nodes, which can be iterated through using methods like `forEach`.

In summary, Query Selectors provide a concise and versatile way to interact with HTML elements in a document, making it easier for developers to manipulate the content and structure of a webpage dynamically.

DOM Node & Methods

The DOM (Document Object Model) is a programming interface that represents the structure of a document as a tree of objects, where each object corresponds to a part of the document. A DOM Node is a fundamental interface in the DOM hierarchy, representing a generic node in the tree structure. All elements, attributes, and text content in an HTML or XML document are nodes.

Here are some key points about DOM Nodes and their methods:

Key Points:

1. Node Types:

- Nodes can have different types, such as elements, text nodes, attributes, comments, etc.
- The `nodeType` property is used to determine the type of a node.

2. Hierarchy:

- Nodes are organized in a hierarchical structure, forming a tree.
- The `parentNode` property allows you to access the parent node of a given node.
- The `childNodes` property provides a NodeList of child nodes.

3. Traversal:

- The `nextSibling` and `previousSibling` properties allow traversal to adjacent nodes.
- The `firstChild` and `lastChild` properties give access to the first and last child nodes.

Types Of Nodes

In the DOM (Document Object Model), nodes represent different parts of an HTML or XML document, forming a tree structure. There are various types of nodes, each serving a specific purpose. Here are the common types of nodes in the DOM:

1. Element Nodes:

- **Description:** Represent HTML or XML elements.
- **Access:** Accessed using methods like `getElementById`, `getElementsByName`, or `querySelector`.
- **Example:**
The `<div>` element is an example of an element node.

```
<div id="example">This is an element node</div>
```

2. Attribute Nodes:

- **Description:** Represent attributes of an HTML or XML element.
- **Access:** Attributes can be accessed through the `attributes` property of an element node.
- **Example:**
In this example, `src` and `alt` are attribute nodes of the `` element.

```

```

3. Text Nodes:

- **Description:** Contain the text content within an HTML or XML element.
- **Access:** Accessed through the `textContent` or `innerText` property of an element node.
- **Example:**
The text "This is a text node" is a text node within the `<p>` element.

```
<p>This is a text node</p>
```

4. Comment Nodes:

- **Description:** Represent comments within the HTML or XML document.
- **Access:** Accessed through the `comment` property of a comment node.
- **Example:**
The content within `<!--` and `-->` is a comment node.

```
<!-- This is a comment -->
```

5. Document Node:

- **Description:** Represents the entire document.
- **Access:** The document node is the entry point for accessing the DOM tree.
- **Example:**

The `<html>` element serves as the document node in this example.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Document Node Example</title>
  </head>
  <body>
    <p>This is the document node.</p>
  </body>
</html>
```

6. Document Type Node:

- **Description:** Represents the document type declaration.
- **Access:** Accessed through the `doctype` property of the document node.
- **Example:**

The `<!DOCTYPE html>` declaration is a document type node.

```
<!DOCTYPE html>
```

DOM Events

DOM events are interactions or occurrences that take place in a web page, such as a user clicking a button, pressing a key, resizing the browser window, or the content of an input field changing. The HTML DOM (Document Object Model) allows JavaScript to respond to these events, enabling developers to create interactive and dynamic web applications. Here's an overview of DOM events and how JavaScript can react to them:

Key Concepts:

1. Event Types:

- Events can be triggered by various actions, such as mouse clicks (`click`), keyboard presses (`keydown`, `keyup`), form submissions (`submit`), document loading (`load`), and more.

2. Event Targets:

- Events are associated with specific HTML elements, known as event targets. For example, a `click` event might be associated with a button, and a `change` event might be associated with a form input.

3. Event Handlers:

- JavaScript can respond to events by using event handlers. Event handlers are functions that get executed when a specific event occurs.

Reacting to Events:

1. Inline Event Handlers:

- You can define event handlers directly within HTML elements using inline attributes like `onclick`, `onmouseover`, etc.

```
<button onclick="myFunction()">Click me</button>
```

2. DOM Level 0 Event Handling:

- You can assign event handlers directly to JavaScript properties of DOM elements.

```
var button = document.getElementById("myButton");
button.onclick = function() {
    // Handle the click event
};
```

3. DOM Level 2 Event Handling:

- The `addEventListener` method is used to attach event handlers to elements. This method provides more flexibility and allows multiple handlers for the same event.

```
var button = document.getElementById("myButton");
button.addEventListener("click", function() {
    // Handle the click event
});
```

4. Event Object:

- Event handlers typically receive an event object that provides information about the event, such as the target element, mouse coordinates, key codes, etc.

```
button.addEventListener("click", function(event) {
    console.log("Button clicked!", event.target);
});
```

Common Events:

1. Click Event:

- Triggered when a mouse button is clicked.

2. Keydown and Keyup Events:

- Fired when a key on the keyboard is pressed or released.

3. Submit Event:

- Triggered when a form is submitted.

4. Change Event:

- Fired when the value of an input field changes.

5. Load Event:

- Occurs when a resource (like an image or script) and the entire page have finished loading.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Event Handling Example</title>
</head>
<body>

    <button id="myButton">Click me</button>

    <script>
        var button = document.getElementById("myButton");

        // Using DOM Level 2 event handling
        button.addEventListener("click", function() {
            alert("Button clicked!");
        });
    </script>

</body>
</html>
```

In this example, a click event handler is attached to a button using the `addEventListener` method. When the button is clicked, an alert is displayed.

Understanding DOM events and how to handle them is crucial for creating interactive and responsive web applications. Developers use events to capture user actions and trigger appropriate JavaScript functionality in response.

The `onload` and `onunload` functions:

The `onload` and `onunload` events are part of the HTML DOM (Document Object Model) and are used to execute JavaScript code when a document or a page finishes loading (`onload`) or unloading (`onunload`). These events are commonly used to perform actions when a user enters or leaves a webpage.

`onload` Event:

The `onload` event is triggered when a document or a webpage has finished loading. This event is often used to ensure that all resources, such as images and scripts, have been fully loaded before executing specific JavaScript code.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>onload Event Example</title>
    <script>
        window.onload = function() {
            // Code to execute after the page has fully loaded
            alert("Page loaded!");
        };
    </script>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

In this example, the `onload` event is used to display an alert when the page has finished loading.

onunload Event:

The `onunload` event is triggered just before a document or a webpage is about to be unloaded, such as when the user navigates away from the page or closes the browser tab. This event is often used to perform cleanup tasks or prompt the user for confirmation before leaving the page.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>onunload Event Example</title>
    <script>
        window.onunload = function() {
            // Code to execute before the page is unloaded
            alert("Goodbye! Come back soon.");
        };
    </script>
</head>
<body>
    <h1>Thanks for visiting!</h1>
</body>
</html>
```

In this example, the `onunload` event is used to display an alert just before the page is unloaded.

These events play a crucial role in managing the lifecycle of a web page and allow developers to execute code at specific points during the page's existence.

DOM Event Listeners

DOM Event Listeners provide a more flexible and powerful way to handle events compared to traditional event attributes (e.g., `onclick`). Event Listeners allow you to attach multiple event handlers to a single event, making your code more modular and easier to maintain.

Using `addEventListener`:

The `addEventListener` method is used to attach an event listener to an HTML element. It takes three parameters: the event type, the function to be executed when the event occurs, and an optional third parameter indicating whether the event should be captured during the event propagation phase.

Syntax:

```
element.addEventListener(eventType, eventHandler, useCapture);
```

- **eventType** : A string representing the type of event (e.g., "click", "keydown", "change").
- **eventHandler** : A function that will be called when the event occurs.
- **useCapture** : (Optional) A boolean value indicating whether to use the capturing phase (`true`) or the bubbling phase (`false` , default).

Example of Multiple Event Listeners:

Here's a code snippet demonstrating the use of multiple event listeners on a button. In this example, we have a button that changes its color and displays a message when clicked, and it resets to its default state when the mouse leaves it:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Multiple Event Listeners Example</title>
    <style>
        #myButton {
            padding: 10px;
            font-size: 16px;
            cursor: pointer;
        }
    </style>
</head>
<body>

    <button id="myButton">Click me</button>

    <script>
        // Get the button element
        var button = document.getElementById("myButton");

        // Event listener for the "click" event
        button.addEventListener("click", function() {
            // Change the button color
            button.style.backgroundColor = "green";
            // Display a message
            alert("Button clicked!");
        });

        // Event listener for the "mouseenter" event
        button.addEventListener("mouseenter", function() {

```

```
// Change the button color on mouse enter
button.style.backgroundColor = "yellow";
});

// Event listener for the "mouseleave" event
button.addEventListener("mouseleave", function() {
    // Reset the button color on mouse leave
    button.style.backgroundColor = "";
});
</script>

</body>
</html>
```

In this example:

- Clicking the button changes its color to green and triggers an alert.
- Hovering over the button changes its color to yellow.
- Moving the mouse away from the button resets its color to the default state.

Using multiple event listeners allows you to handle different aspects of user interaction separately, promoting cleaner and more organized code.

Event Bubbling & Event Capturing

Event bubbling and event capturing are two phases of event propagation in the HTML DOM. When an event occurs on an HTML element, it goes through these two phases:

1. Event Capturing (Capture Phase):

- In this phase, the event travels from the root of the DOM tree to the target element.
- Event handlers attached with `useCapture` set to `true` are triggered during this phase.

2. Event Bubbling (Bubbling Phase):

- In this phase, the event travels from the target element back up to the root of the DOM tree.
- Event handlers attached without specifying `useCapture` or with `useCapture` set to `false` are triggered during this phase.

Example of Event Capturing:

In the following example, we have a nested set of div elements, and we attach event listeners to the document capturing phase (`useCapture` set to `true`). When you click on the innermost div, you'll see that the event handlers for the capturing phase are triggered from the root to the target:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Event Capturing Example</title>
</head>
<body>

    <div id="outer" style="border: 1px solid red; padding: 10px;">
        Outer
        <div id="middle" style="border: 1px solid green; padding: 10px;">
            Middle
            <div id="inner" style="border: 1px solid blue; padding: 10px;">
                Inner
            </div>
        </div>
    </div>

    <script>
        document.getElementById("outer").addEventListener("click", function() {
            console.log("Outer Capturing");
        }, true);

        document.getElementById("middle").addEventListener("click", function() {
            console.log("Middle Capturing");
        }, true);

        document.getElementById("inner").addEventListener("click", function() {
            console.log("Inner Capturing");
        }, true);
    </script>

</body>
</html>
```

When you click on the "Inner" div, you'll see in the console that the capturing phase event handlers are triggered in the order: Outer Capturing, Middle Capturing, Inner Capturing.

Example of Event Bubbling:

In this example, event listeners are attached without specifying `useCapture` or with `useCapture` set to `false` . When you click on the innermost div, the event handlers are triggered in the bubbling phase from the target back up to the root:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Event Bubbling Example</title>
</head>
<body>

    <div id="outer" style="border: 1px solid red; padding: 10px;">
        Outer
        <div id="middle" style="border: 1px solid green; padding: 10px;">
            Middle
            <div id="inner" style="border: 1px solid blue; padding: 10px;">
                Inner
            </div>
        </div>
    </div>

    <script>
        document.getElementById("outer").addEventListener("click", function() {
            console.log("Outer Bubbling");
        });

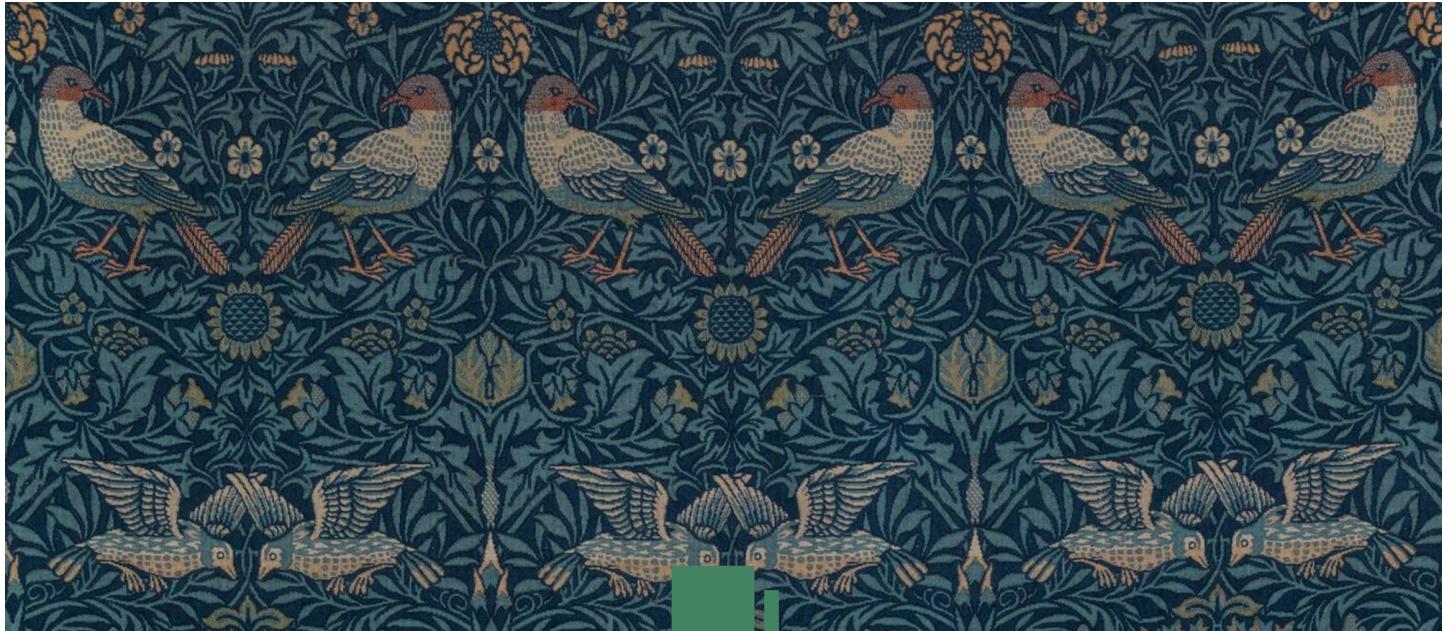
        document.getElementById("middle").addEventListener("click", function() {
            console.log("Middle Bubbling");
        });

        document.getElementById("inner").addEventListener("click", function() {
            console.log("Inner Bubbling");
        });
    </script>

</body>
</html>
```

When you click on the "Inner" div, you'll see in the console that the bubbling phase event handlers are triggered in the order: Inner Bubbling, Middle Bubbling, Outer Bubbling.

In practice, event bubbling is more commonly used, and the `useCapture` parameter is often omitted or set to `false` when attaching event listeners. Event capturing is less commonly used and is mainly applicable in specific scenarios where capturing is explicitly needed.



Week 4.1

Introduction to DOM

In this lecture, Harkirat lays a solid foundation for the upcoming Frontend Lectures, covering crucial topics such as `ECMA` script, `DOM manipulation`, methods for `backend interaction`, and the importance of `debouncing`. This session serves as a foundation for a deeper understanding of front-end development principles.

Introduction to DOM

[ECMA Scripts](#)

[Auxiliary APIs](#)

[Understanding Document](#)

[Manipulating HTML with JS](#)

[Understanding Rendering Via DOM](#)

[Classes vs IDs](#)

[Methods to Select Elements](#)

[1. querySelector\(\):](#)

[2. getElementById\(\):](#)

[3. getElementsByClassName\(\):](#)

[Communicating to the Backend Server](#)

[The onInput\(\) Function](#)

[Understanding Debouncing](#)

Throttling vs Rate Limiting

Throttling:

Rate Limiting:

Key Differences:

ECMA Scripts

ECMAScript, often abbreviated as ES, is a scripting language specification that serves as the standard upon which JavaScript is based. Basically, consider them as a set of rules and guidelines that make sure JavaScript behaves in a certain way. It's like a manual that tells developers what features JavaScript should have and how it should work.

Auxiliary APIs

Auxiliary APIs, in the context of web development, refer to additional interfaces and functionalities provided by browsers or runtime environments beyond the core JavaScript language (as specified by ECMAScript). These APIs extend the capabilities of JavaScript, enabling developers to interact with various aspects of the browser environment or perform tasks that go beyond the language's basic features.

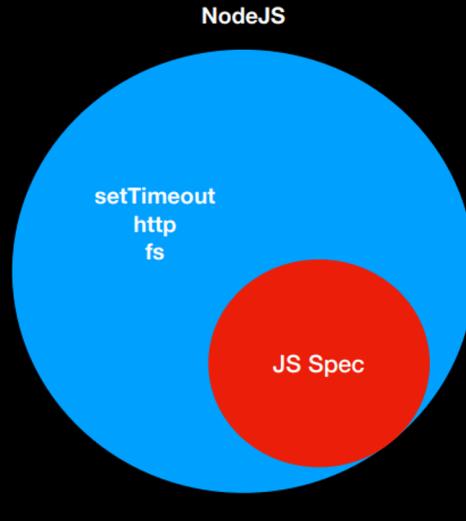
Here are some examples:

1. Node.js APIs:

- In the context of server-side JavaScript using Node.js, there are APIs specific to Node.js that provide access to the file system, networking, and other server-related functionalities.

```
// Example using the Node.js fs module for file system operations
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

But the Javascript that runs in your browser has some extra functionality



1. Third-Party APIs:

- APIs provided by external services or libraries that developers can use to enhance their applications. Examples include Google Maps API, Twitter API, or any other API that allows integration with external services.

```
// Example using the Google Maps API
const map = new google.maps.Map(document.getElementById('map'), {
  center: { lat: -34.397, lng: 150.644 },
  zoom: 8
});
```

1. Web APIs:

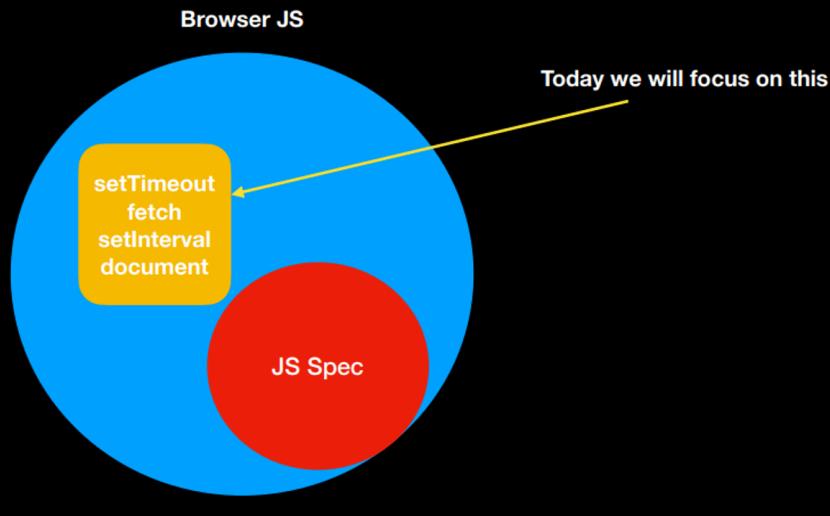
- These are browser-specific APIs that provide additional functionality to JavaScript for interacting with the browser environment. Examples include the DOM (Document Object Model), Fetch API for making network requests, and the Web Storage API for local storage.

```
// Example using the Fetch API
fetch('<https://api.example.com/data>')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

These APIs are not part of the JavaScript language specification (ECMAScript) but are essential for building web applications, interacting with external services,

and handling server-side operations. They extend the capabilities of JavaScript in specific environments.

But the Javascript that runs in your browser has some extra functionality



Further, in these topics of interest, we have already covered `setTimeout`, `fetch` and `setInterval`. Thus, our major focus today will be on understanding the Document.

Understanding Document

In JavaScript, the Document refers to the root object of the DOM.

The DOM (Document Object Model) API is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as a tree of objects; each object represents a part of the page.

The `document` object provides an entry point to the DOM, and it represents the entire HTML or XML document. Developers can use methods and properties provided by the `document` object to interact with and manipulate the content of a web page dynamically.

Manipulating HTML with JS

The DOM (Document Object Model) allows JavaScript to manipulate the HTML of a web page.

Imagine the DOM as a tree-like structure that represents your HTML document. Each element in your HTML, like buttons, paragraphs, and images, is a part of this tree. JavaScript can interact with this

tree, changing, adding, or removing elements. It's like giving JavaScript the power to update what you see on a webpage.

Code Example:

Let's create a simple button that, when clicked, changes the content of a paragraph:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM Magic</title>
</head>
<body>

  <!-- Our HTML with a button and an empty paragraph -->
  <button id="simpleButton">Click me!</button>
  <p id="output"></p>

  <!-- Our JavaScript -->
  <script>
    // Get the button and the paragraph by their IDs
    const simpleButton = document.getElementById('simpleButton');
    const outputParagraph = document.getElementById('output');

    // Add a click event listener to the button
    simpleButton.addEventListener('click', function() {
      // Change the content of the paragraph when the button is clicked
      outputParagraph.textContent = 'Awesome! You clicked the button!';
    });
  </script>

</body>
</html>
```

In this example:

- We have an HTML file with a button (`simpleButton`) and an empty paragraph (`output`).
- JavaScript code at the bottom gets references to these elements using `document.getElementById`.
- An event listener is added to the button. When clicked, the event listener function is triggered.
- Inside the function, we use `textContent` to change the content of the paragraph, making something appear on the page.

When you open this HTML file in a browser, clicking the button will make the paragraph magically display a message. This showcases how the DOM allows JavaScript to interact with and modify the

content of a web page dynamically.

Understanding Rendering Via DOM

Let's create a simple HTML page with two input fields, a button, and JavaScript code to calculate the sum and render it when the button is clicked.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Calculator</title>
</head>
<body>

  <!-- Our HTML with two input fields, a button, and an empty paragraph -->
  <label for="num1">Number 1:</label>
  <input type="number" id="num1" placeholder="Enter a number">

  <label for="num2">Number 2:</label>
  <input type="number" id="num2" placeholder="Enter another number">

  <button id="calculateButton">Calculate Sum</button>
  <p id="sumResult"></p>

  <!-- Our JavaScript to perform the calculation -->
<script>
  // Get input fields, button, and result paragraph by their IDs
  const num1Input = document.getElementById('num1');
  const num2Input = document.getElementById('num2');
  const calculateButton = document.getElementById('calculateButton');
  const sumResultParagraph = document.getElementById('sumResult');

  // Add a click event listener to the button
  calculateButton.addEventListener('click', function() {
    // Get the values from input fields and calculate the sum
    const num1 = parseFloat(num1Input.value) || 0; // Convert to number or default to 0
    const num2 = parseFloat(num2Input.value) || 0; // Convert to number or default to 0
    const sum = num1 + num2;

    // Display the result in the paragraph
    sumResultParagraph.textContent = `Sum: ${sum}`;
  });
</script>
```

```
</body>
</html>
```

Explanation:

1. We have two input fields (`num1` and `num2`), a button (`calculateButton`), and an empty paragraph (`sumResult`) in our HTML.
2. JavaScript code at the bottom gets references to these elements using `document.getElementById`.
3. An event listener is added to the button. When clicked, the event listener function is triggered.
4. Inside the function, it retrieves the values from the input fields, calculates the sum, and displays the result in the paragraph.

Now, when you enter numbers into the input fields and click the "Calculate Sum" button, it will compute the sum and display the result on the page.

Classes vs IDs

Classes:

- **Definition:** Used to group multiple HTML elements together.
- **Syntax (HTML):** `<element class="class-name">Content</element>`
- **Syntax (CSS):** `.class-name { /* styles */ }`
- **Usage:** Can be shared by multiple elements; an element can have multiple classes.

IDs:

- **Definition:** Used to uniquely identify a specific HTML element.
- **Syntax (HTML):** `<element id="unique-id">Content</element>`
- **Syntax (CSS):** `#unique-id { /* styles */ }`
- **Usage:** Must be unique within a page; often used for styling or JavaScript interaction.

Differences:

- **Uniqueness:**
 - Classes can be shared; IDs must be unique within a page.
- **Application:**

- Classes are for styling multiple elements.
- IDs are for styling a specific element or targeting with JavaScript.

In short, classes group elements, allowing shared styles, while IDs uniquely identify elements, often for specific styling or JavaScript interactions.

Methods to Select Elements

1. querySelector():

- **Definition:** `querySelector()` is a method that selects the first element that matches a specified CSS selector.
- **Syntax:** `document.querySelector('selector')`
- **Example:**

```
<div id="example">This is an example.</div>
<script>
  const element = document.querySelector('#example');
  element.style.color = 'blue';
</script>
```

- In this example, `querySelector('#example')` selects the element with the ID "example," and the text color is changed to blue.

2. getElementById():

- **Definition:** `getElementById()` is a method that selects a single element by its ID attribute.
- **Syntax:** `document.getElementById('id')`
- **Example:**

```
<div id="example">This is an example.</div>
<script>
  const element = document.getElementById('example');
  element.style.color = 'red';
</script>
```

- In this example, `getElementById('example')` selects the element with the ID "example," and the text color is changed to red.

3. getElementsByClassName():

- **Definition:** `getElementsByClassName()` is a method that selects all elements with a specific class name.
- **Syntax:** `document.getElementsByClassName('class')`
- **Example:**

```
<p class="highlight">This is a highlighted paragraph.</p>
<p class="highlight">Another highlighted paragraph.</p>
<script>
  const elements = document.getElementsByClassName('highlight');
  for (const element of elements) {
    element.style.fontWeight = 'bold';
  }
</script>
```

- In this example, `getElementsByClassName('highlight')` selects all elements with the class "highlight," and their font weight is changed to bold.

Relationship:

- All three methods are used to select and manipulate HTML elements.
- `getElementById()` is specifically for selecting by ID.
- `getElementsByClassName()` selects by class name, but it returns a collection of elements.
- `querySelector()` is more flexible as it can select by any valid CSS selector and returns the first matching element.

Example Using All Three:

```
<div id="example">This is an example.</div>
<p class="highlight">This is a highlighted paragraph.</p>

<script>
  // Using getElementById
  const elementById = document.getElementById('example');
  elementById.style.color = 'blue';

  // Using getElementsByClassName
  const elementsByClass = document.getElementsByClassName('highlight');
  for (const element of elementsByClass) {
    element.style.fontWeight = 'bold';
```

}

```
// Using querySelector
const elementByQuery = document.querySelector('.highlight');
elementByQuery.style.backgroundColor = 'yellow';
</script>
```

In this example, all three methods are used to select elements by ID, class name, and a general CSS selector, respectively. Each selected element is then styled accordingly.

Communicating to the Backend Server

Let's try to understand the process of connecting the frontend and backend through a small example. In this scenario, the backend is already hosted on the internet. This implies that in the frontend, we only need to make an API call to the provided link to interact with the backend.

```
<html>
<script>
    function populateDiv() {
        // Retrieve values from input fields
        const a = document.getElementById("firstNumber").value;
        const b = document.getElementById("secondNumber").value;

        // Make a fetch API call to the backend
        fetch("<https://sum-server.100xdevs.com/sum?a=>" + a + "&b=" + b)
            .then(function(response) {
                // Parse the response as text
                response.text()
                    .then(function(ans) {
                        // Display the result in the "finalSum" div
                        document.getElementById("finalSum").innerHTML = ans;
                    });
            });
    });

    // Async function for populating the "finalSum" div
    async function populateDiv2() {
        // Retrieve values from input fields
        const a = document.getElementById("firstNumber").value;
        const b = document.getElementById("secondNumber").value;

        // Make a fetch API call to the backend using async/await
        const response = await fetch("https://sum-server.100xdevs.com/sum?a=" + a + "&b=" + b);
        const ans = await response.text();
    }
}</script>
```

```
// Display the result in the "finalSum" div
document.getElementById("finalSum").innerHTML = ans;
}

</script>
<body>
  <!-- Input fields for numbers -->
  <input id="firstNumber" type="text" placeholder="First number"></input> <br><br>
  <input id="secondNumber" type="text" placeholder="Second number"></input> <br><br>

  <!-- Button to trigger the calculation -->
  <button onclick="populateDiv()">Calculate sum</button> <br><br>

  <!-- Display area for the final sum -->
  <div id="finalSum"></div>
</body>
</html>
```

Explanation:

1. **Input Fields:** Two input fields (`firstNumber` and `secondNumber`) are provided for users to enter numeric values.
2. **Button:** The "Calculate sum" button is associated with the `populateDiv()` function, which will be triggered when the button is clicked.
3. **JavaScript Function (`populateDiv`):**
 - Retrieves the values entered by the user from the input fields.
 - Uses the `fetch` API to make a GET request to the specified backend server (`https://sum-server.100xdevs.com/sum`) with the provided parameters `a` and `b`.
 - Processes the response:
 - Converts the response to text.
 - Updates the content of the `finalSum` div with the calculated sum.
4. **Display Area:** The result is displayed in the `finalSum` div.
5. **Async Function (`populateDiv2`):**
 - The `async` keyword is added before the function declaration, indicating that the function will handle asynchronous operations using `await`.

resolve before moving on to the next line of code.

- The response is obtained from the fetch call using `await`, and then the response text is retrieved similarly.

By using `async/await`, the code becomes more concise and easier to read, making it a preferable alternative for handling promises. Both `populateDiv()` and `populateDiv2()` perform the same functionality, but the latter takes advantage of modern JavaScript syntax for better code organization.

The `onInput()` Function

The `onInput` function is an event handler in JavaScript that gets executed when the value of an input field is changed by the user. This event is triggered dynamically as the user types or modifies the content within the input field. The `onInput` event is commonly used to perform actions in real-time as the user interacts with the input element.

Understanding Debouncing

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, making them more efficient. In the context of `onInput` events, debouncing is often applied to delay the execution of certain actions (e.g., sending requests) until after a user has stopped typing for a specific duration.

Implementation:

The following example demonstrates debouncing in the `onInput` event to delay the execution of a function that sends a request based on user input.

```
<html>
  <body>
    <!-- Input field with onInput event and debouncing -->
    <input id="textInput" type="text" onInput="debounce(handleInput, 500)" placeholder="Type something...">

    <!-- Display area for the debounced input value -->
    <p id="displayText"></p>

    <script>
      // Debounce function to delay the execution of a function
      function debounce(func, delay) {
        let timeoutId;

        return function(...args) {
          if (timeoutId) {
            clearTimeout(timeoutId);
          }

          const executeFunc = () => func(...args);

          timeoutId = setTimeout(executeFunc, delay);
        };
      };
    </script>
  </body>
</html>
```

```

return function() {
  // Clear the previous timeout
  clearTimeout(timeoutId);

  // Set a new timeout
  timeoutId = setTimeout(() => {
    func.apply(this, arguments);
  }, delay);
}

// Function to handle the debounced onInput event
function handleInput() {
  // Get the input field's value
  const inputValue = document.getElementById("textInput").value;

  // Display the input value in the paragraph
  document.getElementById("displayText").innerText = "You typed: " + inputValue;

  // Simulate sending a request (replace with actual AJAX call)
  console.log("Request sent:", inputValue);
}

</script>
</body>
</html>

```

Explanation:

- The `debounce` function is a generic debounce implementation that takes a function (`func`) and a delay time (`delay`).
- Inside the `debounce` function, a timeout is set to delay the execution of the provided function (`func`) by the specified delay time (`delay`).
- The `handleInput` function is the actual function to be executed when the `onInput` event occurs. It simulates sending a request (e.g., an AJAX call) based on user input.

How it works:

- When a user types in the input field, the `onInput` event triggers the `debounce` function.
- The `debounce` function sets a timeout, and if the user continues typing within the specified delay time, the previous timeout is cleared, and a new one is set.
- After the user stops typing for the specified delay, the `handleInput` function is executed.

This ensures that the function associated with the `onInput` event is not called on every keystroke but rather after the user has stopped typing for a brief moment, reducing unnecessary and potentially resource-intensive calls, such as sending requests.

Throttling vs Rate Limiting

Throttling:

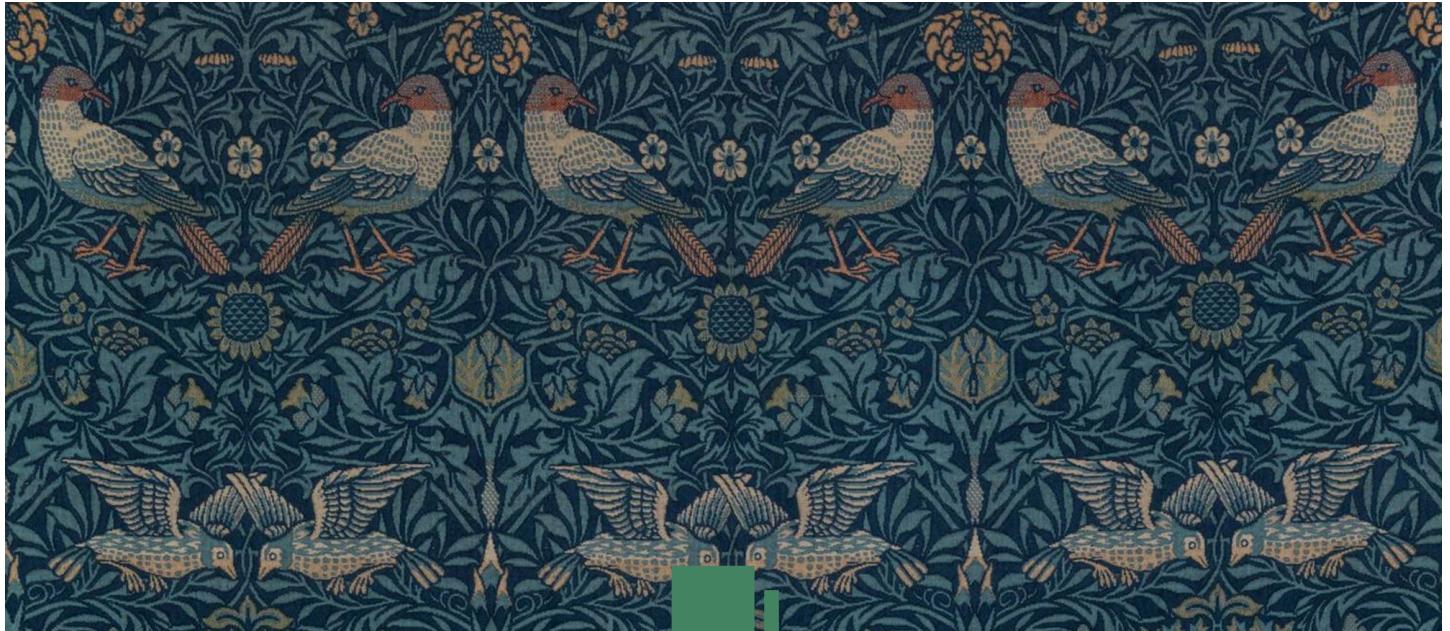
- **Definition:** Controls the rate at which a specific action is performed.
- **Purpose:** Ensures a smooth user experience, preventing rapid consecutive actions.
- **Implementation:** Limits the frequency of a particular function within a specified time frame.

Rate Limiting:

- **Definition:** Controls the number of requests a client can make within a specific time period.
- **Purpose:** Protects server resources, avoids abuse, and maintains fair usage.
- **Implementation:** Typically applied at the server/API level, limiting requests per second or minute.

Key Differences:

- Throttling focuses on action frequency; rate limiting focuses on request count.
- Throttling can be applied to various actions; rate limiting is often used at the API level.
- Throttling aims for a smooth user experience; rate limiting protects server resources and enforces fairness.



Week 4.2

Reconcilers & Intro to React

In this lecture, Harkirat addresses the challenges encountered in vanilla JavaScript while building a Todo application. Focusing on the `limitations of manual DOM manipulation` and the `lack of a centralized state` the discussion sets the context for transitioning to React. The session highlights the pain points faced during development and introduces `React's declarative and component-based approach as a solution` for more efficient and scalable web development.

[Reconcilers & Intro to React](#)

[Why React?](#)

[Todo Application Frontend](#)

[The Current Approach faces Several Challenges:](#)

[What do we mean by State?](#)

[Solution With a Blackbox Fn](#)

[The useState\(\) function](#)

[Virtual DOM](#)

[Complete Solution of the Todo Application](#)

[Conclusions](#)

[Challenges Faced Throughout](#)

[Starting with React](#)

[Building our First Application in React](#)

Why React?

DOM manipulation, in its raw form, poses significant challenges for developers. Constructing dynamic websites using the basic primitives offered by the DOM—such as `document.createElement`, `document.appendChild`, `element.setAttribute`, and `element.children`—can be a complex and labor-intensive process. The inherent difficulty lies in orchestrating intricate interactions and updates within the document structure using these primitive tools.

Recognizing the intricacies involved, React, a JavaScript library, emerged as a powerful solution. React abstracts away much of the manual DOM manipulation complexity, providing developers with a declarative and component-based approach to building user interfaces. This abstraction not only enhances code readability and maintainability but also simplifies the creation of dynamic and interactive web applications.

However, before jumping directly into React, let's us first try building a simple todo application using vanilla JavaScript. This exercise will serve a dual purpose—it will not only reinforce your JavaScript concepts but also allow you to grasp the problem statement, setting the stage to appreciate the elegance of React even more.

Todo Application Frontend

In today's session, our focus shifts to constructing a straightforward todo application. Unlike the previous day's example where we were overwriting the `innerHTML` to display results of `calculateSum`, today's task involves a different approach. We aim to append elements instead of overwriting them, creating a more seamless user experience. The goal is to have all entered todos consistently displayed on the screen, eliminating the abrupt resetting of content and ensuring a continuous and fluid representation of the user's input.

First, let's take a comprehensive look at the entire code for building the frontend of a Todo Application in vanilla JavaScript. Later, we will delve deep into each line and component of the code to gain a thorough understanding of its workings.

```
<body>
  <input type="text" id="title" placeholder="Todo title"></input> <br /><br />
  <input type="text" id="description" placeholder="Todo description"></input> <br /><br />

  <button onclick="addTodo()">Add todo</button>
  <br /> <br />
```

```
<div id="todos">

</div>
</body>

<script>
let globalId = 1;

function markAsDone (todoId) {
  const parent = document.getElementById(todoId);
  parent.children[2].innerHTML = "Done!"
}

function createChild(title, description, id) {
  const child = document.createElement("div");

  const firstGrandParent = document.createElement("div");
  firstGrandParent.innerHTML = title;

  const secondGrandParent = document.createElement("div");
  secondGrandParent.innerHTML = description;

  const thirdGrandParent = document.createElement("button");
  thirdGrandParent.innerHTML = "Mark as done";
  thirdGrandParent.setAttribute("onclick", `markAsDone(${id})`);

  child.appendChild(firstGrandParent);
  child.appendChild(secondGrandParent);
  child.appendChild(thirdGrandParent);
  child.setAttribute("id", id);

  return child;
}

function addTodo() {
  const title = document.getElementById("title").value;
  const description = document.getElementById("description").value;
  const parent = document.getElementById("todos");
  parent.appendChild(createChild(title, description, globalId++));
}
</script>
```

Now, Let's break down the JavaScript code step by step:

1. Global Variables:

```
let globalId = 1;
```

- `globalId` is a variable initialized to 1. It is used to assign a unique identifier to each todo item.

2. `markAsDone` Function:

```
function markAsDone(todoId) {
  const parent = document.getElementById(todoId);
  parent.children[2].innerHTML = "Done!";
}
```

- `markAsDone` is a function that marks a todo as done.
- It takes the `todoId` as a parameter, which corresponds to the unique identifier of the todo item.
- It finds the todo item using `document.getElementById(todoId)` and then accesses its third child element (index 2) to update its HTML content to "Done!".

3. `createChild` Function:

```
function createChild(title, description, id) {
  const child = document.createElement("div");

  const firstGrandParent = document.createElement("div");
  firstGrandParent.innerHTML = title;

  const secondGrandParent = document.createElement("div");
  secondGrandParent.innerHTML = description;

  const thirdGrandParent = document.createElement("button");
  thirdGrandParent.innerHTML = "Mark as done";
  thirdGrandParent.setAttribute("onclick", `markAsDone(${id})`);

  child.appendChild(firstGrandParent);
  child.appendChild(secondGrandParent);
  child.appendChild(thirdGrandParent);
  child.setAttribute("id", id);

  return child;
}
```

- `createChild` is a function that generates a new todo item.
- It takes `title`, `description`, and `id` as parameters.
- It creates three child elements (`firstGrandParent`, `secondGrandParent`, and `thirdGrandParent`) representing the title, description, and "Mark as done" button, respectively.
- These child elements are appended to the `child` div, and the unique identifier (`id`) is set as the id attribute for the child div.
- The function returns the generated child div.

4. `addTodo` Function:

```
function addTodo() {
  const title = document.getElementById("title").value;
  const description = document.getElementById("description").value;
  const parent = document.getElementById("todos");
  parent.appendChild(createChild(title, description, globalId++));
}
```

- `addTodo` is a function that adds a new todo item to the list.
- It retrieves the values of the title and description from the corresponding input fields.
- It then gets the parent container (`todos`) and appends a new todo item using the `createChild` function.
- The `globalId` is passed as the unique identifier, and it is then incremented for the next todo.

This setup allows users to add todo items with a title, description, and a "Mark as done" button that, when clicked, updates the todo's status to "Done!".

The Current Approach faces Several Challenges:

1. Difficulty in Adding and Removing Elements:

The process of adding and removing elements becomes intricate with the existing setup. Direct manipulation of the DOM for such operations can lead to complex and error-prone code.

2. Lack of Central State:

The absence of a centralized state management system poses issues. With each function managing its own state, maintaining consistency across the application becomes challenging.

3. Integration with a Server:

If there is a server where these todos are stored, the current structure lacks a mechanism to seamlessly integrate with it. Handling data from an external server requires a more organized approach.

4. Mobile App Updates:

Imagine updating a TODO item from a mobile app. When the updated list of TODOs arrives on the frontend, there is no provision for efficiently updating the DOM. The current structure lacks functions for updating or removing existing TODOs.

In essence, while there's an `addTodo` function, the absence of `updateTodo` and `removeTodo` functions limits the flexibility and robustness of the application, especially in scenarios involving dynamic changes and external data sources.

What do we mean by State?

When we refer to `state` in the context of our todo application, we are essentially talking about the current representation of the todo data within the application. In a more structured format, the `state` might look something like this:

```
[  
  {  
    id: 1,  
    title: "Go to Gym",  
    description: "Go to Gym from 7-9 PM"  
  },  
  {  
    id: 2,  
    title: "Read a Book",  
    description: "Read 'The Hitchhiker's Guide to the Galaxy'"  
  }  
]
```

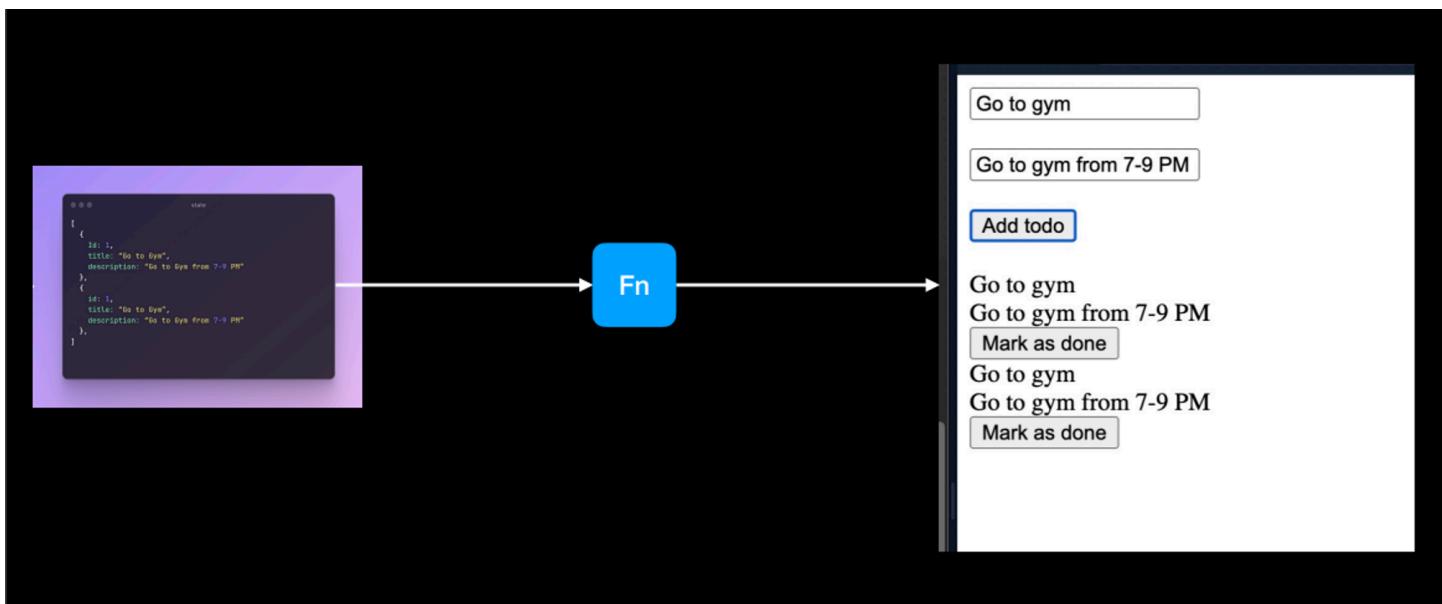
Here, the `state` is a collection of todo items, each represented as an object with properties such as 'id', 'title', and 'description'. This structured format allows us to organize and manage the current state of the application's data.

In a real-world scenario, as the user interacts with the application—adding, updating, or removing todos—the `state` dynamically changes to reflect the most recent data. This concept of `state management` becomes crucial for maintaining a consistent and up-to-date representation of the todo list within the application.

Solution With a Blackbox Fn

Now, if we can write a function, that takes this state as an input and creates the output, that would be much more powerful than our original approach.

When we talk about a function that takes a state as input and creates the desired output, we are referring to a function that can dynamically generate the representation of our todo list based on the current state.



Now for a moment let us consider this function `Fn` — `updateState()` to be a black box. This boils down our todo application code to:

```

<!DOCTYPE html>
<html>

<head>
  <script>
    // Global variable to assign unique identifiers to each todo
    let globalId = 1;

    // Function to add a new todo to the state
    function addTodo() {
      // Initialize an empty array to store todos
      let todoState = [];

```

```
// Get values from input fields
const title = document.getElementById("title").value;
const description = document.getElementById("description").value;

// Add a new todo object to the state array
todoState.push({
  title,
  description,
  id: globalId++
});

// Call the function to update the state and render the todos
updateState(todoState);
}

</script>
</head>

<body>
  <!-- Input fields for todo title and description -->
  <input type="text" id="title" placeholder="Todo title"></input> <br /><br />
  <input type="text" id="description" placeholder="Todo description"></input> <br /><br />

  <!-- Button to trigger the addTodo function -->
  <button onclick="addTodo()">Add todo</button>

  <!-- Container to display the list of todos -->
  <div id="todos"></div>
</body>

</html>
```

Explanation:

1. The `addTodo` function is responsible for adding a new todo to the state.
2. It initializes an empty `todoState` array to store todos.
3. It retrieves the values of the todo title and description from the corresponding input fields.
4. A new todo object is created with a unique identifier (`id`) using the `globalId` variable.
5. The new todo object is pushed to the `todoState` array.
6. The `updateState` function or `Fn` (which is not provided in this code) is called to update the state and render the todos on the screen.
7. The HTML structure consists of input fields for the todo title and description, a button to add a new todo, and a container (`div`) to display the list of todos.

The `updateState()` function

Now Let us focus on this blackbox `Fn` — the `updateState()` function

A Dumb way to constructing the `updateState()` function would be:

1. Clear the Parent Element:

Remove all child elements from the parent element.

2. Repopulate the DOM:

Call `addTodo()` for each element in the state, effectively re-rendering all todos.

However, a more intelligent solution is to:

- **Avoid Clearing the DOM Upfront:**

Instead of clearing the entire parent element and starting from scratch, maintain the existing DOM elements.

- **Update Based on Changes:**

Update the DOM selectively based on what has changed in the state.

Question Arises:

- How does the application determine what has changed?
- Has a todo been marked as complete?
- Has a todo been removed from the backend?

Solution: Keep track of the old todos in a variable, essentially creating a "Virtual DOM.

Virtual DOM

The concept of a `Virtual DOM` comes into play when dealing with efficient updates to the actual DOM.

The Virtual DOM is a lightweight copy of the actual DOM. When updates are made to the state of an application, a new Virtual DOM is created with the changes. This Virtual DOM is then compared with the previous Virtual DOM to identify the differences (known as "diffing").

In the context of a todo application:

1. State Changes:

- If a todo has been marked as complete or removed from the backend, the state of the application changes.

2. Virtual DOM Comparison:

- The new state is used to create a new Virtual DOM.
- This new Virtual DOM is compared with the previous Virtual DOM.

3. Identifying Changes:

- The difference between the new and previous Virtual DOMs is determined.
- For example, if a todo has been marked as complete, the corresponding element in the Virtual DOM is updated to reflect this change.

4. Efficient Updates:

- Instead of clearing the entire parent element and re-rendering everything, the Virtual DOM helps identify specifically what has changed.

5. Selective DOM Manipulation:

- Only the elements that have changed are manipulated in the actual DOM.
- This process is more efficient than a naive approach of clearing and re-rendering the entire DOM.

By employing a Virtual DOM, React optimizes the process of updating the actual DOM, leading to better performance and a smoother user experience. This mechanism is a key feature of React and contributes to its popularity for building dynamic and responsive web applications.

Complete Solution of the Todo Application

Now, let us have a look at our final complete code and draw conclusions out of this exercise:

```
<!DOCTYPE html>
<html>
<head>
  <script>
    let globalId = 1;
```

```
let todoState = [];
let oldTodoState = [];

function addTodo() {
    const title = document.getElementById("title").value;
    const description = document.getElementById("description").value;

    todoState.push({
        title,
        description,
        id: globalId++
    });

    updateState(todoState);
}

function removeTodo(todo) {
    const element = document.getElementById(todo.id);
    element.parentElement.removeChild(element);
}

function updateTodo(oldTodo, newTodo) {
    const element = document.getElementById(newTodo.id);
    element.children[0].innerHTML = newTodo.title;
    element.children[1].innerHTML = newTodo.description;
    element.children[2].innerHTML = newTodo.completed ? "Mark as done" : "Done";
}

function updateState(newTodos) {
    // Calculate the difference between newTodos and oldTodos.
    // More specifically, find out what todos are -
    // 1. added
    // 2. deleted
    // 3. updated
    const added = newTodos.filter(newTodo => !oldTodoState.some(oldTodo => oldTodo.id === newTodo.id));
    const deleted = oldTodoState.filter(oldTodo => !newTodos.some(newTodo => newTodo.id === oldTodo.id));
    const updated = newTodos.filter(newTodo => oldTodoState.some(oldTodo => oldTodo.id === newTodo.id));

    // Call addTodo, removeTodo, updateTodo functions on each of the elements
    added.forEach(newTodo => addTodoElement(newTodo));
    deleted.forEach(oldTodo => removeTodoElement(oldTodo));
    updated.forEach(newTodo => updateTodoElement(newTodo));

    oldTodoState = [...newTodos];
}

function addTodoElement(newTodo) {
    const title = document.getElementById("title").value;
    const description = document.getElementById("description").value;
```

```

todoState.push({
  title: title,
  description: description,
  id: globalId++
});

updateState(todoState);
}

</script>
</head>
<body>
  <input type="text" id="title" placeholder="Todo title"><br><br>
  <input type="text" id="description" placeholder="Todo description"><br><br>
  <button onclick="addTodo()">Add todo</button><br><br>
  <div id="todos"></div>
</body>
</html>

```

Conclusions

To build a dynamic frontend website in the easiest way possible we need 3 things.

A function to :

1. Update a state variable here `addTodo()`
2. Delegate the task of figuring out difference in the DOM tree here `updateStateTodo()`
3. Tell the hefty function how to add, update and remove elements here `updateTodo()` and `removeTodo()`

Now, creating a function to Update a state variable `addTodo()` and a function to Update and Remove elements `updateTodo()` `removeTodo()` is the job of a frontend developer. While this task of a hefty function to figure out the difference in the DOM tree is done by React.

Challenges Faced Throughout

As we transition into React, it's crucial to reflect on the challenges faced while building the Todo application using vanilla JavaScript.

1. The manual manipulation of the DOM, appending and removing elements,
2. lack of a centralized state, and
3. the absence of efficient methods for handling updates posed significant hurdles.

The approach of overwriting the entire DOM or clearing elements upfront proved cumbersome, especially when dealing with dynamic changes such as marking a Todo as complete or removing items from the backend. React, as a declarative and component-based library, addresses these pain points directly.

Starting with React

By abstracting away the complexities of DOM manipulation, React provides a more efficient and maintainable way to handle state changes, updates, and dynamic rendering. The transition to React promises not only a cleaner and more organized code structure but also a streamlined approach to building interactive web applications. Let's delve into React's features and explore how it revolutionizes frontend development, simplifying the implementation of functionalities like updating Todos and managing state seamlessly.

Building our First Application in React

Let's start a new React project using Vite, you can follow these steps:

1. Open your terminal and run the following command to create a new Vite project:

```
npm create vite@latest
```

1. Follow the prompts to set up your project. You can choose the default settings for now.
2. Once the project is created, navigate to the project directory using:

```
cd your-project-name
```

1. Open the project in your code editor.

Now, let's build a simple React component with a button that increases the count dynamically on clicks using `useState`. Update the `src/App.jsx` file with the following code:

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

function App() {
```

```
// State variable 'count' and the function 'setCount' to update it
const [count, setCount] = useState(0);

// Function to handle button click and update the count
const handleButtonClick = () => {
  setCount(count + 1);
};

return (
  <div>
    <h1>Count: {count}</h1>
    <button onClick={handleButtonClick}>Increase Count</button>
  </div>
);
}

export default App;
```

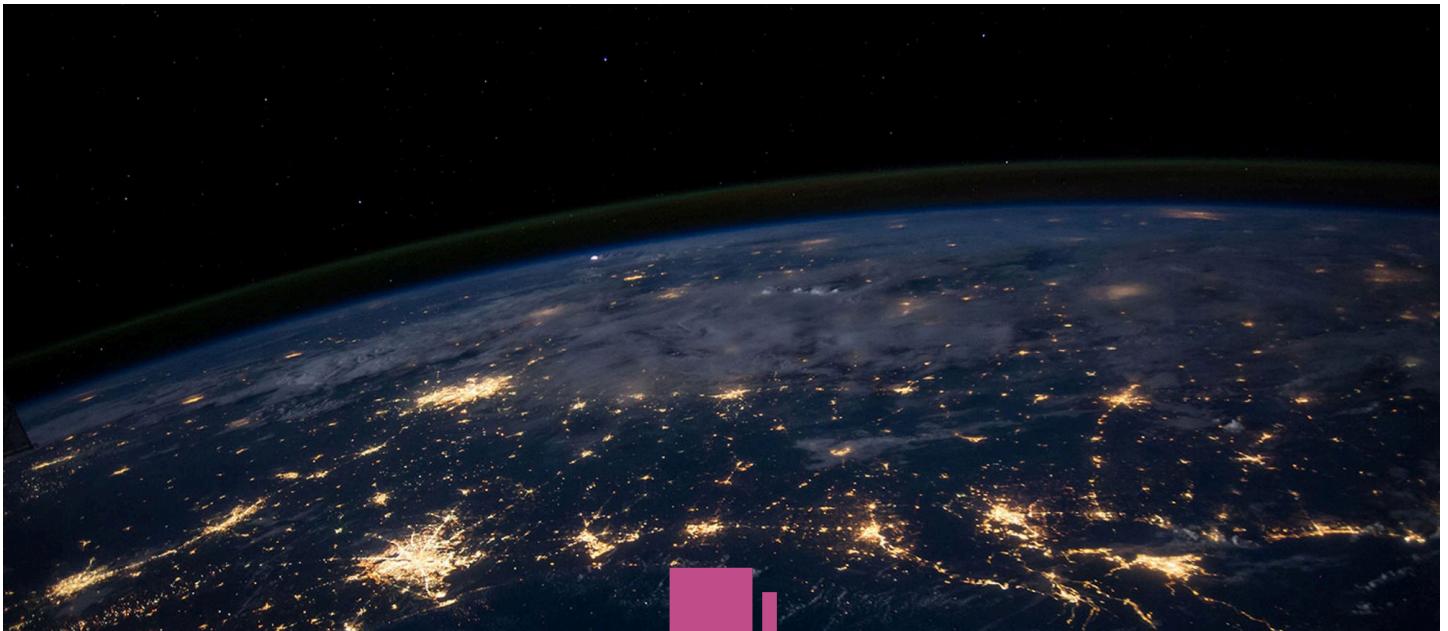
Explanation:

- We import `React` and the `useState` hook from the 'react' package.
- Inside the `App` component, we declare a state variable `count` and a function `setCount` to update it using `useState(0)`. The initial value of `count` is set to `0`.
- We define a function `handleButtonClick` that increases the `count` by 1 when the button is clicked. We use the `setCount` function to update the state.
- The JSX returned by the component displays the current count and a button. The `onClick` attribute of the button is set to the `handleButtonClick` function.

Now, save the file and run your project using:

```
npm run dev
```

Visit `http://localhost:3000` in your browser, and you should see your React app with a button that dynamically increases the count on each click.



Week 5.1

React Foundations

In this lecture, Harkirat explores the essential [React Jargons](#), focusing on [React Props](#), their properties, and the concept of [destructuring props](#) for efficient pass-downs. It's worth noting that this session adopts a slower pace, revisiting topics covered previously. If you're already familiar with these concepts, [feel free to skip](#) ahead to more advanced material covered later in the lecture.

[React Foundations](#)

[Diving into React](#)

[Understanding DOMs](#)

[1\] Virtual DOM](#)

[2\] Real DOM](#)

[Some React Jargon](#)

[1\] State](#)

[2\] Component](#)

[3\] Re-rendering](#)

[Simple Counter Application](#)

[React Props](#)

[1\] Passing Data:](#)

[2\] Functional Components:](#)

[3\] Class Components:](#)

- [4\] Immutable and Read-Only](#)
- [5\] Customization and Reusability](#)
- [6\] Default Values:](#)
- [7\] Destructuring Props:](#)
- [8\] Passing Functions as Props:](#)

Diving into React

As developers encountered challenges with traditional DOM manipulation, various libraries emerged to ease the process, with jQuery being one of them. However, even with such libraries, handling extensive applications remained complex.

Subsequently, Vue.js and React introduced a new syntax for front-end development. Behind the scenes, the React compiler transforms your code into HTML, CSS, and JavaScript, streamlining the development of large-scale applications.

Understanding DOMs

In React, there is a virtual DOM and a real DOM.

1] Virtual DOM

- React uses a virtual DOM to optimize updates and improve performance. The virtual DOM is an in-memory representation of the actual DOM elements. It's a lightweight copy of the real DOM.
- When you make changes to the state of a React component, React creates a new virtual DOM tree representing the updated state.
- React then compares the new virtual DOM with the previous virtual DOM to determine the differences (diffing).
- The differences are used to compute the most efficient way to update the real DOM.

2] Real DOM

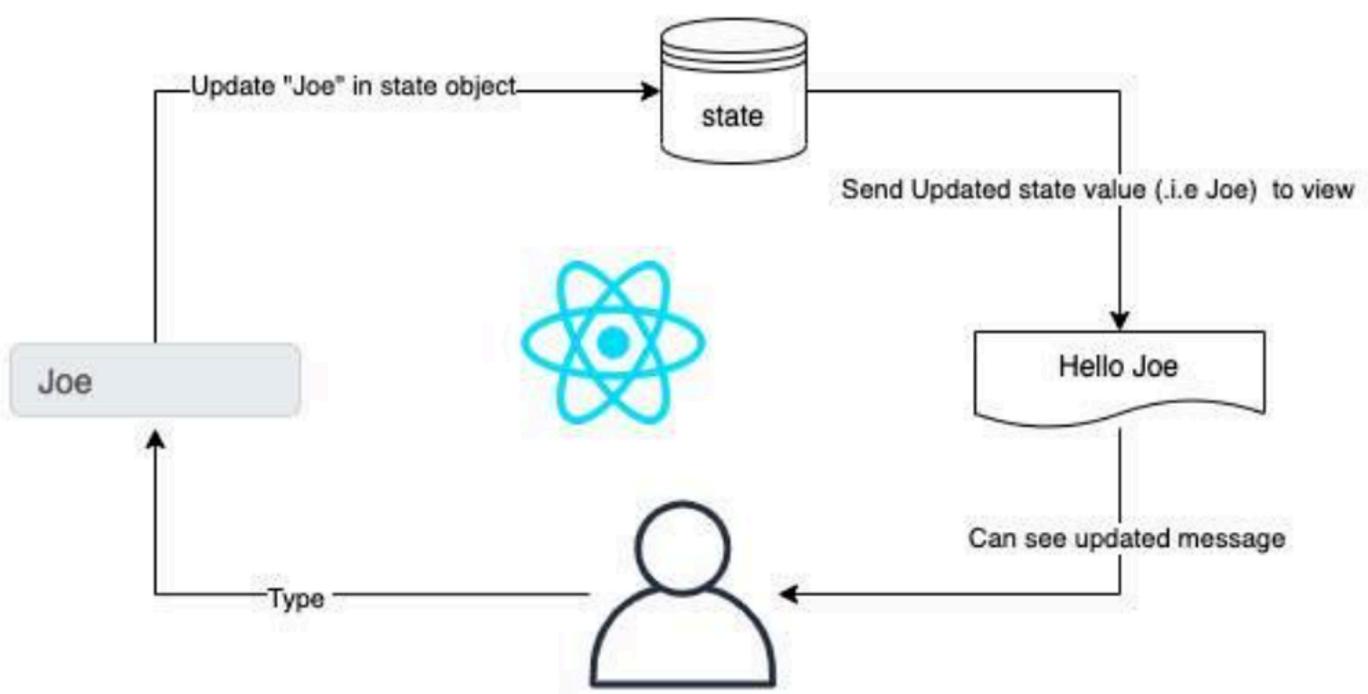
- The real DOM is the actual browser's Document Object Model, representing the structure of the HTML document.
- When React determines the updates needed based on the virtual DOM diffing process, it updates the real DOM with only the necessary changes.
- Manipulating the real DOM can be expensive in terms of performance, so React aims to minimize direct interaction with it.

In summary, while there are two representations—virtual DOM and real DOM—React abstracts the complexity of direct manipulation of the real DOM by using a virtual DOM and efficiently updating only the parts that have changed. This approach contributes to React's efficiency and performance in managing UI updates.

Some React Jargon

1] State

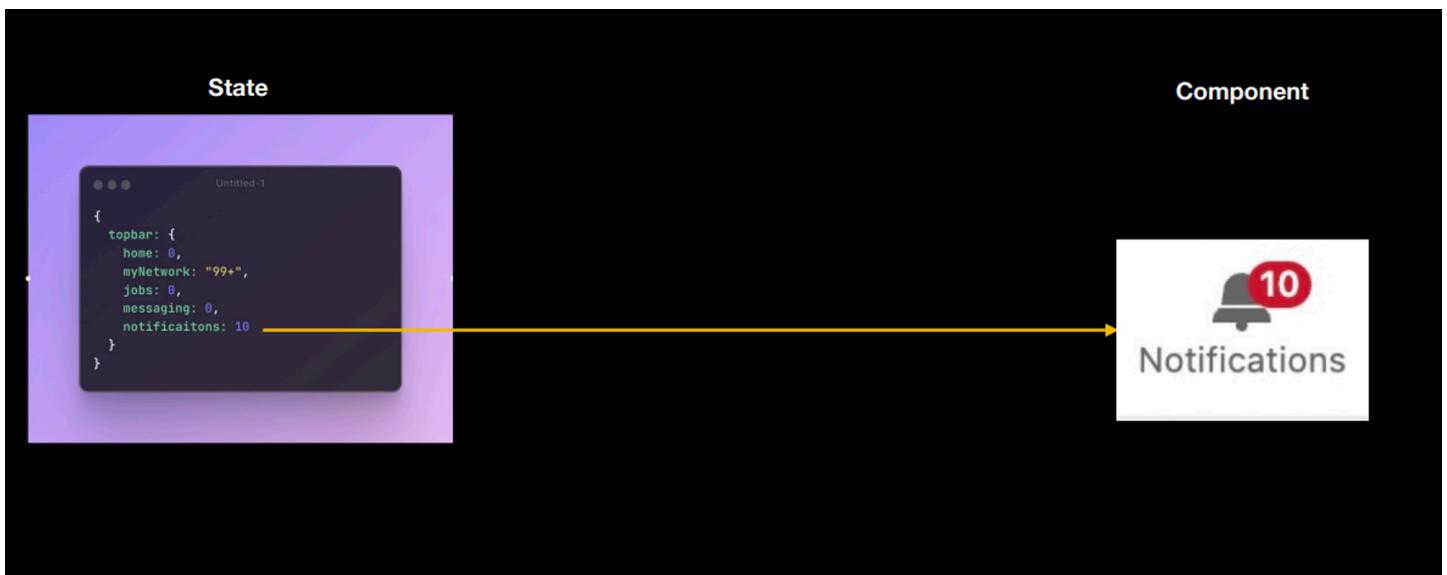
An object that represents the current state of the app. It represents the dynamic things in your app (things that change). For example, the value of the counter



2] Component

A React component is like a LEGO brick. Imagine you're building a spaceship with LEGO pieces. Each piece (component) has its own shape and color, and you can put them together to create the entire spaceship (user interface).

Components in React work in a similar way – they're like building blocks that you can use to make cool things on the computer, like games or websites. Each piece (component) does its own special job, and you can use them over and over again to build different things.



3] Re-rendering

Imagine you have a digital pet on your computer. This pet can be happy or sad based on how it's feeling. In React, we represent this feeling with something called "state."

So, you have a button that, when clicked, changes your pet's feeling from happy to sad or vice versa. When you click the button, React notices that the state (the feeling of your pet) has changed. When the state changes, React is smart enough to say, "Oh, something's different now. Let's update what's shown on the screen to match the new feeling."

So, if your pet was happily dancing, clicking the button might make it change to a sad face. This updating process is what we call "re-rendering." It's like refreshing the screen to show the new state of your digital pet.

In React, when the state changes, React automatically re-renders the component to reflect that change. This way, your users always see the most up-to-date and

accurate information on the screen.

Simple Counter Application

Let's start a new React project using Vite, you can follow these steps:

1. Open your terminal and run the following command to create a new Vite project:

```
npm create vite@latest
```

1. Follow the prompts to set up your project. You can choose the default settings for now.
2. Once the project is created, navigate to the project directory using:

```
cd your-project-name
```

1. Open the project in your code editor.

Now, let's build a simple **Let's create a simple Counter app in React to demonstrate re-rendering based on state changes**. Update the `src/App.jsx` file with the following code

```
import React, { useState } from 'react';
import './Counter.css'; // Assume you have a Counter.css file for styling

function Counter() {
  // Step 1: Define state using the useState hook
  const [count, setCount] = useState(0);

  // Step 2: Create functions to handle state changes
  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  // Step 3: The component renders based on the state
  return (
    <div className="counter">
      <h1>Counter App</h1>
      <p>Count: {count}</p>
    </div>
  );
}
```

```

    <button onClick={increment}>Increment</button>
    <button onClick={decrement}>Decrement</button>
  </div>
);
}

export default Counter;

```

1. State Initialization:

```
const [count, setCount] = useState(0);
```

- We use the `useState` hook to create a state variable `count` initialized to 0.
- `setCount` is a function we'll use to update the `count` state.

1. State Change Functions:

```

const increment = () => {
  setCount(count + 1);
};

const decrement = () => {
  setCount(count - 1);
};

```

- We create two functions (`increment` and `decrement`) that update the state when the buttons are clicked.

2. Rendered UI Based on State:

```

return (
  <div className="counter">
    <h1>Counter App</h1>
    <p>Count: {count}</p>
    <button onClick={increment}>Increment</button>
    <button onClick={decrement}>Decrement</button>
  </div>
);

```

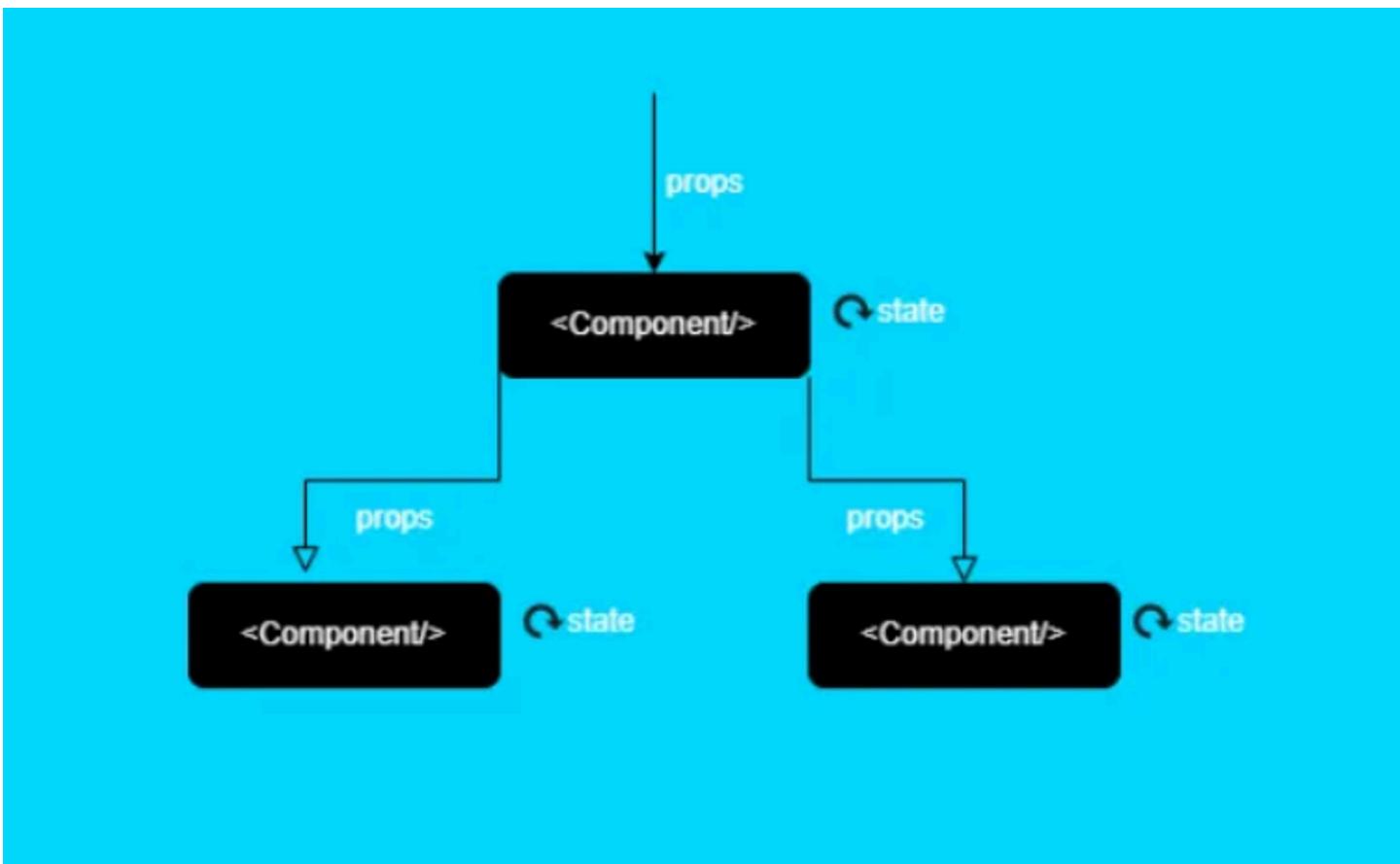
The component returns JSX, representing the UI.

- The UI shows the current count and two buttons for incrementing and decrementing the count.

Now, when you click the "Increment" or "Decrement" button, the `count` state changes, triggering a re-render of the component. The UI is updated to reflect the new count value. This is the essence of re-rendering based on state changes in a React component.

React Props

In React, props (short for properties) are a way to pass data from a parent component to a child component. They allow you to customize and configure child components based on values provided by their parent components.



Here's a breakdown of key concepts related to React props:

1] Passing Data:

- Props enable the flow of data from a parent component to a child component.
- They are passed as attributes in JSX when rendering a child component.

2] Functional Components:

- In functional components, props are received as an argument to the function.

```
function MyComponent(props) {  
  // Access props here  
}
```

3] Class Components:

- In class components, props are accessed using `this.props`.

```
class MyComponent extends React.Component {  
  render() {  
    // Access props using this.props  
  }  
}
```

4] Immutable and Read-Only

- Props in React are read-only. A child component cannot modify the props it receives from a parent. Props are intended to be immutable.

5] Customization and Reusability

- Props allow you to customize the behavior and appearance of a component, making it versatile and reusable in different contexts.

6] Default Values:

- You can provide default values for props to ensure that the component works even if certain props are not explicitly passed.

7] Destructuring Props:

- You can use destructuring to extract specific props from the `props` object, making code cleaner.

```
function MyComponent({ prop1, prop2 }) {  
  // Access prop1 and prop2 directly  
}
```

8] Passing Functions as Props:

- You can pass functions as props, allowing child components to communicate with their parent components.

Here's a simple example to illustrate the use of props:

```
// ParentComponent.js  
import React from 'react';  
import ChildComponent from './ChildComponent';  
  
function ParentComponent() {  
  return <ChildComponent message="Hello from Parent!" />;  
}  
  
// ChildComponent.js  
import React from 'react';  
  
function ChildComponent(props) {  
  return <p>{props.message}</p>;  
}  
  
export default ChildComponent;
```

In this example, `ParentComponent` passes the message "Hello from Parent!" to `ChildComponent` as a prop. The child component then displays this message. Props facilitate communication between components, making it easy to create modular and reusable React applications.



Week 5.2

React Project —Todo Application

Up until now, our discussions have primarily revolved around theoretical concepts. In this lecture, Harkirat takes a **practical approach** by guiding us through the hands-on process of **building a todo application**. We'll be applying the knowledge we've gained so far, specifically focusing on implementing the frontend using **React** and the backend using **Node.js**, **MongoDB**, and **Express** — creating a classic **MERN** stack application.

While there are **no specific notes** provided for this section, a mini guide is outlined below to assist you in navigating through the process of building the application. Therefore, it is strongly **advised to actively follow along** during the lecture for a hands-on learning experience.

React Project —Todo Application

Building a MERN Stack Todo Application with Zod

- 1] Environment Setup
- 2] Frontend Development (React)
- 3] Backend Development (Node.js, Express, MongoDB, Zod)
- 4] Zod Integration
- 5] Connect Frontend to Backend
- 6] Run the Application

Building a MERN Stack Todo Application with Zod

1] Environment Setup

- Install Node.js and npm on your machine.
- Set up a new React project using Vite: `npm create vite@latest`.
- Create a new Node.js project for the backend.

2] Frontend Development (React)

- Design the UI structure for your to-do app.
- Create React components for adding, displaying, and deleting todos.
- Utilize React Hooks (`useState`, `useEffect`) for managing the frontend state.
- Set up a clean and user-friendly interface for a seamless user experience.

3] Backend Development (Node.js, Express, MongoDB, Zod)

- Configure a Node.js and Express backend server.
- Integrate MongoDB for data storage.
- Implement Zod for backend data validation to ensure secure and valid inputs.
- Create API endpoints for handling todo operations like add, fetch, and delete.

4] Zod Integration

- Install Zod in your Node.js backend project using npm: `npm install zod`.
- Define schemas using Zod to validate incoming data.
- Integrate Zod validation within your API routes for robust data validation.

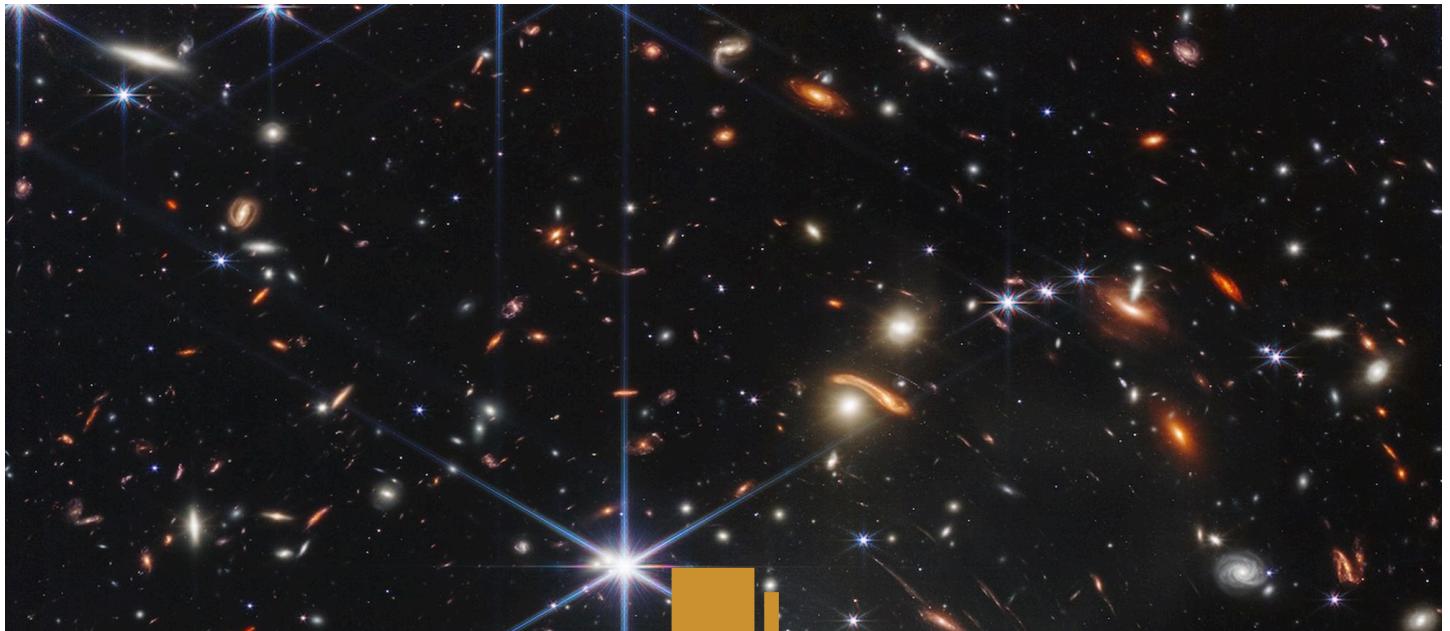
5] Connect Frontend to Backend

- Establish a connection between your React frontend and Node.js backend.
- Make API calls from the frontend to interact with the backend endpoints.

6] Run the Application

- Start both the React and Node.js servers.
- Open your to-do app in a browser and verify its functionality.

By following these steps, you'll gain practical insights into building a MERN stack application with Zod integration, ensuring both frontend and backend components work seamlessly.



Week 6.1

React Hooks

In this lecture, Harkirat explored essential concepts in React development, emphasizing the significance of a single parent component for efficient reconciliation and rerendering. The discussion delved into strategies for minimizing rerenders, highlighting the use of memoization with `useMemo` and the importance of keys in array iteration. The lecture also introduced the concept of wrapper components, showcasing their role in maintaining consistent styling. Furthermore, Harkirat provided insights into the choice between class-based and functional components, concluding with a brief overview of the `useEffect` hook. These insights offer a solid foundation for understanding core React principles.

[React Hooks](#)

[React Returns](#)

[Problem Statement](#)

[Reconciliation](#)

[Solution](#)

[Object Destructuring](#)

[Basic Object Destructuring:](#)

[Default Values:](#)

[Variable Assignment:](#)

[Nested Object Destructuring:](#)

[Re-rendering in React](#)

When Does a Rerender Happen?

Problem Statement

Solutions

Pushing the State Down:

By Using Memoization:

Significance of Key in React

Wrapper Components

Class Components vs Functional Components

React Hooks

useEffect()

React Returns

In React, a component can only return a single root element, commonly wrapped in a parent container (like a `div`). This rule exists because React needs a single entry point to render and manage the component's output.

Problem Statement

```
1
2  function App() {
3    return (
4      <Header title="my name is harkirat" />
5      <Header title="My name is raman" />
6    )
7  }
8
9  function Header({title}) {
10   return <div>
11   |   {title}
12   </div>
13 }
14
15 export default App
16
```



One of the most prominent reasons for it is **Reconciliation**. The single-root element rule in React facilitates the **reconciliation** process, where React efficiently updates the real DOM based on changes in the virtual DOM. By having a single root element, React can easily perform the comparison between the previous and current states of the **virtual DOM**.

Reconciliation

Reconciliation involves identifying what parts of the virtual DOM have changed and efficiently updating only those parts in the actual DOM. The single-root structure simplifies this process by providing a clear entry point for React to determine where updates should occur.

In addition to reconciliation, it aids in maintaining a straightforward and predictable structure in React components, making the code more readable and understandable. This constraint encourages developers to create components with well-defined boundaries, which enhances code organization and modularity.

While a single root element is required, React provides a feature called **fragments** (`<></>` or `<React.Fragment></React.Fragment>`) that allows you to

group multiple elements without introducing an extra node in the real DOM. Fragments don't create an additional parent in the DOM but still satisfy the single-root rule.

Solution

Doesn't introduce an extra DOM element

```
src > App.jsx > ...
1  function App() {
2    return (
3      <>
4        <Header title="my name is harkirat" />
5        <Header title="My name is raman" />
6      </>
7    )
8  }
9
10 function Header({title}) {
11   return <div>
12   | {title}
13   | </div>
14 }
15
16 export default App
17
18
```

Slightly better

```
src > App.jsx > App
1  function App() {
2    return (
3      <div>
4        <Header title="my name is harkirat" />
5        <Header title="My name is raman" />
6      </div>
7    )
8  }
9
10 function Header({title}) {
11   return <div>
12   | {title}
13   | </div>
14 }
15
16 export default App
17
18
```

```
const MyComponent = () => {
  return (
    <>
      <Header />
      <MainContent />
      <Footer />
    </>
  );
};
```

In summary, the single-root rule ensures a clear and efficient rendering process, simplifies styling and layout, and maintains consistency in React components.

Object Destructuring

Object destructuring is a feature in JavaScript that allows you to extract values from objects and assign them to variables in a more concise and convenient way. This can make your code cleaner and more readable. Here's a brief explanation with an example:

Basic Object Destructuring:

```
// Original Object
const person = { firstName: 'John', lastName: 'Doe', age: 30 };

// Destructuring
const { firstName, lastName, age } = person;

// Extracted Values
console.log(firstName); // Output: John
console.log(lastName); // Output: Doe
console.log(age); // Output: 30
```

Default Values:

You can also provide default values in case the property is not present in the object:

```
const { firstName, lastName, age, gender = 'Unknown' } = person;
console.log(gender); // Output: Unknown (since 'gender' is not present in the 'person' object)
```

Variable Assignment:

You can use a different variable name during destructuring:

```
const { firstName: first, lastName: last } = person;
console.log(first); // Output: John
console.log(last); // Output: Doe
```

Nested Object Destructuring:

Destructuring also works with nested objects:

```
const student = { name: 'Alice', details: { grade: 'A', age: 21 } };

const { name, details: { grade, age } } = student;

console.log(name); // Output: Alice
console.log(grade); // Output: A
console.log(age); // Output: 21
```

Object destructuring provides a concise and expressive way to extract values from objects, making your code more readable and maintainable.

Re-rendering in React

Rerendering in React refers to the process of updating and rendering components to reflect changes in the application's state or props. When there's a change in the state or props of a component, React re-renders that component and any affected child components. It's important to note that a rerender doesn't necessarily mean a complete re-rendering of the entire DOM; instead, React efficiently updates only the necessary parts of the DOM.

Basically, anytime a final DOM manipulation happens or when react actually updates the DOM it is called a rerender.

When Does a Rerender Happen?

1. Changes in a state variable utilized within the component.
2. A re-render of a parent component, which subsequently triggers the re-rendering of all its child components. This cascading effect ensures synchronization throughout the component tree.

Problem Statement

- As it is known, we use React to create dynamic websites, which is often achieved through the use of components that can respond to user interactions, state changes, or incoming props.
- A crucial principle guiding efficient React applications is the minimization of unnecessary rerenders. Rerenders occur when there are alterations in a component's state or props, and the rule of thumb is to keep these rerenders to a minimum for optimal performance.
- Consider a scenario with a webpage featuring a counter button, a text element reflecting the change in the `count` state and a static "Hello, World!" text. While clicking the counter button might trigger a rerender of the text element containing the `count` due to state changes, it's essential to prevent the unnecessary rerendering of static elements.

Solutions

There are broadly 2 ways of minimizing the amount of rerenders

1. Push the State down
2. By Using Memoization

Pushing the State Down:

Pushing the state down in React refers to the practice of managing state at the lowest possible level in the component tree. By doing so, you localize the state to the components that absolutely need it, reducing unnecessary re-renders in higher-level components.

When state is kept at a higher level in the component tree, any changes to that state can trigger re-renders for all child components, even if they don't directly use or depend on that particular piece of state. However, by **pushing the state down** and ensuring that each component only has access to the state it needs, you can minimize the impact of state changes on the overall component tree.

The diagram illustrates the concept of "Pushing the state down" through a code transformation. On the left, the initial code has state management at a higher level (App.js). On the right, after applying the technique, state management is moved to a lower level (HeaderWithButton.js), specifically to the Header component.

```

src > App.jsx > ...
1 import { useState } from "react"
2
3 function App() {
4   const [firstTitle, setFirstTitle] = useState("my name is harkirat");
5
6   function changeTitle() {
7     setFirstTitle("My name is " + Math.random());
8   }
9
10  return (
11    <div>
12      <button onClick={changeTitle}>Click me to change the title</button>
13      <Header title={firstTitle} />
14      <Header title="My name is raman" />
15    </div>
16  );
17}
18
19 function Header({title}) {
20  return <div>
21    <{title}>
22  </div>
23}
24
25 export default App
26 |
```

Pushing the state down →

```

Src > App.jsx > ...
1 import { useState } from "react"
2
3 function App() {
4   return (
5     <div>
6       <HeaderWithButton />
7       <Header title="My name is raman" />
8     </div>
9   )
10 }
11
12 function HeaderWithButton() {
13   const [firstTitle, setFirstTitle] = useState("my name is harkirat");
14
15   function changeTitle() {
16     setFirstTitle("My name is " + Math.random());
17   }
18
19   return <>
20     <button onClick={changeTitle}>Click me to change the title</button>
21     <Header title={firstTitle} />
22   </>
23 }
24
25 function Header({title}) {
26  return <div>
27    <{title}>
28  </div>
29}
30
31 export default App
32 |
```

For example, if a specific piece of state is only relevant to a small portion of your application, keeping that state localized to the components in that section prevents unnecessary re-renders elsewhere. This practice contributes to a more efficient and performant React application.

By Using Memoization:

The above problem of reducing the number of rerenders can also be tackled using Memoization. Memoization in React, achieved through the `useMemo` hook, is a technique used to optimize performance by memoizing (caching) the results of expensive calculations. This is particularly useful when dealing with computations that don't need to be recalculated on every render, preventing unnecessary recalculations and re-renders.

In the context of minimizing re-renders, `useMemo` is often employed to memoize the results of computations derived from state or props. By doing so, you can ensure that the expensive computation is only performed when the dependencies (specified as the second argument to `useMemo`) change.

```

src > App.jsx > ...
1 import { useState } from "react"
2
3 function App() {
4   const [firstTitle, setFirstTitle] = useState("my name is harkirat");
5
6   function changeTitle() {
7     setFirstTitle("My name is " + Math.random())
8   }
9
10  return (
11    <div>
12      <button onClick={changeTitle}>Click me to change the title</button>
13      <Header title={firstTitle} />
14      <Header title="My name is raman" />
15    </div>
16  )
17}
18
19 function Header({title}) {
20   return <div>
21     <title>
22   </div>
23 }
24
25 export default App
26

```



```

src > App.jsx > [e] default
1 import { useState } from "react"
2 import { memo } from "react";
3
4 function App() {
5   const [firstTitle, setFirstTitle] = useState("my name is harkirat");
6
7   function changeTitle() {
8     setFirstTitle("My name is " + Math.random())
9   }
10
11  return (
12    <div>
13      <button onClick={changeTitle}>Click me to change the title</button>
14      <Header title={firstTitle} />
15      <br />
16      <Header title="My name is raman" />
17      <Header title="My name is raman" />
18      <Header title="My name is raman" />
19      <Header title="My name is raman" />
20    </div>
21  )
22}
23
24 const Header = memo(function ({title}) {
25   return <div>
26     <title>
27   </div>
28 })
29
30 export default App
31

```

By using `useMemo`, you can strategically memoize computations to optimize performance and minimize the impact of re-renders in React.

Significance of Key in React

In React, when rendering a list of elements using the `map` function, it is crucial to assign a unique `key` prop to each element. The "key" is a special attribute that helps React identify which items have changed, been added, or been removed. This is essential for efficient updates and preventing unnecessary re-renders of the entire list.

When the `key` prop is not provided or not unique within the list, React can't efficiently track the changes, leading to potential issues in the application's performance and rendering.

Here's a simple example illustrating the importance of keys in a todo app:

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

const TodoList = ({ todos }) => (
  <ul>
    {todos.map(todo => (
      // Each todo item needs a unique key
      <li key={todo.id}>{todo.text}</li>
    ))}
  </ul>
);

const App = () => {
  const [todos, setTodos] = useState([
    { id: 1, text: 'Learn React' },
    { id: 2, text: 'Build a Todo App' },
    { id: 3, text: 'Deploy to production' },
  ]);

  const addTodo = () => {
    // Simulating adding a new todo
    const newTodo = { id: todos.length + 1, text: 'New Todo' };
    setTodos([...todos, newTodo]);
  };

  return (
    <div>
      <button onClick={addTodo}>Add Todo</button>
      <TodoList todos={todos} />
    </div>
  );
};

export default App;
```

In this example, each todo item in the list has a unique `id` that serves as the `key` prop. When a new todo is added, the `key` helps React efficiently update and re-render only the necessary parts of the list, maintaining performance and ensuring a smooth user experience.

Wrapper Components

In React, wrapper components are used to encapsulate and group common styling or thematic elements that need to be applied consistently across different parts of an application. These components act as containers for specific sections or functionalities, allowing for a clean and modular structure.

Let's consider an example where we have a wrapper component called `Card` that provides a consistent styling for various content sections, such as blog posts. The `Card` component maintains the overall styling, while different contents can be dynamically injected.

```
// CardWrapper.js

import React from 'react';

const CardWrapper = ({ children }) => {
  return (
    <div style={{ border: '1px solid #ccc', padding: '16px', margin: '16px', borderRadius: '8px' }}>
      {children}
    </div>
  );
}

export default CardWrapper;
```

Now, we can use this `CardWrapper` component to create specific cards for different content, such as blog posts, by providing the dynamic content as children:

```
// BlogPost.js

import React from 'react';
import CardWrapper from './CardWrapper';

const BlogPost = ({ title, content }) => {
  return (
    <CardWrapper>
      <h2>{title}</h2>
      <p>{content}</p>
    </CardWrapper>
  );
}

export default BlogPost;
```

With this structure, we maintain a consistent card styling across different sections of our application, promoting reusability and making it easy to manage the overall theme. This approach is especially beneficial when you want to keep a uniform appearance for similar components while varying their internal content.

Class Components vs Functional Components

In React, components are the building blocks of a user interface. There are two main types of components: class-based components and functional components.

1. Class-Based Components:

- Class-based components are ES6 classes that extend from `React.Component`.
- They have access to the lifecycle methods provided by React, such as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
- State and lifecycle methods are managed within class-based components.
- They were the primary type of components before the introduction of hooks in React 16.8.

Example of a class-based component:

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

class MyClassComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      // state initialization
    };
  }

  componentDidMount() {
    // code to run after component mounts
  }

  render() {
    return <div>Hello from class-based component</div>;
  }
}
```

```
export default MyClassComponent;
```

1. Functional Components:

- Functional components are simpler and more concise. They are essentially JavaScript functions that take props as an argument and return React elements.
- With the introduction of React hooks in version 16.8, functional components gained the ability to manage state and use lifecycle methods through hooks like `useState` and `useEffect`.
- They are generally easier to read and write.

Example of a functional component:

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

const MyFunctionalComponent = () => {
  const [state, setState] = useState(/* initial state */);

  useEffect(() => {
    // code to run after component mounts or when state/props change
  }, /* dependencies */);

  return <div>Hello from functional component</div>;
};

export default MyFunctionalComponent;
```

Note:

- Functional components are now the preferred way to write components in React due to their simplicity and the additional capabilities provided by hooks.
- Hooks like `useState` and `useEffect` allow functional components to manage state and perform side effects, making them as powerful as class-based components.
- Class-based components are still used in some codebases, especially in projects that haven't migrated to functional components or are working with older React versions.

React Hooks

React Hooks are functions that allow functional components in React to have state and lifecycle features that were previously available only in class components. Hooks were introduced in React 16.8 to enable developers to use state and other React features without writing a class.

Using these hooks, developers can manage state, handle side effects, optimize performance, and create more reusable and readable functional components in React applications. Each hook serves a specific purpose, contributing to a more modular and maintainable codebase.

Some commonly used React Hooks are: `useEffect`, `useMemo`, `useCallback`, `useRef`, `useReducer`, `useContext`, `useLayoutEffect`

useEffect()

`useEffect` is a React Hook used for performing side effects in functional components. It is often used for tasks such as data fetching, subscriptions, or manually changing the DOM. The `useEffect` hook accepts two arguments: a function that contains the code to execute, and an optional array of dependencies that determines when the effect should run.

```
useEffect(() => {
  fetch("https://sum-server.100xdevs.com/todos")
    .then(async (res) => {
      const json = await res.json();
      setTodos(json.todos);
    })
}, [])
```

Here's an example of how to use `useEffect`:

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

const DataFetcher = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    // Effect will run after the component is mounted
    const fetchData = async () => {
      try {
        // Simulating a data fetching operation
        const response = await fetch('https://api.example.com/data');
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };
    fetchData();
  });

  // Effect cleanup (will run before unmounting)
  return () => {
    console.log('Component will unmount. Cleanup here.');
  };
}, []); // Empty dependency array means the effect runs once after mount

return (
  <div>
    {data ? (
      <p>Data: {data}</p>
    ) : (
      <p>Loading data...</p>
    )}
  </div>
);
};

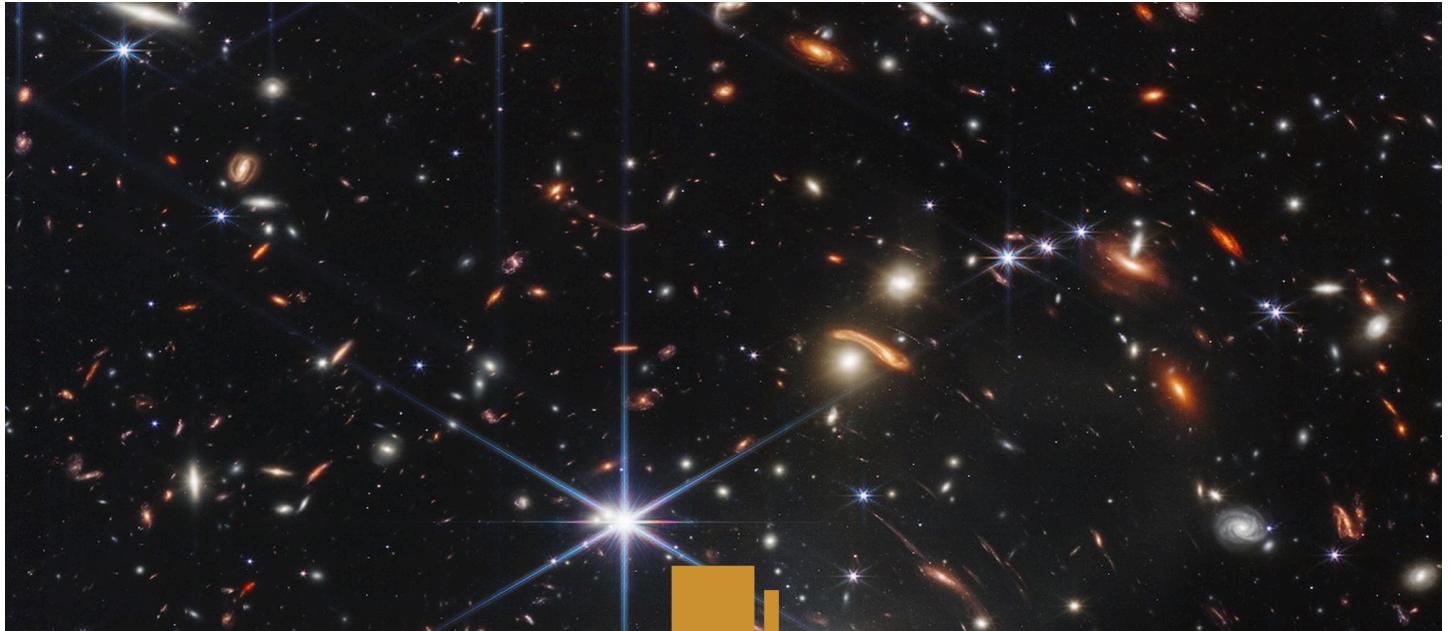
export default DataFetcher;
```

In this example:

1. We import `useState` and `useEffect` from 'react'.
2. Inside the `DataFetcher` component, we use `useState` to manage the state of the `data` variable.
3. The `useEffect` hook is employed to perform the data fetching operation when the component is mounted. The empty dependency array `[]` ensures that the effect runs only once after the initial render.

4. The `fetchData` function, declared inside the effect, simulates an asynchronous data fetching operation. Upon success, it updates the `data` state.
5. The component returns content based on whether the data has been fetched.

`useEffect` is a powerful tool for managing side effects in React components, providing a clean way to handle asynchronous operations and component lifecycle events.



Week 6.2

useEffect, useMemo, useCallback

In this lecture, Harkirat explores key aspects of React, starting with React Hooks like **useEffect**, **useMemo**, **useCallback**, and more, providing practical insights into state management and component functionalities. The discussions extend to creating custom hooks for reusable logic. Prop drilling, a common challenge in passing data between components, is addressed, offering effective solutions. The lecture also covers the Context API, a powerful tool for simplified state management across an entire React application.

[useEffect, useMemo, useCallback](#)

[Side Effects in React](#)

[React Hooks](#)

[1. useState\(\)](#)

[2. useEffect\(\)](#)

[Problem Statement](#)

[3. useMemo\(\)](#)

[4. useCallback\(\)](#)

[Difference between useEffect, useMemo & useCallback](#)

[Significance of Returning a Component from useEffect](#)

[Understanding the Code](#)

[Notes Under Progress —will be updated soon!](#)

Side Effects in React

In the context of React, **side effects** refer to operations or behaviors that occur outside the scope of the typical component rendering process. These can include data fetching, subscriptions, manual DOM manipulations, and other actions that have an impact beyond rendering the user interface.

Thus, "side effects" are the operations outside the usual rendering process, and "hooks," like **useEffect**, are mechanisms provided by React to handle these side effects in functional components. The **useEffect** hook allows you to incorporate side effects into your components in a clean and organized manner.

React Hooks

React Hooks are functions that allow functional components in React to have state and lifecycle features that were previously available only in class components. Hooks were introduced in React 16.8 to enable developers to use state and other React features without writing a class.

Using these hooks, developers can manage state, handle side effects, optimize performance, and create more reusable and readable functional components in React applications. Each hook serves a specific purpose, contributing to a more modular and maintainable codebase.

Some commonly used React Hooks are:

1. useState()

useState is a React Hook that enables functional components to manage state. It returns an array with two elements: the current state value and a function to update that value.

Here's an example of how to use **useState**:

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

const Counter = () => {
  // Using useState to initialize count state with an initial value of 0
  const [count, setCount] = useState(0);

  // Event handler to increment count
```

```
const increment = () => {
  // Using the setCount function to update the count state
  setCount(count + 1);
};

return (
  <div>
    <p>Count: {count}</p>
    <button onClick={increment}>Increment</button>
  </div>
);
};

export default Counter;
```

In this example:

1. We import the `useState` function from the 'react' package.
2. Inside the `Counter` component, we use `useState(0)` to initialize the state variable `count` with an initial value of 0.
3. The `count` state and the `setCount` function are destructured from the array returned by `useState`.
4. The `increment` function updates the `count` state by calling `setCount(count + 1)` when the button is clicked.
5. The current value of the `count` state is displayed within a paragraph element.

The above example helps us understand how `useState` helps manage and update state in functional components, providing a straightforward way to incorporate stateful behavior into React applications.

2. useEffect()

`useEffect` is a React Hook used for performing side effects in functional components. It is often used for tasks such as data fetching, subscriptions, or manually changing the DOM. The `useEffect` hook accepts two arguments: a function that contains the code to execute, and an optional array of dependencies that determines when the effect should run.

Here's an example of how to use `useEffect`:

```

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

const DataFetcher = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    // Effect will run after the component is mounted
    const fetchData = async () => {
      try {
        // Simulating a data fetching operation
        const response = await fetch('https://api.example.com/data');
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };
    fetchData();
  });

  // Effect cleanup (will run before unmounting)
  return () => {
    console.log('Component will unmount. Cleanup here.');
  };
}, []); // Empty dependency array means the effect runs once after mount

return (
  <div>
    {data ? (
      <p>Data: {data}</p>
    ) : (
      <p>Loading data...</p>
    )}
  </div>
);
};

export default DataFetcher;

```

In this example:

1. We import `useState` and `useEffect` from 'react'.
2. Inside the `DataFetcher` component, we use `useState` to manage the state of the `data` variable.
3. The `useEffect` hook is employed to perform the data fetching operation when the component is mounted. The empty dependency array `[]` ensures that the effect runs only once after the initial render.

4. The `fetchData` function, declared inside the effect, simulates an asynchronous data fetching operation. Upon success, it updates the `data` state.
5. The component returns content based on whether the data has been fetched.

`useEffect` is a powerful tool for managing side effects in React components, providing a clean way to handle asynchronous operations and component lifecycle events.

Problem Statement

3. useMemo()

`useMemo` is a React Hook that is used to memoize the result of a computation, preventing unnecessary recalculations when the component re-renders. It takes a function (referred to as the "memoized function") and an array of dependencies. The memoized function will only be recomputed when the values in the dependencies array change.

Here's an example of how to use `useMemo` :

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

const ExpensiveCalculation = ({ value }) => {
  const expensiveResult = useMemo(() => {
    // Simulating a computationally expensive operation
    console.log('Calculating expensive result...');
    return value * 2;
  }, [value]); // Dependency array: recalculates when 'value' changes

  return (
    <div>
      <p>Value: {value}</p>
      <p>Expensive Result: {expensiveResult}</p>
    </div>
  );
};

const MemoExample = () => {
  const [inputValue, setInputValue] = useState(5);

  return (
    <div>
      <input type="text" value={inputValue} onChange={e => setInputValue(e.target.value)} />
      <p>Current Value: {inputValue}</p>
    </div>
  );
};
```

```

<div>
  <input
    type="number"
    value={inputValue}
    onChange={(e) => setInputValue(Number(e.target.value))}>
  />
  <ExpensiveCalculation value={inputValue} />
</div>
);
};

export default MemoExample;

```

In this example:

1. We import `useState` and `useMemo` from 'react'.
2. The `ExpensiveCalculation` component takes a prop `value` and uses `useMemo` to calculate an "expensive" result based on that value.
3. The dependency array `[value]` indicates that the memoized function should be recomputed whenever the `value` prop changes.
4. The `MemoExample` component renders an `input` element and the `ExpensiveCalculation` component. The `value` prop of `ExpensiveCalculation` is set to the current state of `inputValue`.
5. As you type in the input, you'll notice that the expensive result is only recalculated when the input value changes, thanks to `useMemo`.

`useMemo` is particularly useful when dealing with expensive calculations or when you want to optimize performance by avoiding unnecessary computations during renders. It's important to use it judiciously, as overusing memoization can lead to increased complexity.

4. useCallback()

`useCallback` is a React Hook that is used to memoize a callback function, preventing unnecessary re-creation of the callback on each render. This can be useful when passing callbacks to child components to ensure they don't trigger unnecessary renders.

Here's an example of how to use `useCallback`:

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

```
const ChildComponent = ({ onClick }) => {
  console.log('ChildComponent rendering...');
  return <button onClick={onClick}>Click me</button>;
};

const CallbackExample = () => {
  const [count, setCount] = useState(0);

  // Regular callback function
  const handleClick = () => {
    console.log('Button clicked!');
    setCount((prevCount) => prevCount + 1);
  };

  // Memoized callback using useCallback
  const memoizedHandleClick = useCallback(handleClick, []);

  return (
    <div>
      <p>Count: {count}</p>
      <ChildComponent onClick={memoizedHandleClick} />
    </div>
  );
};

export default CallbackExample;
```

In this example:

1. We import `useState` and `useCallback` from 'react'.
2. The `ChildComponent` receives a prop `onClick` and renders a button with that click handler.
3. The `CallbackExample` component maintains a `count` state and has two callback functions: `handleClick` and `memoizedHandleClick`.
4. `handleClick` is a regular callback function that increments the count and logs a message.
5. `memoizedHandleClick` is created using `useCallback`, and its dependency array (`[]`) indicates that it should only be re-created if the component mounts or unmounts.
6. The `ChildComponent` receives the memoized callback (`memoizedHandleClick`) as a prop.
7. As you click the button in the `ChildComponent`, the count increases, and you'll notice that the log statement inside `handleClick` is only printed once, thanks to `useCallback` preventing unnecessary re-creations of the callback.

Using `useCallback` becomes more crucial when dealing with complex components or components with frequent re-renders, optimizing performance

by avoiding unnecessary function creations.

Difference between useEffect, useMemo & useCallback

1. useEffect :

- **Purpose:** Manages side effects in function components.
- **Triggers:** Runs after rendering and on subsequent re-renders.
- **Use Cases:** Fetching data, subscriptions, manually changing the DOM, etc.
- **Syntax:**

```
useEffect(() => {
  // Side effect logic here
  return () => {
    // Cleanup logic (optional)
  };
}, [dependencies]);
```

1. useMemo :

- **Purpose:** Memoizes the result of a computation to avoid unnecessary recalculations.
- **Triggers:** Runs during rendering.
- **Use Cases:** Memoizing expensive calculations, preventing unnecessary re-renders.
- **Syntax:**

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

1. useCallback :

- **Purpose:** Memoizes a callback function to prevent unnecessary re-renders of child components.
- **Triggers:** Runs during rendering.
- **Use Cases:** Preventing unnecessary re-renders when passing callbacks to child components.
- **Syntax:**

```
const memoizedCallback = useCallback(() => {
  // Callback logic here
}, [dependencies]);
```

In summary, `useEffect` is for handling side effects, `useMemo` is for memoizing values, and `useCallback` is for memoizing callback functions. Each serves a different purpose in optimizing and managing the behavior of React components.

Significance of Returning a Component from `useEffect`

In the provided code snippet, we utilize the `useEffect` hook along with the `setInterval` function to toggle the state of the `render` variable every 5 seconds. This, in turn, controls the rendering of the `MyComponent` or an empty `div` based on the value of `render`. Let's break down the significance of returning a component from `useEffect`:

```
import React, { useEffect, useState } from 'react';
import './App.css';

function App() {
  const [render, setRender] = useState(true);

  useEffect(() => {
    // Toggle the state every 5 seconds
    const intervalId = setInterval(() => {
      setRender(r => !r);
    }, 5000);

    // Cleanup function: Clear the interval when the component is unmounted
    return () => {
      clearInterval(intervalId);
    };
  }, []);

  return (
    <>
      {render ? <MyComponent /> : <div></div>}
    </>
  );
}
```

```
}
```

```
function MyComponent() {
  useEffect(() => {
    console.error("Component mounted");

    // Cleanup function: Log when the component is unmounted
    return () => {
      console.log("Component unmounted");
    };
  }, []);

  return <div>
    From inside MyComponent
  </div>;
}

export default App;
```

Understanding the Code

- The `useEffect` hook is used to create a side effect (in this case, toggling the `render` state at intervals) when the component mounts.
- A cleanup function is returned within the `useEffect`, which will be executed when the component is unmounted. In this example, it clears the interval previously set by `setInterval`.
- By toggling the `render` state, the component (`MyComponent` or an empty `div`) is conditionally rendered or unrendered, demonstrating the dynamic nature of component rendering.
- The `return` statement within the `useEffect` of `MyComponent` is used to specify what should be rendered when the component is active, in this case, a simple `div` with the text "From inside MyComponent."

In summary, the ability to return a cleanup function from `useEffect` is crucial for managing resources, subscriptions, or intervals created during the component's lifecycle. It helps ensure proper cleanup when the component is no longer in use, preventing memory leaks or unintended behavior.

Notes Under Progress —will be updated soon!



Week 7.1

Context API & Prop Drilling

In this lecture, Harkirat covers key concepts in React development, specifically focusing on routing, prop drilling, and the Context API. [Routing](#) is vital for managing navigation in React applications, while [prop drilling](#) and the [Context API](#) address challenges related to passing data between components. These insights provide essential knowledge for building well-organized and effective React projects.

[Context API & Prop Drilling](#)

[React Routing](#)

[Some Jargons](#)

[1. SPA \(Single Page Application\):](#)

[2. Client-side Bundle:](#)

[3. Client-side Routing:](#)

[React Router DOM](#)

[1. BrowserRouter:](#)

[2. Routes:](#)

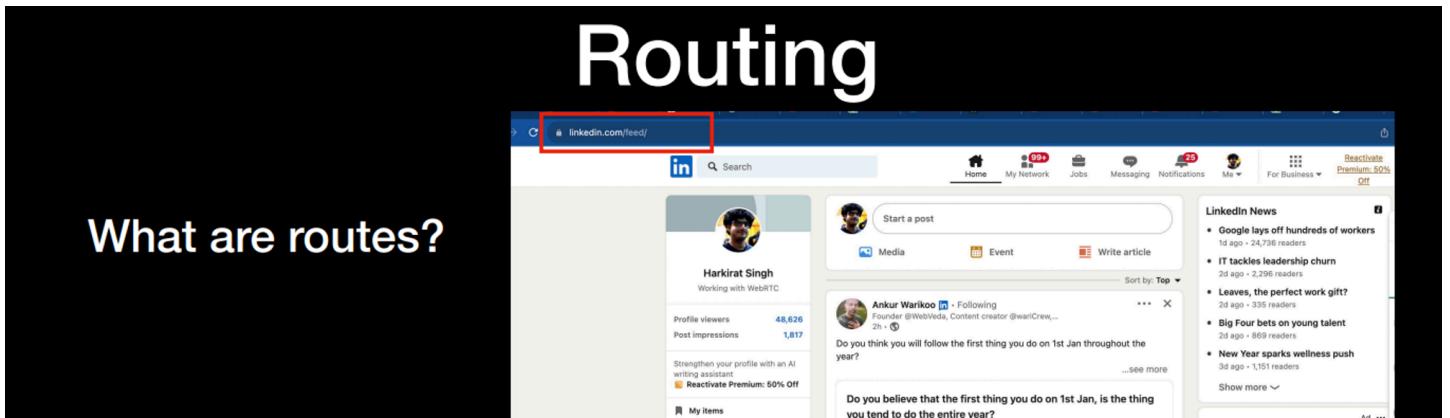
[3. Route:](#)

[Using React Router DOM](#)

[Issue with window.location.href](#)

[Note](#)

[Lazy Loading](#)

[React Suspense](#)[Code Implementation](#)[Prop Drilling in React](#)[Why Prop Drilling?](#)[Code Implementation](#)[Drawbacks](#)[Context API in React](#)[Key Components of Context API:](#)[Code Implementation](#)[Advantages of Context API:](#)[Other Solutions](#)[1. Context API](#)[2. Recoil](#)[3. Redux:](#)[Considerations:](#)[Choosing Between Them:](#)

React Routing

Routing in React is a mechanism that allows you to manage navigation and control the content displayed in your application based on the URL. It's essential for several reasons:

- 1. Multi-Page Applications (MPAs)** : In traditional web development, navigating between pages required a full page reload. React, being a Single Page Application (SPA) library, loads a single HTML page and dynamically updates the content as users navigate. Routing enables SPAs to mimic the behavior of traditional MPAs by updating the view based on the URL.

2. **User Experience** : Routing enhances the overall user experience by providing a seamless and dynamic interface. Users can navigate between different views or sections of your application without experiencing the delays associated with full-page reloads.
3. **Bookmarking and Sharing** : With routing, each view in your React application can have a unique URL. This allows users to bookmark specific pages or share URLs with others, making the application more user-friendly and SEO-friendly.
4. **Code Organization** : As your application grows, organizing code into separate components and views becomes crucial. Routing helps structure your code by associating components with specific routes, making it easier to manage and maintain.
5. **State Preservation** : When users navigate between different views, routing helps preserve the state of the application. React Router, a popular routing library for React, allows you to pass parameters and state between different components based on the route.
6. **Conditional Rendering** : Routing enables conditional rendering of components based on the current URL. Different components or views can be displayed depending on the route, allowing you to create dynamic and context-aware user interfaces.

To implement routing in a React application, developers often use libraries like React Router. React Router provides a set of components and functions to define routes, handle navigation, and manage the application's history, making it an essential tool for building robust and navigable React applications.

Some Jargons

1. SPA (Single Page Application):

A Single Page Application (SPA) is a type of web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server.

- **Key Characteristics:**
 - Loads a single HTML page initially.
 - Subsequent interactions and navigation are handled by dynamically updating the content on the page through JavaScript.
 - Utilizes AJAX or Fetch API to communicate with the server and fetch data without reloading the entire page.

- Provides a more fluid and seamless user experience by avoiding full-page reloads.

2. Client-side Bundle:

In the context of web development, a client-side bundle refers to a collection of JavaScript files and other assets bundled together to be delivered to the client's web browser.

- **Key Components:**

- **JavaScript Files:** The application's logic and functionality are written in JavaScript files. Bundling involves combining these files into a single or multiple bundles.
- **Stylesheets, Images, and Other Assets:** Along with JavaScript, other assets like stylesheets, images, and fonts may be included in the bundle for efficient delivery to the client.

- **Advantages:**

- Reduces the number of HTTP requests, improving loading times.
- Enables code splitting and lazy loading for optimizing performance.
- Simplifies deployment and maintenance by organizing code into manageable bundles.

3. Client-side Routing:

Client-side routing refers to the process of managing navigation within a Single Page Application (SPA) entirely on the client side, without making additional requests to the server for each new view.

- **Key Features:**

- Utilizes the browser's History API to manipulate the URL without triggering full page reloads.
- Enables dynamic content updates based on the route, improving user experience.
- Typically implemented using libraries like React Router for React applications or Vue Router for Vue.js applications.

- **Advantages:**

- Enhances the performance of SPAs by avoiding the need for server round-trips during navigation.
- Allows for a smoother and more responsive user interface as content is updated dynamically.
- Enables bookmarking, sharing, and direct linking to specific views within the SPA.

React Router DOM

In React, routing is commonly achieved using the React Router DOM library, which provides a set of components for handling navigation within a React application. The main components involved in React Router DOM are `BrowserRouter`, `Routes`, and `Route`. Here's an overview of how routing is typically implemented using these components:

1. BrowserRouter:

- The `BrowserRouter` component is a top-level component that should be used to wrap your entire application. It enables the use of routing features throughout your React application.
- It utilizes the HTML5 History API to manipulate the URL without triggering full page reloads.

```
import { BrowserRouter } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      {/* Other components and routing components go here */}
    </BrowserRouter>
  );
}
```

2. Routes:

- The `Routes` component is used to define the routes for your application. Inside the `Routes` component, you specify individual `Route` components for each route in your application.
- The `Routes` component can contain multiple `Route` components, each representing a different view or page.

```
import { Routes } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        {/* Define Route components here */}
      </Routes>
    </BrowserRouter>
  );
}
```

```
</BrowserRouter>
);
}
```

3. Route:

- The `Route` component is responsible for rendering specific components based on the current URL path. It takes two main props: `path` and `element`.
- The `path` prop defines the URL path that should match for the route to be rendered, and the `element` prop specifies the component to render when the path matches.

```
import { Route } from 'react-router-dom';
import Home from './components/Home';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        {/* Add more Route components for other views */}
      </Routes>
    </BrowserRouter>
  );
}

}
```

In the above example, when the URL path is "/", the `Home` component will be rendered.

This is a basic setup for using React Router DOM. You can extend this by adding nested routes, handling dynamic route parameters, and incorporating additional features provided by React Router DOM for more advanced routing scenarios.

Using React Router DOM

In a React app, creating a basic landing page and dashboard page with routing involves using React Router DOM to manage navigation. Here's a simple example:

1. Install React Router DOM:

If you haven't installed React Router DOM, you can do so by running the following command:

```
npm install react-router-dom
```

1. Setting up Routes:

Create two components for the landing page and the dashboard.

```
// Landing.jsx
import React from 'react';

const Landing = () => {
  return (
    <div>
      <h1>Welcome to the Landing Page</h1>
    </div>
  );
};

export default Landing;
```

```
// Dashboard.jsx
import React from 'react';

const Dashboard = () => {
  return (
    <div>
      <h1>Dashboard</h1>
    </div>
  );
};

export default Dashboard;
```

1. Create the Main App Component:

Set up your main App component with React Router to handle routing.

```
// App.jsx
import React from 'react';
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import Landing from './Landing';
import Dashboard from './Dashboard';

const App = () => {
```

```

    return (
      <BrowserRouter>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/dashboard">Dashboard</Link>
            </li>
          </ul>
        </nav>

        <Routes>
          <Route path="/" element={<Landing />} />
          <Route path="/dashboard" element={<Dashboard />} />
        </Routes>
      </BrowserRouter>
    );
  };

export default App;

```

In this example, we use the `Link` component from React Router to create navigation links. The `Routes` component contains individual `Route` components for each page.

1. Navigate Programmatically:

If you want to navigate programmatically, you can use `window.location.href`. For example, in a function:

```

const navigateToDashboard = () => {
  window.location.href = '/dashboard';
};

```

However, using `Link` from React Router is a preferred way for declarative navigation within a React app.

1. Shared UI:

If you want to share UI components between the landing page and the dashboard, you can create a common component and use it in both `Landing` and `Dashboard` components.

```

// SharedComponent.jsx
import React from 'react';

const SharedComponent = () => {
  return (

```

```

<div>
  <p>This component is shared between Landing and Dashboard.</p>
</div>
);
;

export default SharedComponent;

```

Import and use `SharedComponent` in both `Landing` and `Dashboard`.

Via this example, we learn about a basic structure for setting up routing in a React app. React Router's declarative approach makes it easy to manage navigation and share UI components between different pages.

Issue with `window.location.href`

When using `window.location.href` for navigation in a React application, it triggers a full page reload, which is not desirable in client-side routing. A full page reload involves fetching the HTML, CSS, and other assets again, leading to a slower and less efficient user experience.

To address this issue, React Router DOM provides a solution in the form of the `useNavigate` hook. This hook is designed for programmatic navigation within a React component without triggering a full page reload. By using `useNavigate`, you can ensure smoother transitions between different views in a single-page application (SPA) without unnecessary overhead.

Here's an example of how to use `useNavigate`:

```

// App.jsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link, useNavigate } from 'react-router-dom';
import Landing from './Landing';
import Dashboard from './Dashboard';

const App = () => {
  const navigate = useNavigate();

  const navigateToDashboard = () => {
    // Use navigate function instead of window.location.href
    navigate('/dashboard');
  };
}

```

```
return (
  <Router>
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          {/* Use the navigateToDashboard function for navigation */}
          <button onClick={navigateToDashboard}>Go to Dashboard</button>
        </li>
      </ul>
    </nav>

    <Routes>
      <Route path="/" element={<Landing />} />
      <Route path="/dashboard" element={<Dashboard />} />
    </Routes>
  </Router>
);
};

export default App;
```

In this example, the `useNavigate` hook is used to get the `navigate` function, which can be called to navigate to different routes without causing a full page reload. By using this approach, you maintain the benefits of client-side routing in React, ensuring a faster and more seamless user experience.

Note

← →

`BrowserRouter`. It should be used inside a component that is a descendant of `BrowserRouter` to ensure access to the correct router context. This limitation is intentional, as `useNavigate` relies on the router context for scoped navigation, enabling seamless client-side routing without triggering a full page reload. Placing the hook within the correct context ensures its proper functionality for dynamic view and URL updates.

Lazy Loading

Lazy loading in React is a technique used to optimize the performance of a web application by deferring the loading of certain components until they are actually needed. This can significantly reduce the initial bundle size and improve the overall loading time of the application.

In React, lazy loading is typically achieved using the `React.lazy` function along with the `Suspense` component. The `React.lazy` function allows you to load a component lazily, meaning it is only fetched when the component is actually rendered. Here's a simple example:

React Suspense

In React, `Suspense` is a component that enables a better experience for handling asynchronous operations such as code-splitting and lazy loading. It's used in conjunction with `React.lazy` for lazy loading components or with data fetching functions.

When you're using `React.lazy` to load a component lazily, you wrap it with `Suspense` to specify a fallback UI that will be rendered while the component is being loaded. The `fallback` prop of `Suspense` defines what to display during the loading period.

Code Implementation

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

const MyLazyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <MyLazyComponent />
      </Suspense>
    </div>
  );
}
```

In this example, if `MyLazyComponent` is not yet loaded, the `Suspense` component will render the "Loading..." message as the fallback until the component is fully loaded and ready to be displayed.

This mechanism is particularly useful for improving the user experience when dealing with dynamic loading of components or fetching data asynchronously. The `fallback` UI gives users feedback that something is happening in the background, making the application feel more responsive.

Prop Drilling in React

Prop drilling refers to the process of passing data from a top-level component down to deeper levels through intermediate components. It happens when a piece of state needs to be accessible by a component deep in the component tree, and it gets passed down as a prop through all the intermediate components.

Why Prop Drilling?

1. State Management:

Prop drilling is often used to manage state in a React application. By passing state down through the component tree, you can share data between components without resorting to more advanced state management solutions like context or state management libraries.

2. Simplicity:

Prop drilling keeps the application structure simple and makes it easier to understand the flow of data. It's a straightforward way of handling data without introducing more complex tools.

Code Implementation

```
// Top-level component
function App() {
  const data = "Hello from App component";

  return <ChildComponent data={data} />;
}

// Intermediate component
function ChildComponent({ data }) {
  return <GrandchildComponent data={data} />;
}

// Deepest component
function GrandchildComponent({ data }) {
```

```
return <p>{data}</p>;  
}
```

In this example, `App` has a piece of data that needs to be accessed by `GrandchildComponent`. Instead of using more advanced state management tools, we pass the `data` as a prop through `ChildComponent`. This is prop drilling in action.

Drawbacks

1. Readability:

Prop drilling can make the code less readable, especially when you have many levels of components. It might be hard to trace where a particular prop is coming from.

2. Maintenance:

If the structure of the component tree changes, and the prop needs to be passed through additional components, it requires modifications in multiple places.

While prop drilling is a simple and effective way to manage state in some cases, for larger applications or more complex state management, consider using tools like React Context or state management libraries. These can help avoid the drawbacks of prop drilling while providing a cleaner solution for state sharing.

Context API in React

Context API is a feature in React that provides a way to share values like props between components without explicitly passing them through each level of the component tree. It helps solve the prop drilling problem by allowing data to be accessed by components at any level without the need to pass it through intermediate components.

Key Components of Context API:

1. `createContext` :

The `createContext` function is used to create a context. It returns an object with two components - `Provider` and `Consumer`.

```
const MyContext = React.createContext();
```

1. Provider :

The `Provider` component is responsible for providing the context value to its descendants. It is placed at the top of the component tree.

```
<MyContext.Provider value{/* some value */}>
  {/* Components that can access the context value */}
</MyContext.Provider>
```

1. Consumer (or useContext hook):

The `Consumer` component allows components to consume the context value. Alternatively, the `useContext` hook can be used for a more concise syntax.

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

or

```
const value = useContext(MyContext);
```

Code Implementation

```
// Create a context
const UserContext = React.createContext();

// Top-level component with a Provider
function App() {
  const user = { username: "john_doe", role: "user" };

  return (
    <UserContext.Provider value={user}>
      <Profile />
    </UserContext.Provider>
  );
}

// Intermediate component
function Profile() {
  return <Navbar />;
}
```

```
}
```

```
// Deepest component consuming the context value
function Navbar() {
  const user = useContext(UserContext);

  return (
    <nav>
      <p>Welcome, {user.username} ({user.role})</p>
    </nav>
  );
}
```

In this example, the `UserContext.Provider` in the `App` component provides the `user` object to all its descendants. The `Navbar` component, which is deeply nested, consumes the `user` context value without the need for prop drilling.

Advantages of Context API:

1. Avoids Prop Drilling:

Context API eliminates the need for passing props through intermediate components, making the code cleaner and more maintainable.

2. Global State:

It allows you to manage global state that can be accessed by components across the application.

While Context API is a powerful tool, it's essential to use it judiciously and consider factors like the size and complexity of the application. For complex state management needs, additional tools like Redux might be more suitable.

Other Solutions

Recoil, Redux, and Context API are all solutions for managing state in React applications, each offering different features and trade-offs.

1. Context API

- **Role:** Context API is a feature provided by React that allows components to share state without prop drilling. It creates a context and a provider to wrap components that need access to that context.
- **Usage:**

```
// Context creation
import { createContext, useContext } from 'react';

const UserContext = createContext();

// Context provider
function UserProvider({ children }) {
  const user = { name: 'John' };

  return <UserContext.Provider value={user}>{children}</UserContext.Provider>;
}

// Accessing context in a component
function Profile() {
  const user = useContext(UserContext);

  return <p>Welcome, {user.name}</p>;
}
```

- **Advantages:** Simplicity, built-in React feature.

2. Recoil

- **Role:** Recoil is a state management library developed by Facebook for React applications. It introduces the concept of atoms and selectors to manage state globally. It can be considered a more advanced and feature-rich alternative to Context API.
- **Usage:**

```
// Atom creation
import { atom, useRecoilState } from 'recoil';

export const userState = atom({
  key: 'userState',
  default: { name: 'John' },
});

// Accessing Recoil state in a component
function Profile() {
  const [user, setUser] = useRecoilState(userState);
```

```

    return (
      <div>
        <p>Welcome, {user.name}</p>
        <button onClick={() => setUser({ name: 'Jane' })}>Change Name</button>
      </div>
    );
}

```

- **Advantages:** Advanced features like selectors, better performance optimizations.

3. Redux:

- **Role:** Redux is a powerful state management library often used with React. It introduces a global store and follows a unidirectional data flow. While Redux provides more features than Context API, it comes with additional concepts and boilerplate.
- **Usage:**

```

// Store creation
import { createStore } from 'redux';

const initialState = { user: { name: 'John' } };

const rootReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'CHANGE_NAME':
      return { ...state, user: { name: 'Jane' } };
    default:
      return state;
  }
};

const store = createStore(rootReducer);

// Accessing Redux state in a component
function Profile() {
  const user = useSelector((state) => state.user);
  const dispatch = useDispatch();

  return (
    <div>
      <p>Welcome, {user.name}</p>
      <button onClick={() => dispatch({ type: 'CHANGE_NAME' })}>Change Name</button>
    </div>
  );
}

```

- **Advantages:** Middleware support, time-travel debugging, broader ecosystem.

Considerations:

- **Complexity:** Context API is simple and built into React, making it a good choice for simpler state management. Recoil provides more features and optimizations, while Redux is powerful but comes with additional complexity.
- **Scalability:** Recoil and Redux are often preferred for larger applications due to their ability to manage complex state logic.
- **Community Support:** Redux has a large and established community with a wide range of middleware and tools. Recoil is newer but gaining popularity, while Context API is part of the React core.

Choosing Between Them:

- **Use Context API for Simplicity:** For simpler state management needs, especially in smaller applications or when simplicity is a priority.
- **Consider Recoil for Advanced Features:** When advanced state management features, like selectors and performance optimizations, are needed.
- **Opt for Redux for Scalability:** In larger applications where scalability, middleware, and a broader ecosystem are important factors.



Week 7.2

Context API & Recoil

In this lecture, Harkirat covers the drawbacks of the Context API for state management and introduces Recoil as an alternative solution. The discussion focuses on Recoil's core elements, including `RecoilRoot`, `atoms`, `selectors`, and Recoil hooks. Through practical code examples, Harkirat demonstrates how Recoil simplifies state management in React, offering a robust and efficient approach.

[Context API & Recoil](#)

[Statement Management](#)

[Problem with Context API](#)

[Recoil](#)

[Concepts in Recoil](#)

[1\] RecoilRoot](#)

[2\] atom](#)

[Recoil Hooks](#)

[1\] useRecoilState:](#)

[2\] useRecoilValue:](#)

[3\] useSetRecoilState:](#)

[Selectors](#)

[1\] Creating a Selector:](#)

[2\] Using Selectors in Components:](#)

3] Atom and Selector Composition:

Recoil Code Implementation

Statement Management

State management refers to the process of handling and maintaining the state or data of an application throughout its lifecycle. In frontend development, state typically represents the current condition or values of variables in an application. Effective state management is crucial for building dynamic and interactive user interfaces.

In React and other frontend frameworks, there are various methods to manage state:

1. Local Component State:

- Each component in React can have its own local state managed using the `useState` hook.
- Local state is confined to the component it belongs to and is primarily used for managing component-specific data.

2. Context API:

- React provides the Context API to manage global state that needs to be accessed by multiple components.
- It allows the sharing of state across the component tree without having to pass props manually through each level.

3. State Management Libraries (e.g., Redux, Recoil):

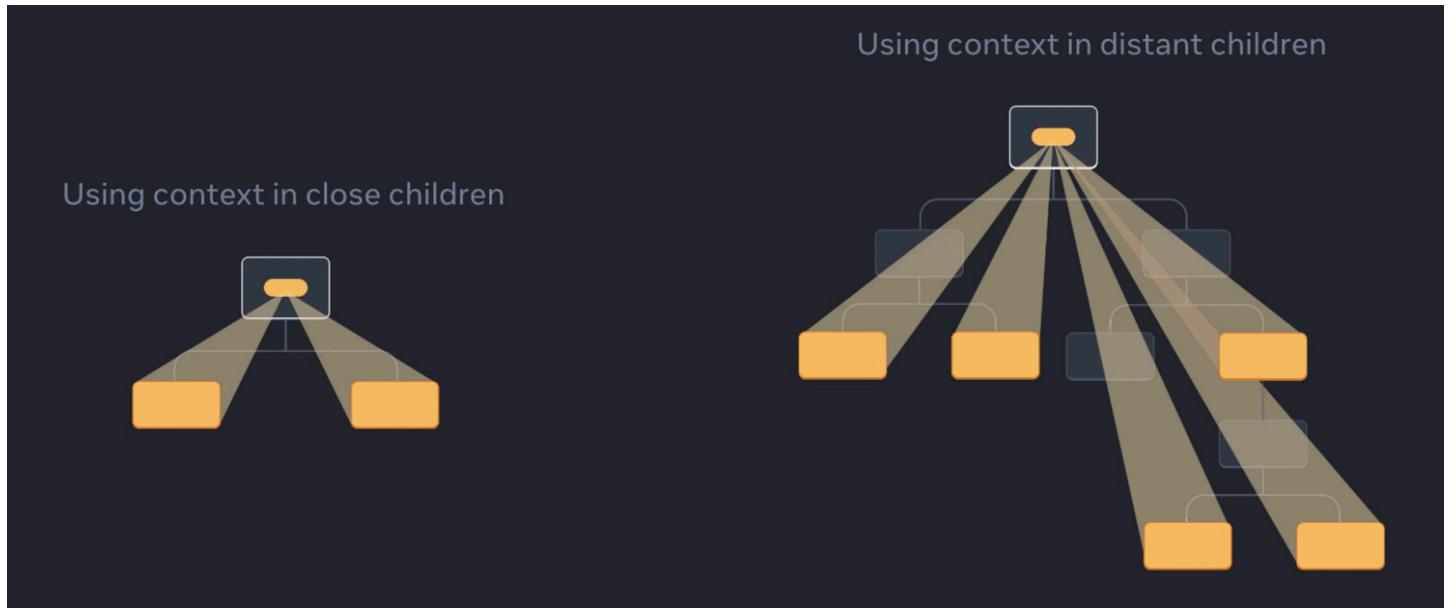
- Specialized state management libraries offer advanced features for handling complex global state in large applications.
- These libraries often introduce concepts like actions, reducers, and a centralized store for maintaining state.

4. Recoil:

- Recoil is a state management library developed by Facebook specifically for React applications.
- It introduces the concept of atoms and selectors, providing a more flexible and scalable approach to managing and sharing state.

The choice of state management method depends on the complexity and requirements of the application. Effective state management enhances the

predictability, maintainability, and scalability of the application, ensuring a smooth and responsive user experience.



Problem with Context API

Context API in React is a powerful tool for solving the prop drilling problem by allowing the passing of data through the component tree without the need for explicit props at every level. However, it does not inherently address the re-rendering issue.

When using the Context API, updates to the context can trigger re-renders of all components that consume that context, even if the specific data they need hasn't changed. This can potentially lead to unnecessary re-renders and impact the performance of the application.

To mitigate this, developers can use techniques such as memoization (with `useMemo` or `React.memo`) to prevent unnecessary re-renders of components that don't depend on the changes in context. Additionally, libraries like Redux, Recoil, or Zustand provide more fine-grained control over state updates and re-renders compared to the built-in Context API.

This leads us to Recoil, a state management library designed explicitly for React applications.

Recoil

Recoil, developed by Facebook, is a state management library for React applications. It introduces a more sophisticated approach to handling state, offering features like atoms, selectors, and a global state tree. With Recoil, we can overcome some of the challenges associated with prop drilling and achieve a more scalable and organized state management solution. As we make this transition, we'll explore Recoil's unique features and understand how it enhances the efficiency and maintainability of our React applications.

Concepts in Recoil

1] RecoilRoot

The `RecoilRoot` is a component provided by Recoil that serves as the root of the Recoil state tree. It must be placed at the top level of your React component tree to enable the use of Recoil atoms and selectors throughout your application.

Here's a simple code snippet demonstrating the usage of `RecoilRoot`:

```
import React from 'react';
import { RecoilRoot } from 'recoil';
import App from './App';

const RootComponent = () => {
  return (
    <RecoilRoot>
      <App />
    </RecoilRoot>
  );
};

export default RootComponent;
```

In this example, `RecoilRoot` wraps the main `App` component, providing the context needed for Recoil to manage the state. By placing it at the top level, you ensure that all components within the `App` have access to Recoil's global state. This structure allows you to define and use Recoil atoms and selectors across different parts of your application.

2] atom

In Recoil, an atom is a unit of state. It represents a piece of state that can be read from and written to by various components in your React application. Atoms act as shared pieces of state that can be

used across different parts of your component tree.

Here's a simple example of defining an atom:

```
import { atom } from 'recoil';

export const countState = atom({
  key: 'countState', // unique ID (with respect to other atoms/selectors)
  default: 0,        // default value (aka initial value)
});
```

In this example, `countState` is an atom that represents a simple counter. The `key` is a unique identifier for the atom, and the `default` property sets the initial value of the atom.

Once defined, you can use this atom in different components to read and update its value. Components that subscribe to the atom will automatically re-render when the atom's value changes, ensuring that your UI stays in sync with the state. This makes atoms a powerful and flexible tool for managing shared state in Recoil-based applications.

Recoil Hooks

In Recoil, the hooks `useRecoilState`, `useRecoilValue`, and `useSetRecoilState` are provided to interact with atoms and selectors.

1] `useRecoilState` :

- This hook returns a tuple containing the current value of the Recoil state and a function to set its new value.
- Example:

```
const [count, setCount] = useRecoilState(countState);
```

2] `useRecoilValue` :

- This hook retrieves and subscribes to the current value of a Recoil state.
- Example:

```
const count = useRecoilValue(countState);
```

3] useSetRecoilState :

- This hook returns a function that allows you to set the value of a Recoil state without subscribing to updates.
- Example:

```
const setCount = useSetRecoilState(countState);
```

These hooks provide a convenient way to work with Recoil states in functional components.

`useRecoilState` is used when you need both the current value and a setter function, `useRecoilValue` when you only need the current value, and `useSetRecoilState` when you want to set the state without subscribing to updates. They contribute to making Recoil-based state management more ergonomic and straightforward.

Selectors

In Recoil, selectors are functions that derive new pieces of state from existing ones. They allow you to compute derived state based on the values of atoms or other selectors. Selectors are an essential part of managing complex state logic in a Recoil application.

Here are some key concepts related to selectors:

1] Creating a Selector:

- You can create a selector using the `selector` function from Recoil.
- Example:

```
import { selector } from 'recoil';

const doubledCountSelector = selector({
  key: 'doubledCount',
```

```
get: ({ get }) => {
  const count = get(countState);
  return count * 2;
},
});
```

2] Using Selectors in Components:

- You can use selectors in your components using the `useRecoilValue` hook.
- Example:

```
import { useRecoilValue } from 'recoil';

const DoubledCountComponent = () => {
  const doubledCount = useRecoilValue(doubledCountSelector);

  return <div>Doubled Count: {doubledCount}</div>;
};
```

3] Atom and Selector Composition:

- Selectors can depend on atoms or other selectors, allowing you to compose more complex state logic.
- Example:

```
const totalSelector = selector({
  key: 'total',
  get: ({ get }) => {
    const count = get(countState);
    const doubledCount = get(doubledCountSelector);
    return count + doubledCount;
  },
});
```

Selectors provide a powerful way to manage derived state in a Recoil application, making it easy to compute and consume state values based on the current state of your atoms.

Recoil Code Implementation

To create a Recoil-powered React application with the described functionality, follow the steps below:

1. Install Recoil in your project:

```
npm install recoil
```

1. Set up your project structure:

Assuming a folder structure like this:

```
/src
  /components
    Counter.jsx
  /store/atoms
    countState.jsx
  App.jsx
```

1. Create `countState.js` in the `atoms` folder:

```
// store/atoms/countState.jsx
import { atom } from 'recoil';

export const countState = atom({
  key: 'countState',
  default: 0,
});
```

1. Create `Counter.js` in the `components` folder:

```
// components/Counter.jsx
import React from 'react';
import { useRecoilState, useRecoilValue } from 'recoil';
import { countState } from '../store/atoms/countState';

const Counter = () => {
  const [count, setCount] = useRecoilState(countState);
```

```
const handleIncrease = () => {
  setCount(count + 1);
};

const handleDecrease = () => {
  setCount(count - 1);
};

const isEven = useRecoilValue(countIsEven);

return (
  <div>
    <h1>Count: {count}</h1>
    <button onClick={handleIncrease}>Increase</button>
    <button onClick={handleDecrease}>Decrease</button>
    {isEven && <p>It is EVEN</p>}
  </div>
);
};

export default Counter;
```

1. Create App.js :

```
// App.jsx
import React from 'react';
import { RecoilRoot } from 'recoil';
import Counter from './components/Counter';

function App() {
  return (
    <RecoilRoot>
      <Counter />
    </RecoilRoot>
  );
}

export default App;
```

Make sure to adjust your project's entry point to use `App.js`.

Now, your Recoil-powered React application should render a counter with increase and decrease buttons. The message "It is EVEN" will be displayed when the count is an even number.



Week 8.1

Tailwind

In this lecture, Harkirat explores [Tailwind CSS](#), the go-to framework for frontend development. We've covered four crucial CSS concepts: Flexbox, Grid, responsiveness, and basic styling. To make things practical, Harkirat walks us through cloning a Dukaan Figma page using [React and Tailwind CSS](#), giving us a hands-on experience to reinforce our learning.

Tailwind

What is Tailwind CSS?

Essentials

1] Flexbox

2] Grid

3] Responsiveness

4] Background Color, Text Color, Hover

Putting it All Together:

Fundamentals in CSS & Tailwind

Flex in CSS:

Flex in Tailwind CSS:

Grid in CSS:

Grid in Tailwind CSS:

Responsiveness in CSS:

Responsiveness in Tailwind CSS:

Mobile First Approach

Why Mobile-First in Tailwind CSS?

Implementation in Tailwind CSS:

Other Styles in CSS:

Styles in Tailwind CSS:

Key Points:

What is Tailwind CSS?

Tailwind CSS is a utility-first CSS framework that provides a set of low-level utility classes to build designs directly in your markup. It follows a unique approach where styles are applied using classes in the HTML, eliminating the need for writing custom CSS. Unlike traditional CSS frameworks, Tailwind doesn't impose a predefined UI design, offering maximum flexibility.

Key Points:

- **Utility-First Approach:** Tailwind encourages a utility-first approach, where individual classes directly apply styles.
- **Configurability:** Tailwind is highly configurable, allowing developers to customize styles and add new utilities.
- **No Preprocessor:** Unlike other frameworks that use preprocessor languages like Sass or Less, Tailwind relies on vanilla CSS.

Essentials

Although CSS may seem very daunting and exhaustive to master. In reality, proficiency in a select few fundamental concepts is all you need to efficiently tackle a substantial portion of frontend development tasks in the practical world.

1] Flexbox

- **What it does:** Helps you easily arrange and organize elements on your webpage.
- **Example Use Cases:** Designing navigation bars, aligning items in a row or column, and centering content both vertically and horizontally.

- **Why it's Important:** It simplifies layout creation and makes your designs more flexible and responsive.

2] Grid

- **What it does:** Enables you to create structured and organized layouts with rows and columns.
- **Example Use Cases:** Designing complex layouts, aligning images and text neatly, and ensuring your design adjusts well to different screen sizes.
- **Why it's Important:** It gives you precise control over how your page elements are positioned and arranged.

3] Responsiveness

- **What it does:** Ensures your website looks good on all devices, from large desktop screens to small mobile phones.
- **Example Use Cases:** Using media queries to adjust layout and font sizes based on the device, creating designs that smoothly adapt to different screen sizes.
- **Why it's Important:** It provides a consistent and user-friendly experience regardless of the device being used.

4] Background Color, Text Color, Hover

- **What it does:** Adds visual appeal to your website by styling colors and creating interactive effects.
- **Example Use Cases:** Setting background colors, defining text colors for readability, and creating interactive hover effects for buttons.
- **Why it's Important:** It enhances the look and feel of your site, making it visually pleasing and engaging for users.

Putting it All Together:

- With Flexbox and Grid, you can structure your page the way you want.
- Responsiveness ensures your design looks good on any device.

- Background and text colors, along with hover effects, add the finishing touches to make your site visually appealing and interactive.

These fundamental concepts, when used together, allow you to create a variety of web designs efficiently. Whether you're building a simple webpage or a more complex application, mastering these basics provides a strong foundation for frontend development.

Let's take a look at all these fundamentals in detail, first in CSS and then followed in Tailwind CSS

Fundamentals in CSS & Tailwind

Flex in CSS:

Flexbox (Flexible Box Layout):

Flexbox is a layout model in CSS that allows you to design complex layouts more efficiently and with less code. It's especially useful for distributing space and aligning items within a container, even when their sizes are unknown or dynamic.

Key Concepts:

- **Flex Container:** The parent element with `display: flex` or `display: inline-flex` is known as the flex container.
- **Flex Items:** The child elements of a flex container are the flex items.
- **Main and Cross Axes:** Flexbox works along two axes - the main axis and the cross axis. The `flex-direction` property defines the main axis direction.

Example:

```
.container {  
  display: flex;  
  flex-direction: row; /* or column, column-reverse, row-reverse */  
  justify-content: space-between; /* or flex-start, flex-end, center, space-around, space-evenly */  
  align-items: center; /* or flex-start, flex-end, stretch, baseline */  
}
```

In this example, the `container` class becomes a flex container with its child elements as flex items. The `flex-direction` property sets the direction of the main axis, while `justify-content` and `align-items` control the alignment of items along the main and cross axes.

Flex in Tailwind CSS:

Tailwind CSS simplifies the use of Flexbox by providing utility classes that directly apply Flexbox properties to elements in your HTML.

Example:

```
<div class="flex justify-between items-center">
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
</div>
```

In this example, the `flex` class makes the container a flex container, `justify-between` distributes the items along the main axis with space between them, and `items-center` aligns the items along the cross axis in the center.

Key Points:

- **Responsive Classes:** Tailwind allows you to apply responsive classes for different screen sizes, making it easy to create layouts that adapt to various devices.
- **Utility-First Approach:** Rather than writing custom CSS, you use utility classes directly in your HTML, allowing for quick prototyping and easy-to-understand code.

While CSS Flexbox provides a more extensive and fine-grained control over layout, Tailwind CSS simplifies the process by offering utility classes that encapsulate common Flexbox patterns. It's a matter of choosing the approach that aligns with your project's needs and your preferred development style.

Grid in CSS:

CSS Grid is a two-dimensional layout system that enables you to create complex layouts with rows and columns. It's particularly useful for building grid-based designs, providing precise control over the placement and sizing of elements.

Key Concepts:

- **Grid Container:** The parent element with `display: grid` becomes a grid container.

- **Grid Items:** The children of the grid container are grid items.
- **Grid Template:** You can define rows and columns using properties like `grid-template-rows` and `grid-template-columns`.
- **Grid Areas:** Named areas within the grid can be defined, allowing items to span multiple rows and columns.

Example:

```
.container {
  display: grid;
  grid-template-rows: 100px auto 100px;
  grid-template-columns: 1fr 2fr 1fr;
  gap: 10px; /* Gap between grid items */
}
```

In this example, the `container` class becomes a grid container with specific rows and columns, providing a clear structure for layout.

Grid in Tailwind CSS:

Tailwind CSS Grid Utilities:

Tailwind CSS simplifies the use of CSS Grid by providing utility classes that directly apply grid-related properties to elements in your HTML.

Example:

```
<div class="grid grid-rows-3 grid-cols-3 gap-10">
  <div class="row-span-1 col-span-1">Item 1</div>
  <div class="row-span-3 col-span-2">Item 2</div>
  <div class="row-span-1 col-span-1">Item 3</div>
</div>
```

In this example, the `grid` class makes the container a grid container with three rows and three columns. The `gap-10` class sets a gap of 10 pixels between grid items. The `row-span-X` and `col-span-Y` classes define how many rows or columns an item should span.

Key Points:

- **Responsive Classes:** Tailwind allows you to use responsive classes for different screen sizes, adapting your grid layout accordingly.

- **Utility-First Approach:** Instead of writing custom CSS, you apply utility classes directly in your HTML, promoting a rapid and efficient development process.

CSS Grid provides a comprehensive and fine-tuned layout system, while Tailwind CSS simplifies the process by offering utility classes that encapsulate common grid patterns. The choice between the two depends on the project's requirements and your preferred development workflow. Tailwind CSS's utility-first approach can be advantageous for quick prototyping and development efficiency.

Responsiveness in CSS:

Media Queries:

Responsiveness in CSS is achieved through the use of media queries. Media queries allow you to apply different styles based on the characteristics of the device, such as its width, height, or screen orientation. This is crucial for ensuring that your website or application looks and functions well across a variety of devices and screen sizes.

Example:

```
@media screen and (max-width: 768px) {  
    /* Styles for screens with a maximum width of 768 pixels */  
    body {  
        font-size: 14px;  
    }  
}
```

In this example, when the screen width is 768 pixels or less, the font size of the body text is adjusted to make it more suitable for smaller screens.

Responsiveness in Tailwind CSS:

Responsive Classes:

Tailwind CSS simplifies the process of making your designs responsive by providing responsive utility classes. These classes are designed to apply styles based on screen size breakpoints.

Example:

```
<div class="text-lg md:text-xl lg:text-2xl xl:text-3xl">  
    Responsive Text
```

</div>

In this example, the `text-lg` class sets the text size to large by default. However, on medium (`md`), large (`lg`), and extra-large (`xl`) screens, the text size is increased to extra-large, `2xl`, and `3xl`, respectively.

Key Points:

- **Breakpoints:** Tailwind uses breakpoints like `sm`, `md`, `lg`, and `xl` to define different screen sizes.
- **Utility Classes:** You can apply utility classes directly in your HTML to change styles at different breakpoints.

While traditional CSS media queries provide extensive control and customization for responsiveness, Tailwind CSS simplifies the process with utility classes that are quick to apply. Tailwind's responsive classes are convenient for common scenarios and can significantly speed up the development process, particularly for smaller to medium-sized projects.

Mobile First Approach

In the context of Tailwind CSS, "mobile-first" refers to an approach where the design and styling of a website or application are primarily focused on smaller screens, such as those of mobile devices, before addressing larger screens. This approach aligns with the philosophy that mobile devices are commonly used for accessing the internet, and starting with a design that caters to smaller screens ensures a user-friendly and responsive experience across all devices.

Why Mobile-First in Tailwind CSS?

1. Default Styles:

- **Mobile-Friendly Defaults:** Tailwind CSS is designed with mobile-first principles in mind. By default, many of its utility classes are optimized for smaller screens, providing a clean and user-friendly appearance on mobile devices.

2. Responsive Classes:

- **Breakpoint Classes:** Tailwind provides responsive utility classes that can be used to adjust styles based on screen size breakpoints. By starting with mobile-friendly styles and

gradually enhancing them with responsive classes, you ensure a smooth transition across various devices.

3. Reduced Overhead:

- **Lighter Styles for Mobile:** A mobile-first approach in Tailwind often leads to more concise and efficient styles for smaller screens. This can result in faster loading times and better performance, especially on mobile devices with limited resources.

4. Streamlined Development:

- **Efficient Prototyping:** The utility-first approach of Tailwind allows for quick prototyping by applying styles directly in the HTML. This facilitates a rapid development process, and starting with mobile styles enables developers to iterate and experiment effectively.

Implementation in Tailwind CSS:

When implementing a mobile-first design in Tailwind CSS, you start with styles that are optimized for smaller screens and then use Tailwind's responsive utility classes to enhance the design for larger screens. Here's a simplified example:

```
<div class="bg-blue-500 text-white p-4">
  <!-- Mobile-friendly styling -->
  <p class="text-lg">This is a mobile-friendly text.</p>

  <!-- Responsive enhancement for larger screens -->
  <p class="lg:text-xl">This text grows larger on larger screens.</p>
</div>
```

In this example, the `text-lg` class sets the text size to large by default, which is suitable for mobile screens. The `lg:text-xl` class adjusts the text size to extra-large for screens that meet the large (`lg`) breakpoint.

By embracing a mobile-first approach with Tailwind CSS, developers can create responsive and efficient designs that cater to the diverse range of devices users might utilize to access their websites or applications. It not only enhances user experience but also contributes to a more optimized and performant web presence.

Other Styles in CSS:

1. Background Color:

- CSS:

```
.container {  
    background-color: #3498db;  
}
```

2. Text Color:

- CSS:

```
.text-example {  
    color: #2ecc71;  
}
```

3. Hover Effects:

- CSS:

```
.button {  
    background-color: #e74c3c;  
}  
.button:hover {  
    background-color: #c0392b;  
}
```

Styles in Tailwind CSS:

1. Background Color:

- Tailwind CSS:

```
<div class="bg-blue-500">  
    <!-- Content -->  
</div>
```

2. Text Color:

- Tailwind CSS:

```
<p class="text-green-600">  
  This text has a green color.  
</p>
```

3. Hover Effects:

- Tailwind CSS:

```
<button class="bg-red-500 hover:bg-red-700">  
  Click me  
</button>
```

Key Points:

1. Background Color:

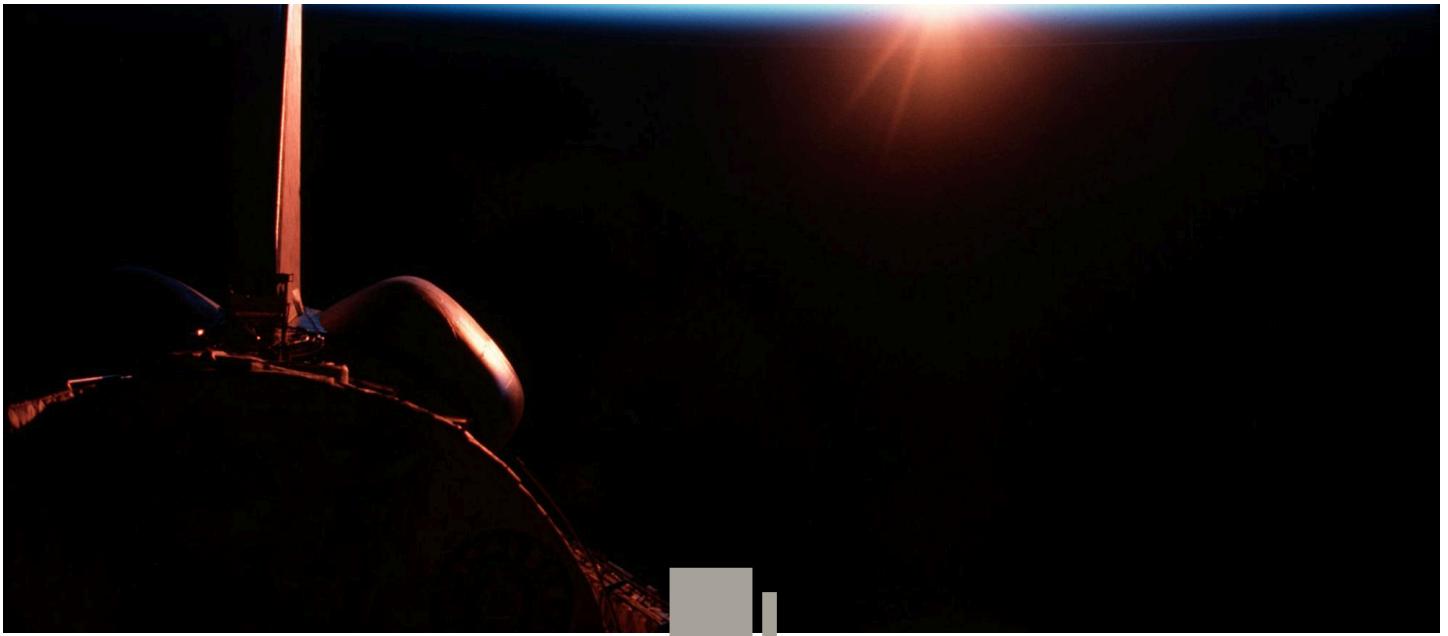
- **CSS:** In traditional CSS, you specify the `background-color` property to set the background color of an element.
- **Tailwind CSS:** In Tailwind, background color is set using utility classes like `bg-blue-500`, where the color is defined by the class name.

2. Text Color:

- **CSS:** In traditional CSS, the `color` property is used to set the text color of an element.
- **Tailwind CSS:** Tailwind uses utility classes like `text-green-600` to define the text color. The number in the class name represents the shade or intensity of the color.

3. Hover Effects:

- **CSS:** In CSS, hover effects are applied by using the `:hover` pseudo-class. In the example, the background color of a button changes when hovered over.
- **Tailwind CSS:** Tailwind simplifies hover effects by providing utility classes like `hover:bg-red-700`. This class is applied when the element is hovered over, changing the background color.



Week 8.2

Recap Everything, Build PayTM Backend

In this lecture, Harkirat guides us through an [end-to-end tutorial](#) on building a comprehensive [full-stack application](#) resembling [Paytm](#). While there are no specific notes provided for this section, a mini guide is outlined below to assist you in navigating through each step of the tutorial. Therefore, it is strongly advised to actively follow along during the lecture for a hands-on learning experience.

It's important to note that this session primarily focuses on the backend section of the application. For the frontend portion, be sure to check out the content covered in the subsequent 8.4 lecture.

Recap Everything, Build PayTM Backend

Step 1 - What are we building, Clone the starter repo

Things to do

Explore the repository

Backend

Frontend

Step 2 - User Mongoose schemas

Solution

Step 3 - Create routing file structure

[Step 1](#)[Solution](#)[Step 2](#)[Solution](#)[Step 4 - Route user requests](#)[1. Create a new user router](#)[Solution](#)[2. Create a new user router](#)[Solution](#)[Step 5 - Add cors, body parser and jsonwebtoken](#)[1. Add cors](#)[Hint](#)[Solution](#)[2. Add body-parser](#)[Hint](#)[Solution](#)[3. Add jsonwebtoken](#)[4. Export JWT_SECRET](#)[Solution](#)[5. Listen on port 3000](#)[Solution](#)[Step 6 - Add backend auth routes](#)[1. Signup](#)[Solution](#)[2. Route to sign in](#)[Solution](#)[Solution](#)[Step 7 - Middleware](#)[Solution](#)[Step 8 - User routes](#)[1. Route to update user information](#)[Solution](#)[2. Route to get users from the backend, filterable via firstName/lastName](#)

[Hints](#)[Solution](#)

[Step 9 - Create Bank related Schema](#)

[Accounts table](#)[Solution](#)[By the end of it, db.js should look like this](#)

[Step 10 - Transactions in databases](#)

[Solution](#)

[Step 11 - Initialize balances on signup](#)

[Solution](#)

[Step 12 - Create a new router for accounts](#)

[1. Create a new router](#)[Solution](#)[2. Route requests to it](#)[Solution](#)

[Step 13 - Balance and transfer Endpoints](#)

[1. An endpoint for user to get their balance.](#)[Solution](#)[2. An endpoint for user to transfer money to another account](#)[Bad Solution \(doesn't use transactions\)](#)[Good solution \(uses txns in db\)](#)[Problems you might run into](#)[Final Solution](#)[Finally, the account.js file should look like this](#)[Experiment to ensure transactions are working as expected](#)[Code](#)[Error](#)

[Step 14 - Checkpoint your solution](#)

[Get balance](#)[Make transfer](#)[Get balance again \(notice it went down\)](#)[Mongo should look something like this](#)

Step 1 - What are we building, Clone the starter repo

We're building a PayTM like application that let's users send money to each other given an initial dummy balance

The screenshot shows a web-based application titled "Payments App". At the top right, there is a user profile icon with the text "Hello, User" and a small "U". Below the title, it says "Your Balance \$5000". Under the heading "Users", there is a search bar with the placeholder "Search users...". Below the search bar, three user profiles are listed: "User 1" (U1), "User 2" (U2), and "User 3" (U3). To the right of each user profile is a "Send Money" button.

Things to do

Clone the 8.2 repository from <https://github.com/100xdevs-cohort-2/paytm>

```
git clone https://github.com/100xdevs-cohort-2/paytm
```



Please keep a MongoDB URL handy before you proceed. This will be your primary database for this assignment

1. Create a free one here - <https://www.mongodb.com/>
2. There is a Dockerfile in the codebase, you can run mongo locally using it.

Explore the repository

The repo is a basic **express + react + tailwind** boilerplate

Backend

1. Express - HTTP Server
2. mongoose - ODM to connect to MongoDB
3. zod - Input validation

```
// index.js
const express = require("express");
const app = express();
```

Frontend

1. React - Frontend framework
2. Tailwind - Styling framework

```
// App.jsx
function App() {

  return (
    <div>
      Hello world
    </div>
  )
}

export default App
```

Step 2 - User Mongoose schemas

We need to support 3 routes for user authentication

1. Allow user to sign up.
2. Allow user to sign in.
3. Allow user to update their information (firstName, lastName, password).

To start off, create the mongo schema for the users table

1. Create a new file (db.js) in the root folder
2. Import mongoose and connect to a database of your choice
3. Create the mongoose schema for the users table
4. Export the mongoose model from the file (call it User)

▼ Solution

Simple solution

```
// backend/db.js
const mongoose = require('mongoose');

// Create a Schema for Users
const userSchema = new mongoose.Schema({
    username: String,
    password: String,
    firstName: String,
    lastName: String
});

// Create a model from the schema
const User = mongoose.model('User', userSchema);

module.exports = {
    User
};
```

Elegant Solution

```
// backend/db.js
const mongoose = require('mongoose');

// Create a Schema for Users
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        lowercase: true,
        minLength: 3,
        maxLength: 30
    },
    password: {
        type: String,
        required: true,
        minLength: 6
    },
    firstName: {
        type: String,
        required: true,
        trim: true,
        lowercase: true
    }
});
```

```

maxLength: 50
},
lastName: {
  type: String,
  required: true,
  trim: true,
  maxLength: 50
}
});

// Create a model from the schema
const User = mongoose.model('User', userSchema);

module.exports = {
  User
};

```

Step 3 - Create routing file structure

In the index.js file, route all the requests to `/api/v1` to a apiRouter defined in `backend/routes/index.js`

Step 1

Create a new file `backend/routes/index.js` that exports a new express router.

(How to create a router - <https://www.geeksforgeeks.org/express-js-express-router-function/>)

▼ Solution

```

// backend/api/index.js
const express = require('express');

const router = express.Router();

module.exports = router;

```

Step 2

Import the router in index.js and route all requests from `/api/v1` to it

▼ Solution

```
// backend/index.js
const express = require("express");
const rootRouter = require("./routes/index");

const app = express();

app.use("/api/v1", rootRouter);
```

Step 4 - Route user requests

1. Create a new user router

Define a new router in `backend/routes/user.js` and import it in the index router.

Route all requests that go to `/api/v1/user` to the user router.

▼ Solution

```
// backend/routes/user.js
const express = require('express');

const router = express.Router();

module.exports = router;
```

2. Create a new user router

Import the userRouter in `backend/routes/index.js` so all requests to `/api/v1/user` get routed to the userRouter.

▼ Solution

```
// backend/routes/index.js
const express = require('express');
```

```
const userRouter = require("./user");

const router = express.Router();

router.use("/user", userRouter)

module.exports = router;
```

Step 5 - Add cors, body parser and jsonwebtoken

1. Add cors

Since our frontend and backend will be hosted on separate routes, add the `cors` middleware to `backend/index.js`

▼ Hint

Look at <https://www.npmjs.com/package/cors>

▼ Solution

```
// backend/index.js
const express = require('express');
const cors = require("cors");

app.use(cors());

const app = express();

module.exports = router;
```

2. Add body-parser

Since we have to support the JSON body in post requests, add the express body parser middleware to `backend/index.js`

You can use the `body-parser` npm library, or use `express.json`

▼ Hint

<https://medium.com/@mmajdanski/express-body-parser-and-why-may-not-need-it-335803cd048c>

▼ Solution

```
// backend/index.js
const express = require('express');
const cors = require("cors");
const rootRouter = require("./routes/index");

const app = express();

app.use(cors());
app.use(express.json());

app.use("/api/v1", rootRouter);
```

3. Add jsonwebtoken

We will be adding authentication soon to our application, so install jsonwebtoken library. It'll be useful in the next slide

```
npm install jsonwebtoken
```

4. Export JWT_SECRET

Export a JWT_SECRET from a new file `backend/config.js`

▼ Solution

```
//backend/config.js
module.exports = {
  JWT_SECRET: "your-jwt-secret"
}
```

5. Listen on port 3000

Make the express app listen on PORT 3000 of your machine

▼ Solution

```
// backend/index.js
...
app.listen(3000);
```

Step 6 - Add backend auth routes

In the user router (`backend/routes/user`), add 3 new routes.

1. Signup

This route needs to get user information, do input validation using zod and store the information in the database provided

1. Inputs are correct (validated via zod)
2. Database doesn't already contain another user

If all goes well, we need to return the user a jwt which has their user id encoded as follows -

```
{
  userId: "userId of newly added user"
}
```



Note - We are not hashing passwords before putting them in the database. This is standard practise that should be done, you can find more details here - <https://mojoauth.com/blog/hashing-passwords-in-nodejs/>

Method: POST

Route: /api/v1/user/signup

Body:

```
{
  username: "name@gmail.com",
```

```

  firstName: "name",
  lastName: "name",
  password: "123456"
}

```

Response:

Status code - 200

```
{
  message: "User created successfully",
  token: "jwt"
}
```

Status code - 411

```
{
  message: "Email already taken / Incorrect inputs"
}
```

▼ Solution

```

const zod = require("zod");
const { User } = require("../db");
const jwt = require("jsonwebtoken");
const { JWT_SECRET } = require("../config");

const signupBody = zod.object({
  username: zod.string().email(),
  firstName: zod.string(),
  lastName: zod.string(),
  password: zod.string()
})

router.post("/signup", async (req, res) => {
  const { success } = signupBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const existingUser = await User.findOne({
    username: req.body.username
  })

```

```
if (existingUser) {
    return res.status(411).json({
        message: "Email already taken/Incorrect inputs"
    })
}

const user = await User.create({
    username: req.body.username,
    password: req.body.password,
    firstName: req.body.firstName,
    lastName: req.body.lastName,
})
const userId = user._id;

const token = jwt.sign({
    userId
}, JWT_SECRET);

res.json({
    message: "User created successfully",
    token: token
})
})
```

2. Route to sign in

Let's an existing user sign in to get back a token.

Method: POST

Route: /api/v1/user/signin

Body:

```
{  
    username: "name@gmail.com",  
    password: "123456"  
}
```

Response:

Status code - 200

```
{  
    token: "jwt"  
}
```

Status code - 411

```
{  
  message: "Error while logging in"  
}
```

▼ Solution

```
const signinBody = zod.object({  
  username: zod.string().email(),  
  password: zod.string()  
})  
  
router.post("/signin", async (req, res) => {  
  const { success } = signinBody.safeParse(req.body)  
  if (!success) {  
    return res.status(411).json({  
      message: "Incorrect inputs"  
    })  
  }  
  
  const user = await User.findOne({  
    username: req.body.username,  
    password: req.body.password  
});  
  
  if (user) {  
    const token = jwt.sign({  
      userId: user._id  
    }, JWT_SECRET);  
  
    res.json({  
      token: token  
    })  
    return;  
  }  
  
  res.status(411).json({  
    message: "Error while logging in"  
  })  
})
```

By the end, `routes/user.js` should look like follows

▼ Solution

```
// backend/routes/user.js
const express = require('express');

const router = express.Router();
const zod = require("zod");
const { User } = require("../db");
const jwt = require("jsonwebtoken");
const { JWT_SECRET } = require("../config");

const signupBody = zod.object({
    username: zod.string().email(),
    firstName: zod.string(),
    lastName: zod.string(),
    password: zod.string()
})

router.post("/signup", async (req, res) => {
    const { success } = signupBody.safeParse(req.body)
    if (!success) {
        return res.status(411).json({
            message: "Email already taken / Incorrect inputs"
        })
    }

    const existingUser = await User.findOne({
        username: req.body.username
    })

    if (existingUser) {
        return res.status(411).json({
            message: "Email already taken/Incorrect inputs"
        })
    }

    const user = await User.create({
        username: req.body.username,
        password: req.body.password,
        firstName: req.body.firstName,
        lastName: req.body.lastName,
    })
    const userId = user._id;
```

```
const token = jwt.sign({
  userId
}, JWT_SECRET);

res.json({
  message: "User created successfully",
  token: token
})
})

const signinBody = zod.object({
  username: zod.string().email(),
  password: zod.string()
})

router.post("/signin", async (req, res) => {
  const { success } = signinBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const user = await User.findOne({
    username: req.body.username,
    password: req.body.password
  });

  if (user) {
    const token = jwt.sign({
      userId: user._id
    }, JWT_SECRET);

    res.json({
      token: token
    })
    return;
  }

  res.status(411).json({
    message: "Error while logging in"
  })
})

module.exports = router;
```

Step 7 - Middleware

Now that we have a user account, we need to `gate` routes which authenticated users can hit.

For this, we need to introduce an auth middleware

Create a `middleware.js` file that exports an `authMiddleware` function

1. Checks the headers for an Authorization header `Bearer <token>`
2. Verifies that the token is valid
3. Puts the `userId` in the request object if the token checks out.
4. If not, return a 403 status back to the user

Header -

`Authorization: Bearer <actual token>`

▼ Solution

```
const { JWT_SECRET } = require("./config");
const jwt = require("jsonwebtoken");

const authMiddleware = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith('Bearer ')) {
        return res.status(403).json({});
    }

    const token = authHeader.split(' ')[1];

    try {
        const decoded = jwt.verify(token, JWT_SECRET);

        req.userId = decoded.userId;

        next();
    } catch (err) {
        return res.status(403).json({});
    }
};

module.exports = {
    authMiddleware
}
```

Step 8 - User routes

1. Route to update user information

User should be allowed to `optionally` send either or all of

1. password
2. firstName
3. lastName

Whatever they send, we need to update it in the database for the user.

Use the `middleware` we defined in the last section to authenticate the user

Method: PUT

Route: /api/v1/user

Body:

```
{  
  password: "new_password",  
  firstName: "updated_first_name",  
  lastName: "updated_first_name",  
}
```

Response:

Status code - 200

```
{  
  message: "Updated successfully"  
}
```

Status code - 411 (Password is too small...)

```
{  
  message: "Error while updating information"  
}
```

▼ Solution

```

const { authMiddleware } = require("../middleware");

// other auth routes

const updateBody = zod.object({
  password: zod.string().optional(),
  firstName: zod.string().optional(),
  lastName: zod.string().optional(),
})

router.put("/", authMiddleware, async (req, res) => {
  const { success } = updateBody.safeParse(req.body)
  if (!success) {
    res.status(411).json({
      message: "Error while updating information"
    })
  }

  await User.updateOne(req.body, {
    _id: req.userId
  })

  res.json({
    message: "Updated successfully"
  })
})

```

2. Route to get users from the backend, filterable via firstName/lastName

This is needed so users can search for their friends and send them money

Method: GET

Route: /api/v1/user/bulk

Query Parameter: ?filter=harkirat

Response:

Status code - 200

```
{
  users: [
    {
      firstName: "",
      lastName: "",
      _id: "id of the user"
    }
  ]
}
```

```
}]  
}
```

▼ Hints

<https://stackoverflow.com/questions/7382207/mongooses-find-method-with-or-condition-does-not-work-properly>

<https://stackoverflow.com/questions/3305561/how-to-query-mongodb-with-like>

▼ Solution

```
router.get("/bulk", async (req, res) => {  
    const filter = req.query.filter || "";  
  
    const users = await User.find({  
        $or: [{  
            firstName: {  
                "$regex": filter  
            }  
        }, {  
            lastName: {  
                "$regex": filter  
            }  
        }]  
    })  
  
    res.json({  
        user: users.map(user => ({  
            username: user.username,  
            firstName: user.firstName,  
            lastName: user.lastName,  
            _id: user._id  
        }))  
    })
})
```

Step 9 - Create Bank related Schema

Update the `db.js` file to add one new schemas and export the respective models

Accounts table

The `Accounts` table will store the INR balances of a user.

The schema should look something like this -

```
{
  userId: ObjectId (or string),
  balance: float/number
}
```

In the real world, you shouldn't store `floats` for balances in the database. You usually store an integer which represents the INR value with decimal places (for eg, if someone has 33.33 rs in their account, you store 3333 in the database).

There is a certain precision that you need to support (which for india is 2/4 decimal places) and this allows you to get rid of precision errors by storing integers in your DB

You should reference the users table in the schema (Hint -

<https://medium.com/@mendes.develop/joining-tables-in-mongodb-with-mongoose-489d72c84b60>

▼ Solution

```
const accountSchema = new mongoose.Schema({
  userId: {
    type: mongoose.Schema.Types.ObjectId, // Reference to User model
    ref: 'User',
    required: true
  },
  balance: {
    type: Number,
    required: true
  }
});

const Account = mongoose.model('Account', accountSchema);

module.exports = {
  Account
}
```

▼ By the end of it, `db.js` should look lie this

```
// backend/db.js
const mongoose = require('mongoose');

mongoose.connect("mongodb://localhost:27017/paytm")

// Create a Schema for Users
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        lowercase: true,
        minLength: 3,
        maxLength: 30
    },
    password: {
        type: String,
        required: true,
        minLength: 6
    },
    firstName: {
        type: String,
        required: true,
        trim: true,
        maxLength: 50
    },
    lastName: {
        type: String,
        required: true,
        trim: true,
        maxLength: 50
    }
});

const accountSchema = new mongoose.Schema({
    userId: {
        type: mongoose.Schema.Types.ObjectId, // Reference to User model
        ref: 'User',
        required: true
    },
    balance: {
        type: Number,
        required: true
    }
});
```

```
const Account = mongoose.model('Account', accountSchema);
const User = mongoose.model('User', userSchema);

module.exports = {
  User,
  Account,
};
```

Step 10 - Transactions in databases

A lot of times, you want multiple databases transactions to be **atomic**

Either all of them should update, or none should

This is super important in the case of a **bank**

Can you guess what's wrong with the following code -

```
const mongoose = require('mongoose');
const Account = require('./path-to-your-account-model');

const transferFunds = async (fromAccountId, toAccountId, amount) => {
  // Decrement the balance of the fromAccount
  await Account.findByIdAndUpdate(fromAccountId, { $inc: { balance: -amount } });

  // Increment the balance of the toAccount
  await Account.findByIdAndUpdate(toAccountId, { $inc: { balance: amount } });
}

// Example usage
transferFunds('fromAccountID', 'toAccountID', 100);
```

▼ Solution

1. What if the database crashes right after the first request (only the balance is decreased for one user, and not for the second user)
2. What if the Node.js crashes after the first update?

It would lead to a **database inconsistency**. Amount would get debited from the first user, and not credited into the other users account.

If a failure ever happens, the first txn should rollback.

This is what is called a **transaction** in a database. We need to implement a **transaction** on the next set of endpoints that allow users to transfer INR

Step 11 - Initialize balances on signup

Update the `signup` endpoint to give the user a random balance between 1 and 10000.

This is so we don't have to integrate with banks and give them random balances to start with.

▼ Solution

```
router.post("/signup", async (req, res) => {
  const { success } = signupBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const existingUser = await User.findOne({
    username: req.body.username
  })

  if (existingUser) {
    return res.status(411).json({
      message: "Email already taken/Incorrect inputs"
    })
  }

  const user = await User.create({
    username: req.body.username,
    password: req.body.password,
    firstName: req.body.firstName,
    lastName: req.body.lastName,
  })
  const userId = user._id;

  // --- Create new account ----

  await Account.create({
    userId,
    balance: 1 + Math.random() * 10000
  })

  // --- ---

  const token = jwt.sign({
    userId
  }, JWT_SECRET);

  res.json({
    token
  })
})
```

```
    message: "User created successfully",
    token: token
  })
}
```

Step 12 - Create a new router for accounts

1. Create a new router

All user balances should go to a different express router (that handles all requests on `/api/v1/account`).

Create a new router in `routes/account.js` and add export it

▼ Solution

```
// backend/routes/account.js
const express = require('express');

const router = express.Router();

module.exports = router;
```

2. Route requests to it

Send all requests from `/api/v1/account/*` in `routes/index.js` to the router created in step 1.

▼ Solution

```
// backend/user/index.js
const express = require('express');
const userRouter = require("./user");
const accountRouter = require("./account");

const router = express.Router();

router.use("/user", userRouter);
router.use("/account", accountRouter);

module.exports = router;
```

Step 13 - Balance and transfer Endpoints

Here, you'll be writing a bunch of APIs for the core user balances. There are 2 endpoints that we need to implement

1. An endpoint for user to get their balance.

Method: GET

Route: /api/v1/account/balance

Response:

Status code - 200

```
{
  balance: 100
}
```

▼ Solution

```
router.get("/balance", authMiddleware, async (req, res) => {
  const account = await Account.findOne({
    userId: req.userId
  });

  res.json({
    balance: account.balance
  });
});
```

2. An endpoint for user to transfer money to another account

Method: POST

Route: /api/v1/account/transfer

Body

```
{
  to: string,
  amount: number
}
```

Response:

Status code - 200

```
{  
  message: "Transfer successful"  
}
```

Status code - 400

```
{  
  message: "Insufficient balance"  
}
```

Status code - 400

```
{  
  message: "Invalid account"  
}
```

▼ Bad Solution (doesn't use transactions)

```
router.post("/transfer", authMiddleware, async (req, res) => {  
  const { amount, to } = req.body;  
  
  const account = await Account.findOne({  
    userId: req.userId  
  });  
  
  if (account.balance < amount) {  
    return res.status(400).json({  
      message: "Insufficient balance"  
    })
  }  
  
  const toAccount = await Account.findOne({  
    userId: to  
  });  
  
  if (!toAccount) {  
    return res.status(400).json({  
      message: "Invalid account"  
    })
  }  
  
  await Account.updateOne({  
    userId: req.userId  
  }
```

```

}, {
  $inc: {
    balance: -amount
  }
})

await Account.updateOne({
  userId: to
}, {
  $inc: {
    balance: amount
  }
})
}

res.json({
  message: "Transfer successful"
})
});

);

```

▼ Good solution (uses txns in db

```

router.post("/transfer", authMiddleware, async (req, res) => {
  const session = await mongoose.startSession();

  session.startTransaction();
  const { amount, to } = req.body;

  // Fetch the accounts within the transaction
  const account = await Account.findOne({ userId: req.userId }).session(session);

  if (!account || account.balance < amount) {
    await session.abortTransaction();
    return res.status(400).json({
      message: "Insufficient balance"
    });
  }

  const toAccount = await Account.findOne({ userId: to }).session(session);

  if (!toAccount) {
    await session.abortTransaction();
    return res.status(400).json({
      message: "Invalid account"
    });
  }
}

```

```
// Perform the transfer
await Account.updateOne({ userId: req.userId }, { $inc: { balance: -amount } });
await Account.updateOne({ userId: to }, { $inc: { balance: amount } }).session(session);

// Commit the transaction
await session.commitTransaction();
res.json({
  message: "Transfer successful"
});
});
```

▼ Problems you might run into

Problems you might run into If you run into the problem mentioned above, feel free to proceed with the bad solution

<https://stackoverflow.com/questions/51461952/mongodb-v4-0-transaction-mongoerror-transaction-numbers-are-only-allowed-on-a>

Final Solution

▼ Finally, the account.js file should look like this



Experiment to ensure transactions are working as expected

Try running this code locally. It calls transfer twice on the same account ~almost concurrently

▼ Code

```
// backend/routes/account.js
const express = require('express');
const { authMiddleware } = require('../middleware');
const { Account } = require('../db');
const { default: mongoose } = require('mongoose');

const router = express.Router();

router.get("/balance", authMiddleware, async (req, res) => {
    const account = await Account.findOne({
        userId: req.userId
    });

    res.json({
        balance: account.balance
    })
});

async function transfer(req) {
    const session = await mongoose.startSession();

    session.startTransaction();
    const { amount, to } = req.body;

    // Fetch the accounts within the transaction
    const account = await Account.findOne({ userId: req.userId }).session(session);

    if (!account || account.balance < amount) {
        await session.abortTransaction();
        console.log("Insufficient balance")
        return;
    }

    const toAccount = await Account.findOne({ userId: to }).session(session);

    if (!toAccount) {
        await session.abortTransaction();
        console.log("Invalid account")
        return;
    }

    // Perform the transfer
    await Account.updateOne({ userId: req.userId }, { $inc: { balance: -amount } }).session(session);
    await Account.updateOne({ userId: to }, { $inc: { balance: amount } }).session(session);

    // Commit the transaction
    await session.commitTransaction();
    console.log("done")
}
}
```

```

transfer({
  userId: "65ac44e10ab2ec750ca666a5",
  body: {
    to: "65ac44e40ab2ec750ca666aa",
    amount: 100
  }
})

transfer({
  userId: "65ac44e10ab2ec750ca666a5",
  body: {
    to: "65ac44e40ab2ec750ca666aa",
    amount: 100
  }
})
module.exports = router;

```

▼ Error

```

balance: 2099.53996900008773,
} _v: 0
/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/connection.js:205
      callback(new error_1.MongoServerError(document));
^

MongoServerError: WriteConflict error: this operation conflicted with another operation. Please retry your operation or multi-document transaction.
  at Connection.onMessage (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/connection.js:205:26)
  at MessageStream.<anonymous> (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/connection.js:64:60)
  at MessageStream.emit (node:events:519:28)
  at processIncomingData (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/message_stream.js:117:16)
  at MessageStream._write (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/message_stream.js:33:9)
  at writeOrBuffer (node:internal/streams/writable:564:12)
  at _write (node:internal/streams/writable:493:10)
  at Writable.write (node:internal/streams/writable:502:10)
  at Socket.ondata (node:internal/streams/readable:1007:22)
  at Socket.emit (node:events:519:28) {
  operationTime: Timestamp { low: 1, high: 1705839061, unsigned: true },
  ok: 0,
  code: 112,
}

```

Step 14 - Checkpoint your solution

A completely working backend can be found here - <https://github.com/100xdevs-cohort-2/paytm/tree/backend-solution>

Try to send a few calls via postman to ensure you are able to sign up/sign in/get balance

Get balance

The screenshot shows the Postman interface with a dark theme. On the left, there's a sidebar titled "History" showing a list of recent API calls. In the main area, a new request is being configured:

- Method:** GET
- URL:** `http://localhost:3000/api/v1/account/balance`
- Headers:** (9)
 - Authorization:** Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJc2VySW...
- Body:** (8)
- Test Results:** (8)

The response pane shows a 200 OK status with a JSON body containing a single key-value pair:

```
1: "balance": 2099.5399690668773
```

Make transfer

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:3000/api/v1/account/transfer`. The method is `POST`. The response status is `200 OK` with a time of `54 ms` and a size of `300 B`. The response body is:

```

1  "to": "65ac44e10ab2ec750ca666a5",
2  "amount": "100"
3
4
    
```

Get balance again (notice it went down)

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:3000/api/v1/account/balance`. The method is `GET`. The response status is `200 OK` with a time of `21 ms` and a size of `297 B`. The response body is:

```

1  "balance": 1999.5399690668773
2
3
    
```

Mongo should look something like this

This screenshot shows the MongoDB Compass interface for the `paytm.accounts` collection. The left sidebar lists databases and collections, with `paytm` and `accounts` selected. The main pane displays three documents:

```

_id: ObjectId('65ac44df0ab2ec750ca666a2')
userId: ObjectId('65ac44df0ab2ec750ca666a0')
balance: 1736.5183102866142
__v: 0

_id: ObjectId('65ac44e10ab2ec750ca666a7')
userId: ObjectId('65ac44e10ab2ec750ca666a5')
balance: 513.8107076284458
__v: 0

_id: ObjectId('65ac44e40ab2ec750ca666ac')
userId: ObjectId('65ac44e40ab2ec750ca666aa')
balance: 1999.5399690668773
__v: 0
  
```

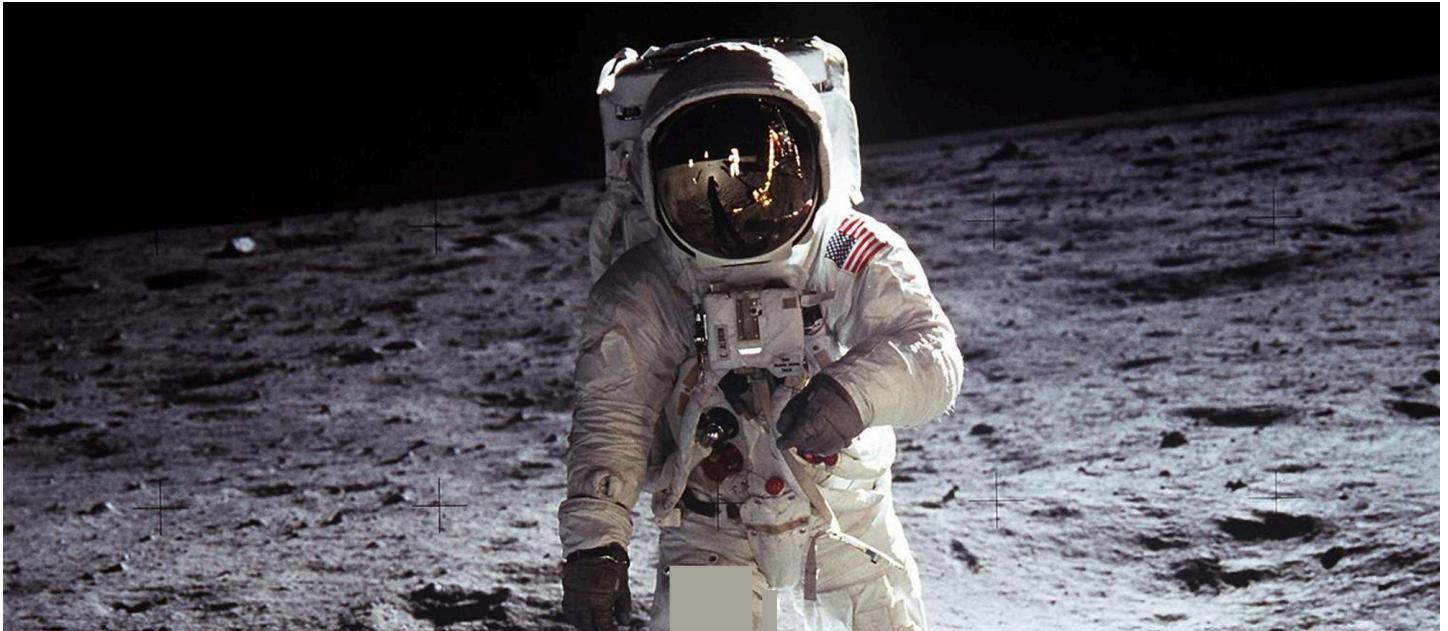
This screenshot shows the MongoDB Compass interface for the `paytm.users` collection. The left sidebar lists databases and collections, with `paytm` and `users` selected. The main pane displays three documents:

```

_id: ObjectId('65ac44df0ab2ec750ca666a0')
username: "harkirat1111@gmail.com"
password: "123456"
firstName: "harkirat"
lastName: "Singh"
__v: 0

_id: ObjectId('65ac44e10ab2ec750ca666a5')
username: "harkirat1111@gmail.com"
password: "123456"
firstName: "harkirat"
lastName: "Singh"
__v: 0

_id: ObjectId('65ac44e40ab2ec750ca666aa')
username: "harkirat11@gmail.com"
password: "123456"
firstName: "harkirat"
lastName: "Singh"
__v: 0
  
```



Week 8.3

Axios vs Fetch (Offline)

In this quick offline tutorial, Harkirat covers the [Fetch method](#) and [Axios](#) for data fetching in web development. He explains the differences between them and provides straightforward code examples for both, making it easy to grasp the essentials of these commonly used techniques.

[Axios vs Fetch \(Offline\)](#)

[fetch\(\) Method](#)

[What is the fetch\(\) Method?](#)

[Why is it Used?](#)

[Basic Example:](#)

[fetch\(\) vs axios\(\)](#)

[Fetch API](#)

[Axios](#)

[Comparison Points](#)

[Conclusion](#)

[Comparing Code Implementation](#)

[Fetch \(GET Request\)](#)

[Axios \(GET Request\)](#)

[Fetch \(POST Request\)](#)

[Axios \(POST Request\)](#)

fetch() Method

The `fetch()` method in JavaScript is a modern API that allows you to make network requests, typically to retrieve data from a server. It is commonly used to interact with web APIs and fetch data asynchronously. Here's a breakdown of what the `fetch()` method is and why it's used:

What is the `fetch()` Method?

The `fetch()` method is a built-in JavaScript function that simplifies making HTTP requests. It returns a Promise that resolves to the `Response` to that request, whether it is successful or not.

Why is it Used?

1. Asynchronous Data Retrieval:

- The primary use of the `fetch()` method is to asynchronously retrieve data from a server. Asynchronous means that the code doesn't wait for the data to arrive before moving on. This is crucial for creating responsive and dynamic web applications.

2. Web API Interaction:

- Many web applications interact with external services or APIs to fetch data. The `fetch()` method simplifies the process of making HTTP requests to these APIs.

3. Promise-Based:

- The `fetch()` method returns a Promise, making it easy to work with asynchronous operations using the `.then()` and `.catch()` methods. This promotes cleaner and more readable code.

4. Flexible and Powerful:

- `fetch()` is more flexible and powerful compared to older methods like `XMLHttpRequest`. It supports a wide range of options, including headers, request methods, and handling different types of responses (JSON, text, etc.).

Basic Example:

Here's a basic example of using the `fetch()` method to retrieve data from a server:

```
fetch('<https://api.example.com/data>')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
```

```
.then(data => {
  console.log('Data from server:', data);
})
.catch(error => {
  console.error('Fetch error:', error);
});
```

In this example, we use `fetch()` to make a GET request to '<https://api.example.com/data>', handle the response, and then parse the JSON data. The `.then()` and `.catch()` methods allow us to handle the asynchronous flow and potential errors.

fetch() vs axios()

Fetch API

1. Native Browser API:

- `fetch` is a native JavaScript function built into modern browsers for making HTTP requests.

2. Promise-Based:

- It returns a Promise, allowing for a more modern asynchronous coding style with `async/await` or using `.then()`.

3. Lightweight:

- `fetch` is lightweight and comes bundled with browsers, reducing the need for external dependencies.

Example Usage:

```
fetch('<https://api.example.com/data>')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Axios

1. External Library:

- Axios is a standalone JavaScript library designed to work in both browsers and Node.js environments.

2. Promise-Based:

- Similar to `fetch`, Axios also returns a Promise, providing a consistent interface for handling asynchronous operations.

3. HTTP Request and Response Interceptors:

- Axios allows the use of interceptors, enabling the modification of requests or responses globally before they are handled by `then` or `catch`.

4. Automatic JSON Parsing:

- Axios automatically parses JSON responses, simplifying the process compared to `fetch`.

Example Usage:

```
import axios from 'axios';

axios.get('<https://api.example.com/data>')
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

Comparison Points

1. Syntax:

- `fetch` uses a chain of `.then()` to handle responses, which might lead to a more verbose syntax. Axios, on the other hand, provides a concise syntax with `.then()` directly on the Axios instance.

2. Handling HTTP Errors:

- Both `fetch` and Axios allow error handling using `.catch()` or `.finally()`, but Axios may provide more detailed error information by default.

3. Interceptors:

- Axios provides a powerful feature with interceptors for both requests and responses, allowing global modifications. `fetch` lacks built-in support for interceptors.

4. Request Configuration:

- Axios allows detailed configuration of requests through a variety of options. `fetch` may require more manual setup for headers, methods, and other configurations.

5. JSON Parsing:

- Axios automatically parses JSON responses, while with `fetch`, you need to manually call `.json()` on the response.

6. Browser Support:

- `fetch` is natively supported in modern browsers, but if you need to support older browsers, you might need a polyfill. Axios has consistent behavior across various browsers and does not rely on native implementations.

7. Size:

- `fetch` is generally considered lightweight, being a part of the browser. Axios, being a separate library, introduces an additional file size to your project.

Conclusion

- Use `fetch` when:

- Working on a modern project without the need for additional features.
 - Prefer a lightweight solution and have no concerns about polyfills.

- Use Axios when:

- Dealing with more complex scenarios such as interceptors.
 - Needing consistent behavior across different browsers.
 - Desiring a library with built-in features like automatic JSON parsing.

In summary, both `fetch` and Axios have their strengths, and the choice depends on the specific requirements and preferences of the project. `fetch` is excellent for simplicity and lightweight projects, while Axios provides additional features and consistent behavior across different environments.

Comparing Code Implementation

Below are code snippets for making GET and POST requests using `async/await` with Fetch and Axios:
[Good Resource to Inspect Incoming HTTP Requests](#)

Fetch (GET Request)

```
async function fetchData() {  
  try {  
    const response = await fetch('<https://api.example.com/data>');  
    const data = await response.json();  
    console.log(data);  
  } catch (error) {
```

```
        console.error('Error:', error);
    }
}

fetchData();
```

Axios (GET Request)

```
// Install Axios using npm or yarn: npm install axios
import axios from 'axios';

async function fetchData() {
    try {
        const response = await axios.get('<https://api.example.com/data>');
        console.log(response.data);
    } catch (error) {
        console.error('Error:', error);
    }
}

fetchData();
```

Fetch (POST Request)

```
async function postData() {
    const url = '<https://api.example.com/postData>';
    const dataToSend = {
        key1: 'value1',
        key2: 'value2',
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(dataToSend),
        });
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
}
```

```
postData();
```

Axios (POST Request)

```
// Install Axios using npm or yarn: npm install axios
import axios from 'axios';

async function postData() {
  const url = '<https://api.example.com/postData>';
  const dataToSend = {
    key1: 'value1',
    key2: 'value2',
  };

  try {
    const response = await axios.post(url, dataToSend);
    console.log(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
}

postData();
```

The above examples demonstrate how to use `async/await` with Fetch and Axios to make asynchronous HTTP requests. In GET Requests although you can send HEADERS but you cannot send BODY while in case of POST, DELETE, PUT Requests you can send both HEADERS as well as BODY.



Week 8.4

Recap Everything, Build PayTM Frontend

In this lecture, Harkirat guides us through an [end-to-end tutorial](#) on building a comprehensive [full-stack application](#) resembling [Paytm](#). While there are no specific notes provided for this section, a mini guide is outlined below to assist you in navigating through each step of the tutorial. Therefore, it is strongly advised to actively follow along during the lecture for a hands-on learning experience.

It's important to note that this session primarily focuses on the frontend section of the application. For the backend portion, be sure to check out the content covered in the previous 8.2 lecture.

[Recap Everything, Build PayTM Frontend](#)

[Step 1 - Add routing to the react app](#)

[Solution](#)

[Step 2 - Create and hook up Signup page](#)

[Step 3 - Create the signin page](#)

[Step 4 - Dashboard page](#)

[Step 5 - Auth Components](#)

[1. Heading component](#)

[Code](#)

[2. Sub Heading component](#)[Code](#)[3. InputBox component](#)[Code](#)[4. Button Component](#)[Code](#)[5. BottomWarning](#)[Code](#)[Full Signup component](#)[Code](#)[Full Signin component](#)[Code](#)[Step 6 - Signin-ed Comonents](#)[1. Appbar](#)[Code](#)[2. Balance](#)[Code](#)[3. Users component](#)[Code](#)[4. SendMoney Component](#)[Code](#)[Step 7 - Wiring up the backend calls](#)

Step 1 - Add routing to the react app

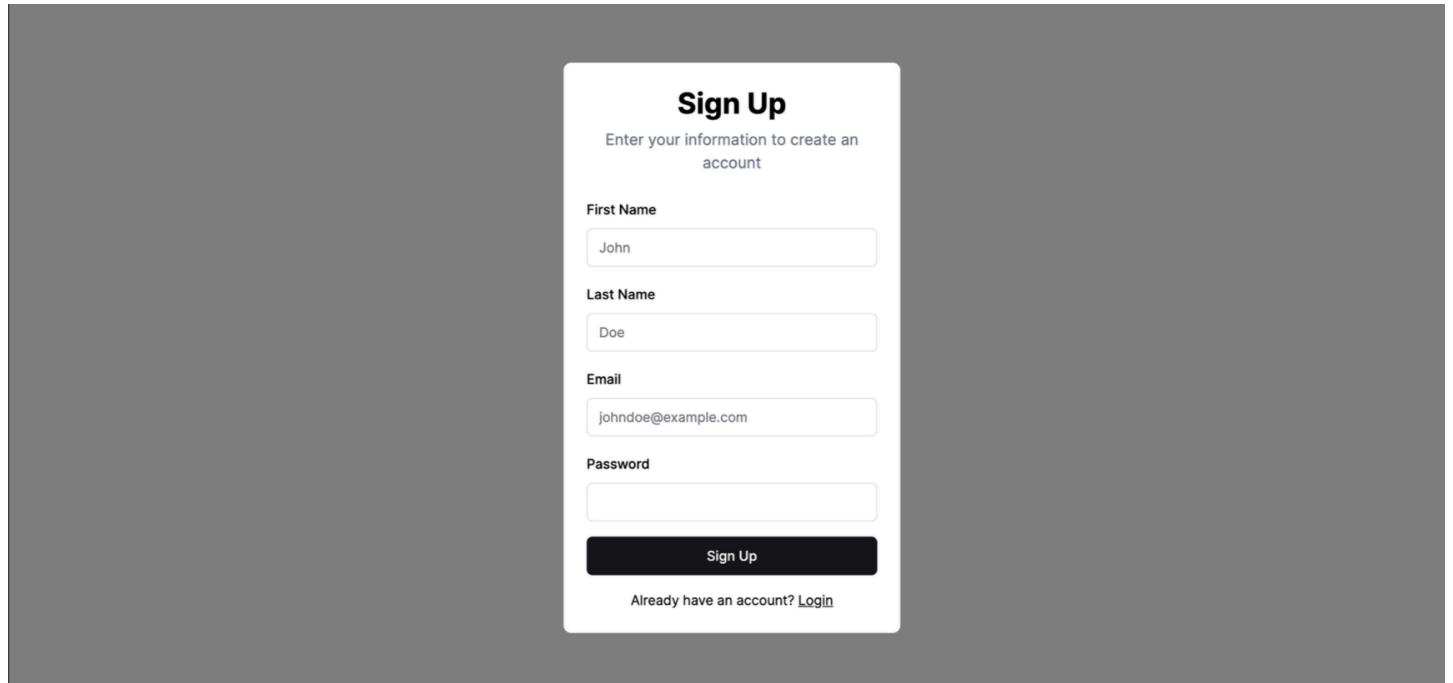
Import `react-router-dom` into your project and add the following routes -

1. `/signup` - The signup page
2. `/signin` - The signin page
3. `/dashboard` - Balances and see other users on the platform.
4. `/send` - Send money to other users

▼ Solution

```
function App() {
  return (
    <>
    <BrowserRouter>
      <Routes>
        <Route path="/signup" element={<Signup />} />
        <Route path="/signin" element={<Signin />} />
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/send" element={<SendMoney />} />
      </Routes>
    </BrowserRouter>
  </>
)
}
```

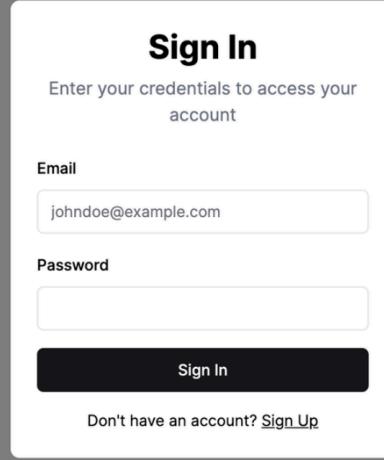
Step 2 - Create and hook up Signup page



If the user signup is successful, take the user to [/dashboard](#)

If not, show them an error message

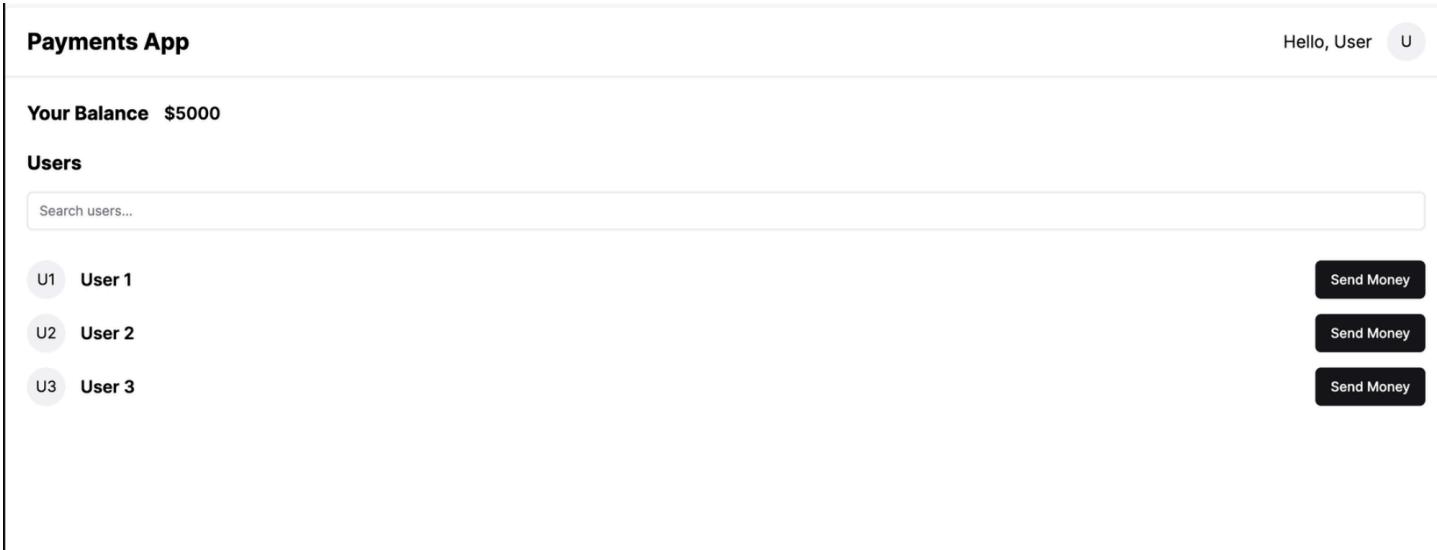
Step 3 - Create the signin page



A screenshot of a sign-in form titled "Sign In". The form instructions say "Enter your credentials to access your account". It has fields for "Email" (containing "johndoe@example.com") and "Password". A "Sign In" button is at the bottom, and a link "Don't have an account? [Sign Up](#)" is below it.

If the signin is successful, take the user to </dashboard>

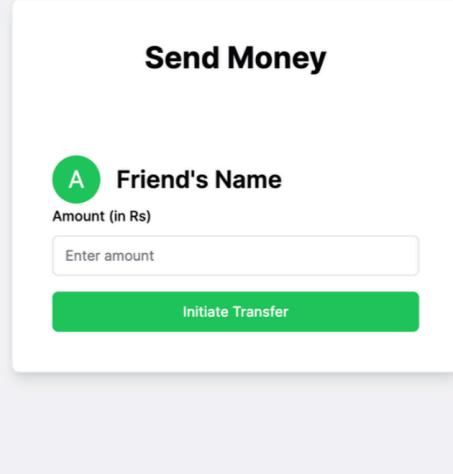
Step 4 - Dashboard page



A screenshot of a dashboard page for a "Payments App". The top navigation bar shows "Hello, User" and a user icon. The main content area starts with "Your Balance \$5000". Below that is a "Users" section with a search bar labeled "Search users...". Three users are listed: "User 1", "User 2", and "User 3", each with a "Send Money" button to their right.

Show the user their balance, and a list of users that exist in the database

Clicking on [Send money](#) should open a modal that lets the user send money

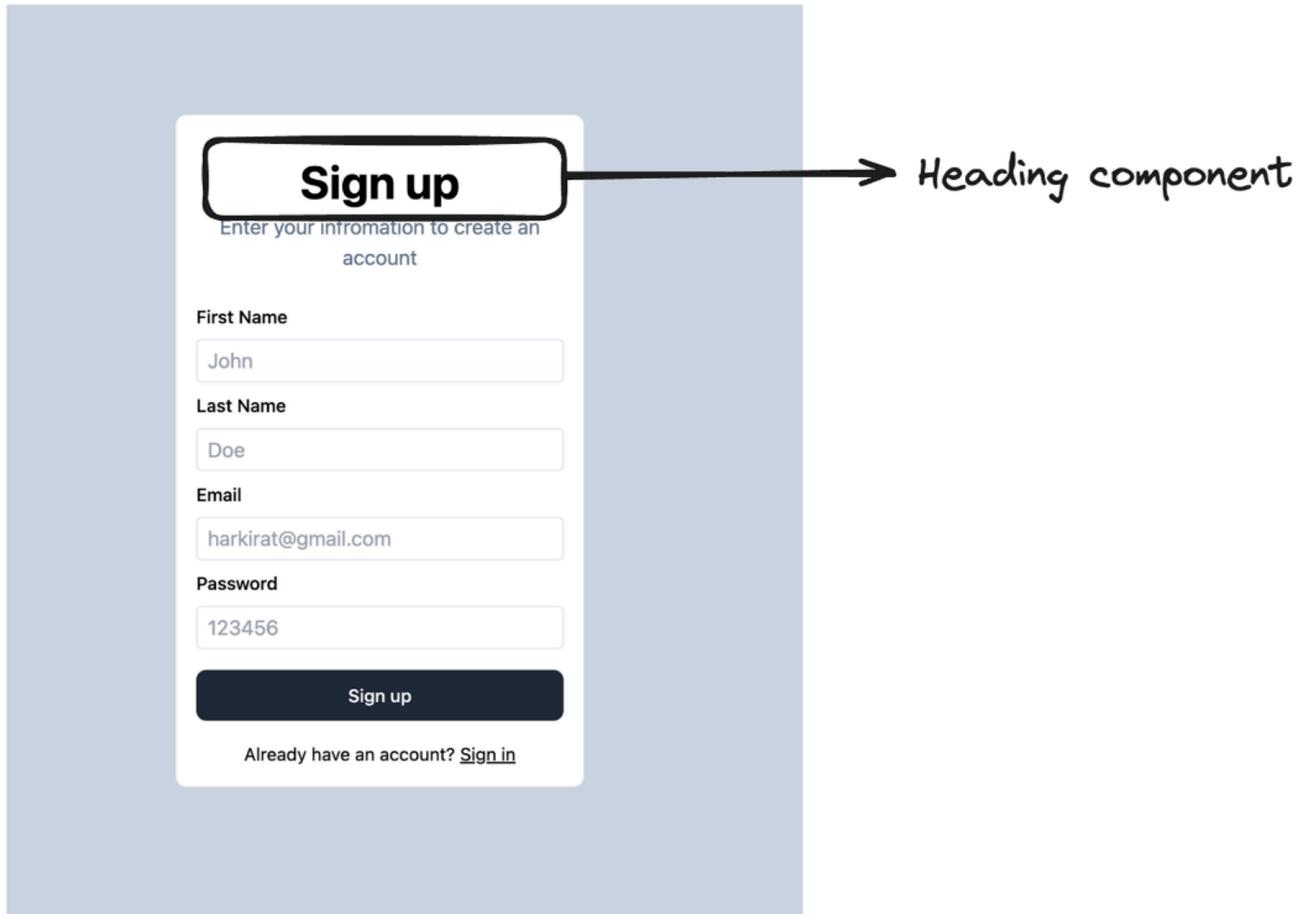


Step 5 - Auth Components

Full Signup component

You can break down the app into a bunch of components. The code only contains the styles of the component, not any onclick functionality.

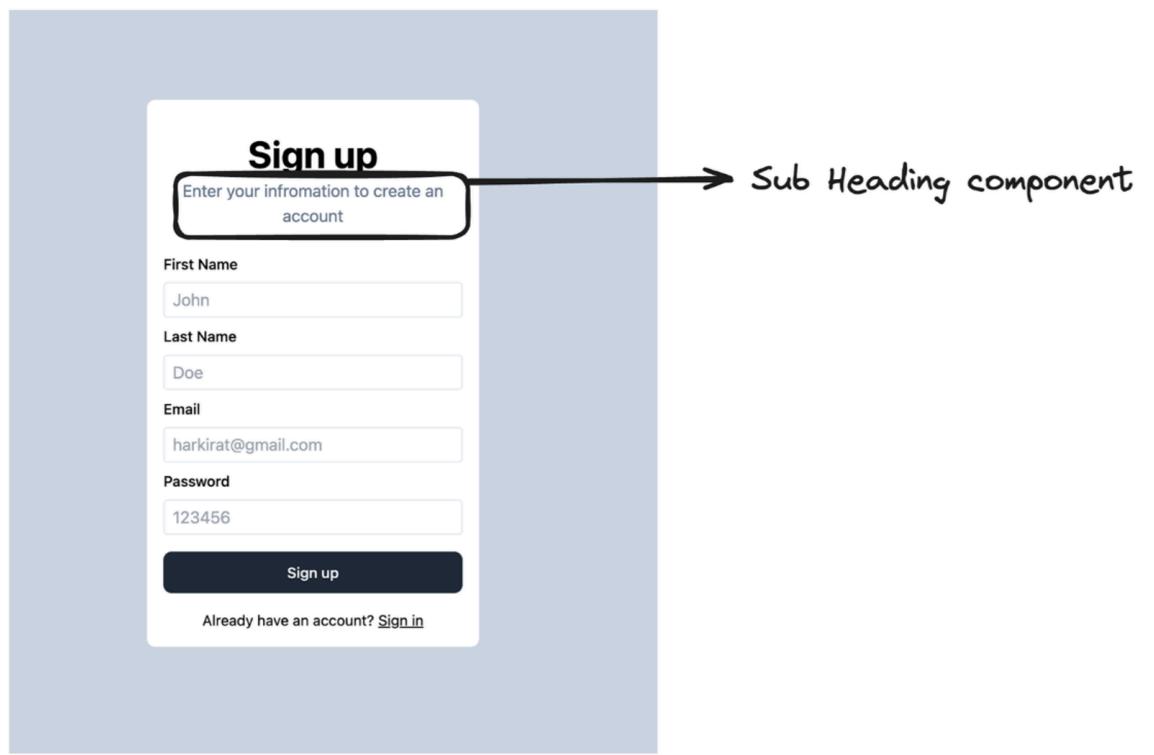
1. Heading component



▼ Code

```
export function Heading({label}) {  
  return <div className="font-bold text-4xl pt-6">  
    {label}  
  </div>  
}
```

2. Sub Heading component

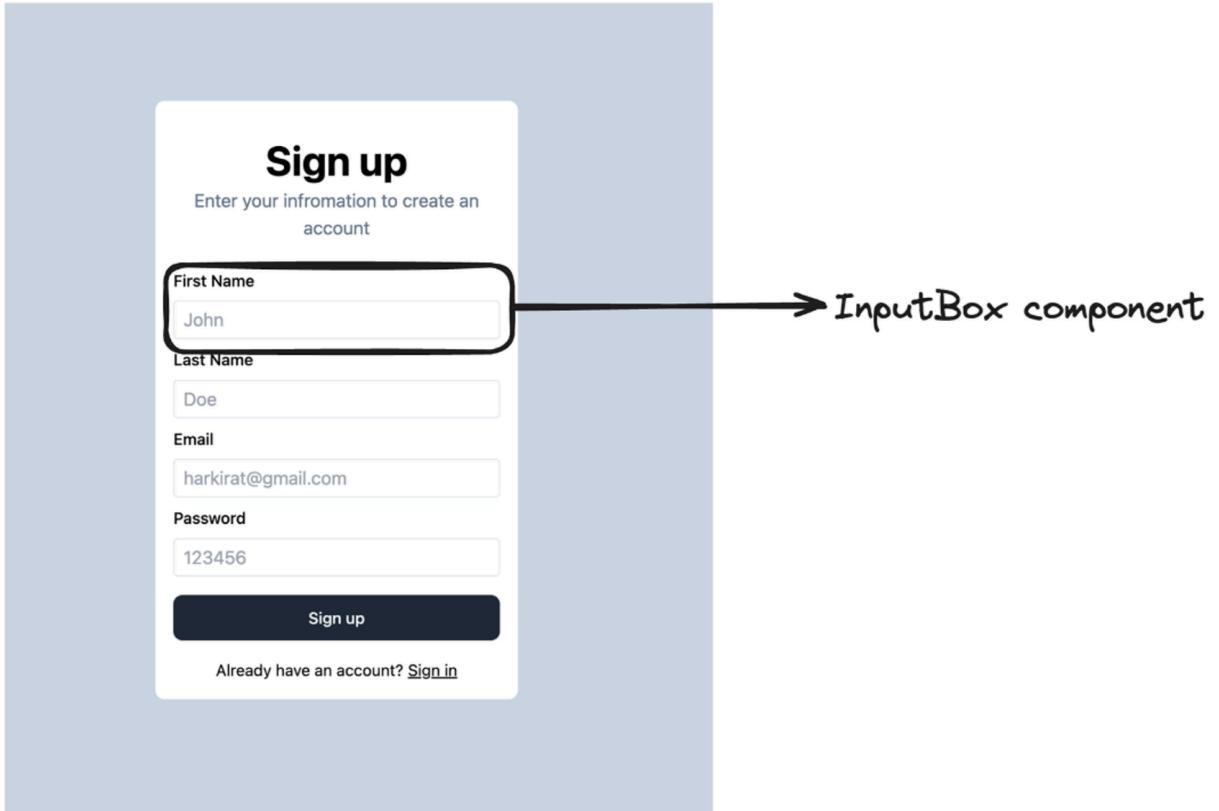


▼ Code

```
export function SubHeading({label}) {  
  return <div className="text-slate-500 text-md pt-1 px-4 pb-4">  
    {label}  
  </div>  
}
```

3. InputBox component

To move canvas, hold mouse wheel or spacebar while dragging, or use the hand tool

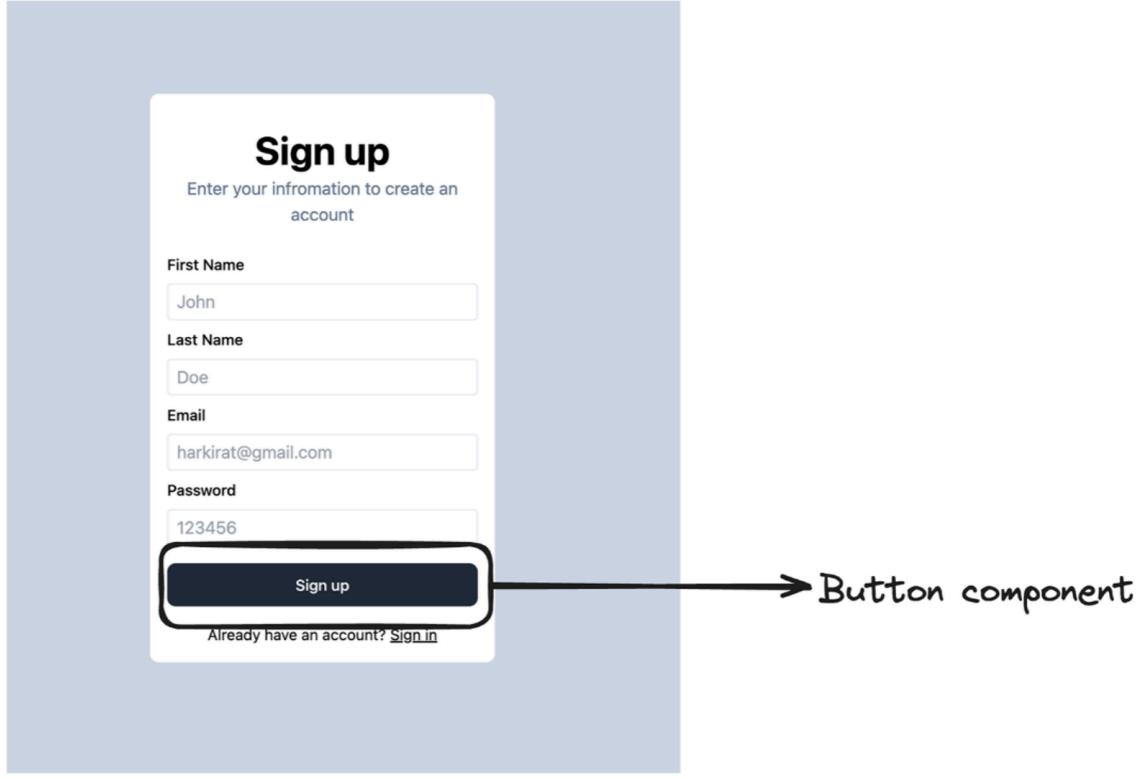


▼ Code

```
export function InputBox({label, placeholder}) {
  return <div>
    <div className="text-sm font-medium text-left py-2">
      {label}
    </div>
    <input placeholder={placeholder} className="w-full px-2 py-1 border rounded border-sla
  </div>
}
```



4. Button Component

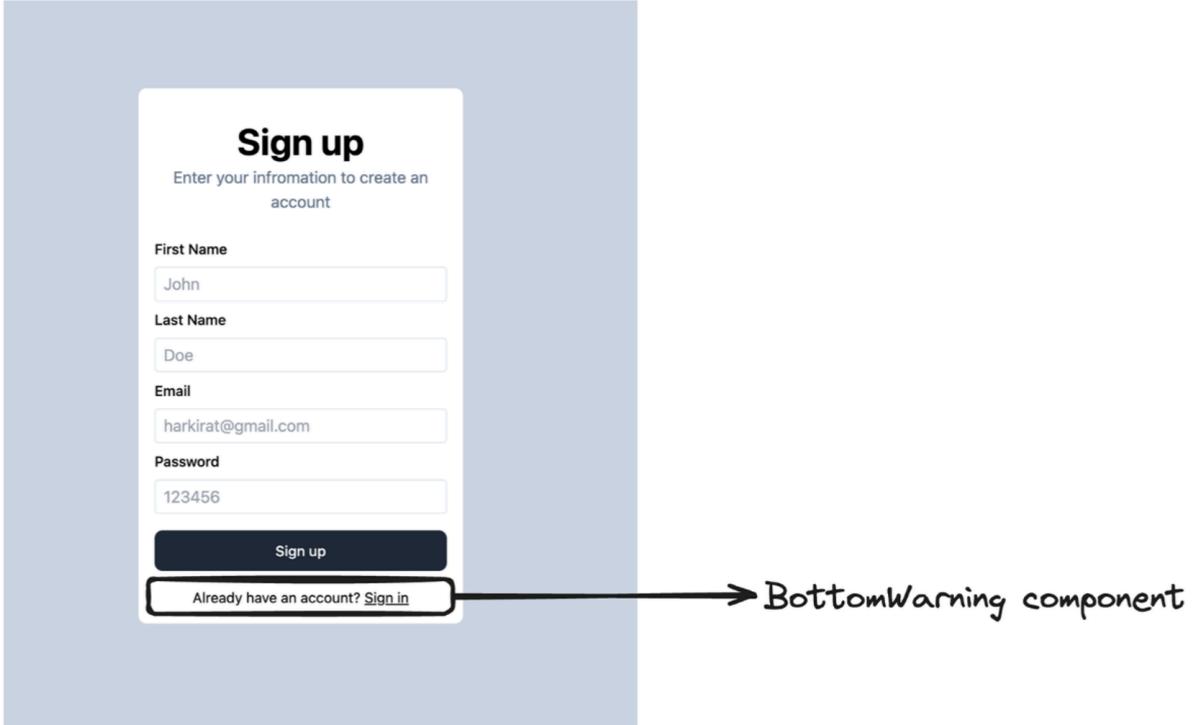


▼ Code

```
export function Button({label, onClick}) {
  return <button onClick={onClick} type="button" class=" w-full text-white bg-gray-800 hov
}
```

This section was blindly copied from <https://flowbite.com/docs/components/buttons/>

5. BottomWarning

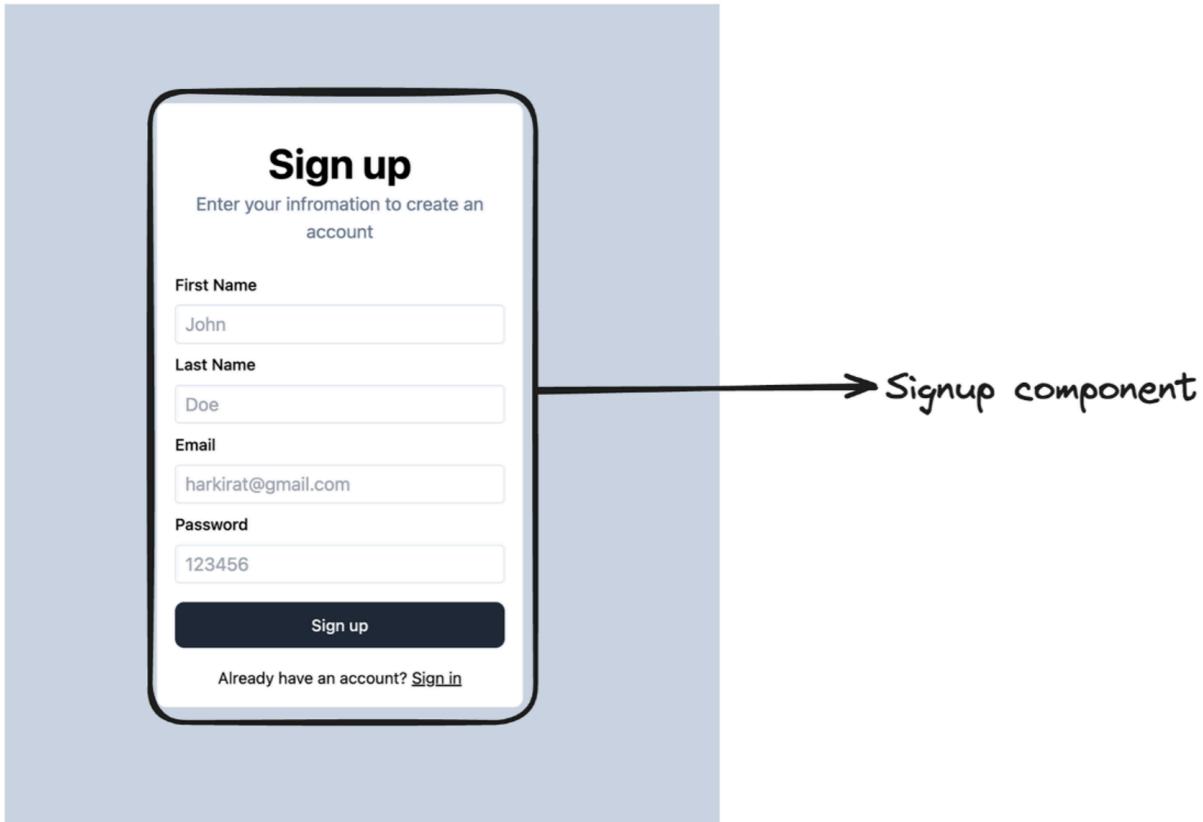


▼ Code

```
import { Link } from "react-router-dom"

export function BottomWarning({label, buttonText, to}) {
  return <div className="py-2 text-sm flex justify-center">
    <div>
      {label}
    </div>
    <Link className="pointer underline pl-1 cursor-pointer" to={to}>
      {buttonText}
    </Link>
  </div>
}
```

Full Signup component



▼ Code

```
import { BottomWarning } from "../components/BottomWarning"
import { Button } from "../components/Button"
import { Heading } from "../components/Heading"
import { InputBox } from "../components/InputBox"
import { SubHeading } from "../components/SubHeading"

export const Signup = () => {
  return <div className="bg-slate-300 h-screen flex justify-center">
    <div className="flex flex-col justify-center">
      <div className="rounded-lg bg-white w-80 text-center p-2 h-max px-4">
        <Heading label={"Sign up"} />
        <SubHeading label={"Enter your information to create an account"} />
        <InputBox placeholder="John" label={"First Name"} />
        <InputBox placeholder="Doe" label={"Last Name"} />
        <InputBox placeholder="harkirat@gmail.com" label={"Email"} />
        <InputBox placeholder="123456" label={"Password"} />
        <div className="pt-4">
          <Button label={"Sign up"} />
        </div>
        <BottomWarning label={"Already have an account?"} buttonText={"Sign in"} to={"/sign-in"} />
      </div>
    </div>
  </div>
```

```
</div>
}
```

Full Signin component

▼ Code

```
import { BottomWarning } from "../components/BottomWarning"
import { Button } from "../components/Button"
import { Heading } from "../components/Heading"
import { InputBox } from "../components/InputBox"
import { SubHeading } from "../components/SubHeading"

export const Signin = () => {

  return <div className="bg-slate-300 h-screen flex justify-center">
    <div className="flex flex-col justify-center">
      <div className="rounded-lg bg-white w-80 text-center p-2 h-max px-4">
        <Heading label={"Sign in"} />
        <SubHeading label={"Enter your credentials to access your account"} />
        <InputBox placeholder="harkirat@gmail.com" label={"Email"} />
        <InputBox placeholder="123456" label={"Password"} />
        <div className="pt-4">
          <Button label={"Sign in"} />
        </div>
        <BottomWarning label={"Don't have an account?"} buttonText={"Sign up"} to={"/signup"} />
      </div>
    </div>
  </div>
}

}
```

Step 6 - Signin-ed Comonents

1. Appbar



▼ Code

```
export const AppBar = () => {
  return <div className="shadow h-14 flex justify-between">
    <div className="flex flex-col justify-center h-full ml-4">
      PayTM App
    </div>
    <div className="flex">
      <div className="flex flex-col justify-center h-full mr-4">
        Hello
      </div>
      <div className="rounded-full h-12 w-12 bg-slate-200 flex justify-center mt-1 mr-2">
        <div className="flex flex-col justify-center h-full text-xl">
          U
        </div>
      </div>
    </div>
  </div>
}
```

2. Balance



▼ Code

```
export const Balance = ({ value }) => {
  return <div className="flex">
    <div className="font-bold text-lg">
      Your balance
    </div>
    <div className="font-semibold ml-4 text-lg">
      Rs {value}
    </div>
  </div>
}
```

3. Users component



▼ Code

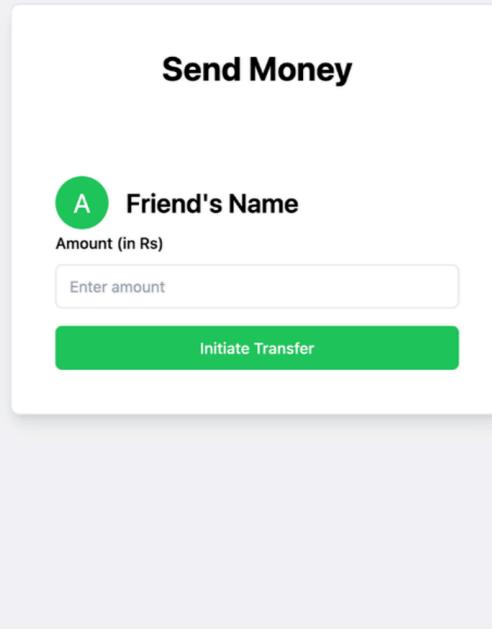
```
import { useState } from "react"
import { Button } from "./Button"

export const Users = () => {
    // Replace with backend call
    const [users, setUsers] = useState([
        {
            firstName: "Harkirat",
            lastName: "Singh",
            _id: 1
        }
    ]);

    return <>
        <div className="font-bold mt-6 text-lg">
            Users
        </div>
        <div className="my-2">
            <input type="text" placeholder="Search users..." className="w-full px-2 py-1 border rounded" />
        </div>
        <div>
            {users.map(user => <User user={user} />)}
        </div>
    </>
}

function User({user}) {
    return <div className="flex justify-between">
        <div className="flex">
            <div className="rounded-full h-12 w-12 bg-slate-200 flex justify-center mt-1 mr-2">
                <div className="flex flex-col justify-center h-full text-xl">
                    {user.firstName[0]}
                </div>
            </div>
            <div className="flex flex-col justify-center h-ful">
                <div>
                    {user.firstName} {user.lastName}
                </div>
            </div>
        </div>
        <div className="flex flex-col justify-center h-ful">
            <Button label={"Send Money"} />
        </div>
    </div>
}
```

4. SendMoney Component



▼ Code

```
export const SendMoney = () => {
  return <div class="flex justify-center h-screen bg-gray-100">
    <div className="h-full flex flex-col justify-center">
      <div
        class="border h-min text-card-foreground max-w-md p-4 space-y-8 w-96 bg-white">
        <div class="flex flex-col space-y-1.5 p-6">
          <h2 class="text-3xl font-bold text-center">Send Money</h2>
          </div>
          <div class="p-6">
            <div class="flex items-center space-x-4">
              <div class="w-12 h-12 rounded-full bg-green-500 flex items-center justify-center">
                <span class="text-2xl text-white">A</span>
              </div>
              <h3 class="text-2xl font-semibold">Friend's Name</h3>
            </div>
            <div class="space-y-4">
              <div class="space-y-2">
                <label
                  class="text-sm font-medium leading-none peer-disabled:cursor-not-allowed"
                  for="amount">
                  Amount (in Rs)
                </label>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
```

```

</label>
<input
    type="number"
    class="flex h-10 w-full rounded-md border border-input bg-background-100 p-2.5 text-sm font-medium ring-offset-1 ring-ring-100 placeholder-transparent"
    id="amount"
    placeholder="Enter amount"
/>
</div>
<button class="justify-center rounded-md text-sm font-medium ring-offset-1 ring-ring-100 py-2.5 px-4.5 text-white transition-colors duration-200 ease-in-out"
        type="button">
    Initiate Transfer
</button>
</div>
</div>
</div>
</div>
</div>
</div>
</div>
}

```

Step 7 - Wiring up the backend calls

You can use

1. fetch or
2. axios

to wire up calls to the backend server.

The final code looks something like this -

<https://github.com/100xdevs-cohort-2/paytm/tree/complete-solution> (complete-solution branch on the repo)

The important bits here are -

1. Signup call - <https://github.com/100xdevs-cohort-2/paytm/blob/complete-solution/frontend/src/pages/Signup.jsx#L36>
2. Call to get all the users given the filter - <https://github.com/100xdevs-cohort-2/paytm/blob/complete-solution/frontend/src/components/Users.jsx#L13>
3. Call to transfer money b/w accounts - <https://github.com/100xdevs-cohort-2/paytm/blob/complete-solution/frontend/src/pages/SendMoney.jsx#L45>



Week 9.1

Custom Hooks

In this lecture, Harkirat presents a comprehensive guide to [Custom Hooks in React](#). The discussion begins by contrasting [class-based](#) and [functional component](#) approaches. It then delves into the rationale behind the advent of custom hooks, highlighting their role in maintaining cleaner code. The session concludes with [hands-on exploration](#) of diverse examples, providing practical insights into effectively [implementing custom hooks](#).

Note: Today's lecture notes are comprehensive, surpassing the depth of previous lectures. Consider them an all-inclusive resource for a thorough understanding and effective utilization of Custom Hooks in your React applications.

Custom Hooks

A Few Concepts Before We Begin

- [1\] Ternary Operator](#)
- [2\] Lifecycle Events](#)
- [3\] Understanding Debouncing](#)

Class Components vs Functional Components

State Management

- [Using Class Component](#)
- [Using Functional Component](#)

Lifecycle Events

Using Class Component

Using Functional Component

Significance of Returning a Component from useEffect

React Hooks

Custom Hooks

What?

Why?

How?

Examples

Use Cases of Custom Hooks

1] Data Fetching Hooks

Step 1 - Converting the data fetching bit to a custom hook

Step 2 - Cleaning the hook to include a loading parameter

Step 3 - Auto refreshing hook

Step 4 - We Clear the Interval

SWR Library

2] Browser Functionality Related Hooks

useOnlineStatus

use.mousePosition

3] Performance/Timer Based

useInterval

useDebounce

A Few Concepts Before We Begin

1] Ternary Operator

The ternary operator, also known as the conditional operator, is a concise way to write an `if-else` statement in a single line. It has the following syntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

Here's how it works:

- The `condition` is evaluated. If it is `true`, the expression before the `:` (colon) is executed; otherwise, the expression after the `:` is executed.

Let's look at a simple example:

```
const isRaining = true;

const weatherMessage = isRaining ? "Bring an umbrella" : "Enjoy the sunshine";

console.log(weatherMessage);
```

In this example, if `isRaining` is `true`, the `weatherMessage` will be set to "Bring an umbrella"; otherwise, it will be set to "Enjoy the sunshine."

The ternary operator is often used for simple conditional assignments, making the code more concise. However, it's important to use it judiciously, as overly complex ternary expressions can reduce code readability.

2] Lifecycle Events

Lifecycle events in React represent various phases a component goes through from its birth to its removal from the DOM. These events are associated with class components and provide developers with the ability to execute code at specific points during a component's existence.

It is completely okay if you find this section challenging at the moment, as we delve into a more in-depth discussion in the latter part of today's notes. For now, aim to grasp an overview of the concepts introduced and the associated terminology.

The key lifecycle events in a class-based React component are:

1. `componentDidMount` : This method is called after a component has been rendered to the DOM. It is commonly used to perform initial setup, data fetching, or subscriptions.

```
componentDidMount() {
  // Perform setup or data fetching here
}
```

1. **componentDidUpdate** : This method is invoked immediately after an update occurs. It's useful for reacting to prop or state changes and performing additional actions.

```
componentDidUpdate(prevProps, prevState) {  
  // Perform actions based on prop or state changes  
}
```

1. **componentWillUnmount** : This method is called just before a component is removed from the DOM. It's suitable for cleanup tasks, such as removing event listeners or canceling subscriptions.

```
componentWillUnmount() {  
  // Clean up (e.g., remove event listeners or cancel subscriptions)  
}
```

With the introduction of React Hooks, functional components also gained lifecycle-like behavior through the **useEffect** hook. The equivalent hooks are:

1. **useEffect** with an empty dependency array: Equivalent to **componentDidMount** . Runs after the initial render.

```
useEffect(() => {  
  // Code to run after initial render  
}, []);
```

1. **useEffect** with dependencies: Equivalent to **componentDidUpdate** . Runs whenever the specified dependencies change.

```
useEffect(() => {  
  // Code to run when dependencies change  
}, [dependency1, dependency2]);
```

1. **useEffect** with a cleanup function: Equivalent to **componentWillUnmount** . Runs before the component is unmounted.

```
useEffect(() => {  
  // Code to run on component mount
```

```
return () => {
  // Cleanup code (similar to componentWillUnmount)
};
}, []);
```

These lifecycle events are crucial for managing side effects, updating UI in response to changes, and maintaining clean-up procedures for optimal application performance.

Well as emphasized before it is completely okay if you found the above section challenging at the moment, as we delve into a more in-depth discussion in the latter part of today's notes. For now, aim to grasp an overview of the concepts introduced and the associated terminology.

3] Understanding Debouncing

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, making them more efficient. In the context of `onInput` events, debouncing is often applied to delay the execution of certain actions (e.g., sending requests) until after a user has stopped typing for a specific duration.

Implementation:

The following example demonstrates debouncing in the `onInput` event to delay the execution of a function that sends a request based on user input.

```
<html>
  <body>
    <!-- Input field with onInput event and debouncing -->
    <input id="textInput" type="text" onInput="debounce(handleInput, 500)" placeholder="Type something..." />

    <!-- Display area for the debounced input value -->
    <p id="displayText"></p>

    <script>
      // Debounce function to delay the execution of a function
      function debounce(func, delay) {
        let timeoutId;

        return function() {
          // Clear the previous timeout
          clearTimeout(timeoutId);

```

```
// Set a new timeout
timeoutId = setTimeout(() => {
  func.apply(this, arguments);
}, delay);
};

// Function to handle the debounced onInput event
function handleInput() {
  // Get the input field's value
  const inputValue = document.getElementById("textInput").value;

  // Display the input value in the paragraph
  document.getElementById("displayText").innerText = "You typed: " + inputValue;

  // Simulate sending a request (replace with actual AJAX call)
  console.log("Request sent:", inputValue);
}

</script>
</body>
</html>
```

Explanation:

- The `debounce` function is a generic debounce implementation that takes a function (`func`) and a delay time (`delay`).
- Inside the `debounce` function, a timeout is set to delay the execution of the provided function (`func`) by the specified delay time (`delay`).
- The `handleInput` function is the actual function to be executed when the `onInput` event occurs. It simulates sending a request (e.g., an AJAX call) based on user input.

How it works:

- When a user types in the input field, the `onInput` event triggers the `debounce` function.
- The `debounce` function sets a timeout, and if the user continues typing within the specified delay
- After the user stops typing for the specified delay, the `handleInput` function is executed.

This ensures that the function associated with the `onInput` event is not called on every keystroke but rather after the user has stopped typing for a brief moment, reducing unnecessary and potentially resource-intensive calls, such as sending requests.

Class Components vs Functional Components

State Management

Let us take a look at the implementation of a simple counter component using both functional and class-based approaches in React, emphasizing state management.

Using Class Component

```
import React from 'react';

class MyComponent extends React.Component {
  // Constructor to initialize state
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  // Method to increment the count when the button is clicked
  incrementCount = () => {
    // Updating the count state using this.setState
    this.setState({ count: this.state.count + 1 });
  }

  // Render method to display the current count and the "Increment" button
  render() {
    return (
      <div>
        <p>{this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

1. In the class-based component, state is initialized in the constructor using `this.state`.
2. `this.state = { count: 0 };` initializes the `count` state variable with an initial value of 0.
3. The `incrementCount` method increases the `count` state by 1 when the "Increment" button is clicked.

4. The `render` method displays the current count and a button that triggers the `incrementCount` method.

Using Functional Component

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

function MyComponent() {
  // Using the useState hook to manage state in a functional component
  const [count, setCount] = useState(0);

  // Function to increment the count when the button is clicked
  const incrementCount = () => {
    // Updating the count state using the setCount function
    setCount(count + 1);
  };

  // Rendering the component with the current count and an "Increment" button
  return (
    <div>
      <p>{count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

1. The `useState` hook is used to declare state variables within functional components.
2. `const [count, setCount] = useState(0);` initializes the `count` state variable with an initial value of 0, and `setCount` is a function used to update the `count` state.
3. The `incrementCount` function increases the `count` state by 1 each time the "Increment" button is clicked.
4. The JSX returned by the component displays the current count and a button that triggers the `incrementCount` function.

Both implementations achieve the same result, showcasing different approaches to managing state in React components. The functional component uses the `useState` hook for a more concise and modern syntax, while the class-based

component follows the traditional class syntax for state management. The choice between them often depends on personal preference and the specific requirements of the application.

Lifecycle Events

Now, let's explore the implementation of how lifecycle events are handled in both class-based and functional components in React.

Using Class Component

```
import React from 'react';

class MyComponent extends React.Component {
  // componentDidMount: Invoked after the component is first mounted to the DOM
  componentDidMount() {
    // Perform setup or data fetching here
    console.log('Component is mounted to the DOM');
  }

  // componentWillUnmount: Invoked just before the component is unmounted and destroyed
  componentWillUnmount() {
    // Clean up (e.g., remove event listeners or cancel subscriptions)
    console.log('Component is about to be unmounted');
  }

  // render: Renders the UI of the component
  render() {
    return (
      // Render UI
      <div>
        Component Lifecycle Events (Class-Based)
      </div>
    );
  }
}
```

1. **componentDidMount** : Invoked after the component is first mounted to the DOM. It's a suitable place for initial setup or data fetching operations.

2. `componentWillUnmount` : Invoked just before the component is unmounted and destroyed. It's used for cleanup tasks, such as removing event listeners or canceling subscriptions.
3. `render` : The method responsible for rendering the UI of the component.

Using Functional Component

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

function MyComponent() {
  // useEffect: Invoked after the component is mounted and reinvoked if dependencies change
  useEffect(() => {
    // Perform setup or data fetching here

    // Cleanup function (similar to componentWillUnmount)
    return () => {
      console.log('Component is about to be unmounted (cleanup)');
      // Cleanup code goes here
    };
  }, []); // Empty dependency array ensures useEffect runs only on mount and unmount

  // Render UI
  return (
    <div>
      Component Lifecycle Events (Functional)
    </div>
  );
}
```

1. `useEffect` : Invoked after the component is mounted and reinvoked if dependencies change. It can be used for both setup and cleanup.
2. The empty dependency array (`[]`) ensures that the `useEffect` runs only on mount and unmount, simulating the behavior of `componentDidMount` and `componentWillUnmount` .
3. The cleanup function within `useEffect` is invoked just before the component is unmounted.

In functional components, the `useEffect` hook is a versatile replacement for various lifecycle events in class-based components. It handles both mounting and unmounting scenarios, offering a more concise and expressive way to manage side effects.

Significance of Returning a Component from useEffect

In the provided code snippet, we utilize the `useEffect` hook along with the `setInterval` function to toggle the state of the `render` variable every 5 seconds. This, in turn, controls the rendering of the `MyComponent` or an empty `div` based on the value of `render`. Let's break down the significance of returning a component from `useEffect`:

```
import React, { useEffect, useState } from 'react';
import './App.css';

function App() {
  const [render, setRender] = useState(true);

  useEffect(() => {
    // Toggle the state every 5 seconds
    const intervalId = setInterval(() => {
      setRender(r => !r);
    }, 5000);

    // Cleanup function: Clear the interval when the component is unmounted
    return () => {
      clearInterval(intervalId);
    };
  }, []);

  return (
    <>
      {render ? <MyComponent /> : <div></div>}
    </>
  );
}

function MyComponent() {
  useEffect(() => {
    console.error("Component mounted");

    // Cleanup function: Log when the component is unmounted
    return () => {
      console.log("Component unmounted");
    };
  }, []);
}
```

```
return <div>
  From inside MyComponent
</div>;
}

export default App;
```

Understanding the Code

- The `useEffect` hook is used to create a side effect (in this case, toggling the `render` state at intervals) when the component mounts.
- A cleanup function is returned within the `useEffect`, which will be executed when the component is unmounted. In this example, it clears the interval previously set by `setInterval`.
- By toggling the `render` state, the component (`MyComponent` or an empty `div`) is conditionally rendered or unrendered, demonstrating the dynamic nature of component rendering.
- The `return` statement within the `useEffect` of `MyComponent` is used to specify what should be rendered when the component is active, in this case, a simple `div` with the text "From inside MyComponent."

In summary, the ability to return a cleanup function from `useEffect` is crucial for managing resources, subscriptions, or intervals created during the component's lifecycle. It helps ensure proper cleanup when the component is no longer in use, preventing memory leaks or unintended behavior.

React Hooks

React Hooks are functions that allow functional components in React to have state and lifecycle features that were previously available only in class components. Hooks were introduced in React 16.8 to enable developers to use state and other React features without writing a class.

Using these hooks, developers can manage state, handle side effects, optimize performance, and create more reusable and readable functional components in React applications. Each hook serves a specific purpose, contributing to a more modular and maintainable codebase.

In previous lectures, specifically Week 6 —we have already covered in depth the most commonly used hooks provided to us by React: `useEffect` , `useMemo` , `useCallback` , `useRef` , `useReducer` , `useContext` , `useLayoutEffect`

Custom Hooks

What?

Custom Hooks in React are user-defined functions that encapsulate reusable logic and stateful behavior. They allow developers to extract and share common functionality across multiple components, promoting code reusability and maintaining cleaner and more modular code.

Why?

The need for custom hooks arises from the desire to avoid code duplication and create a clean separation of concerns. By encapsulating specific logic in a custom hook, you can isolate and organize the functionality related to a particular concern or feature. This not only makes your codebase more maintainable but also facilitates easier testing and debugging.

How?

Custom hooks solve the problem of sharing logic between components without the need for higher-order components or render props. They provide a mechanism to encapsulate complex behavior, making it easier to reason about and reuse across different parts of your application. Ultimately, custom hooks contribute to a more efficient and scalable React codebase.

Examples

1. Data fetching hooks
2. Browser functionality related hooks - `useOnlineStatus` , `useWindowSize`, `useMousePosition`
3. Performance/Timer based - `useInterval`, `useDebounce`

Use Cases of Custom Hooks

1] Data Fetching Hooks

Data fetching hooks can be used to encapsulate all the logic to fetch the data from your backend

Now below is how our code looks before using custom hooks

```
import { useEffect, useState } from 'react'
import axios from 'axios'

function App() {
  const [todos, setTodos] = useState([])

  useEffect(() => {
    axios.get("https://sum-server.100xdevs.com/todos")
      .then(res => {
        setTodos(res.data.todos);
      })
  }, [])

  return (
    <>
      {todos.map(todo => <Track todo={todo} />)}
    </>
  )
}

function Track({ todo }) {
  return <div>
    {todo.title}
    <br />
    {todo.description}
  </div>
}

export default App
```

We will now see a step by step guide on how you can use custom hook for the above use case.

Step 1 - Converting the **data fetching** bit to a custom hook

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

// Custom hook for fetching todos
function useTodos() {
  const [todos, setTodos] = useState([]);
```

```

useEffect(() => {
  // Fetching todos using Axios
  axios.get("<https://sum-server.100xdevs.com/todos>")
    .then(res => {
      setTodos(res.data.todos);
    })
}, []);

// Return the todos state
return todos;
}

// Main App component
function App() {
  // Using the custom hook to fetch todos
  const todos = useTodos();

  return (
    <>
    {/* Rendering Track component for each todo */}
    {todos.map(todo => <Track key={todo.id} todo={todo} />)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;

```

Explanation:

1. Custom Hook (`useTodos`):

- It encapsulates the data fetching logic using `axios` within a custom hook.
- Utilizes the `useState` and `useEffect` hooks to manage state and perform side effects respectively.
- Returns the `todos` state, making it accessible in the component that uses this custom hook.

2. Main App Component:

- Imports and uses the `useTodos` custom hook, fetching todos and storing them in the `todos` variable.
- Maps over the `todos` array, rendering the `Track` component for each todo.

3. Track Component:

- Receives an individual `todo` as a prop and renders its `title` and `description`.

By creating a custom hook (`useTodos`), the data fetching logic is abstracted and can be easily reused across different components. This promotes a cleaner and more modular code structure.

Step 2 - Cleaning the hook to include a `loading` parameter

What if you want to show a loader when the data is not yet fetched from the backend?

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

// Custom hook for fetching todos with loading indicator
function useTodos() {
  const [loading, setLoading] = useState(true);
  const [todos, setTodos] = useState([]);

  useEffect(() => {
    // Fetching todos using Axios
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching todos:", error);
        setLoading(false);
      });
  }, []);

  // Return todos and loading state
  return {
    todos: todos,
    loading: loading
  };
}
```

```

}

// Main App component
function App() {
  // Using the custom hook to fetch todos
  const { todos, loading } = useTodos();

  // Rendering loading message if data is still loading
  if (loading) {
    return <div>Loading...</div>;
  }

  // Rendering Track component for each todo
  return (
    <>
      {todos.map(todo => <Track key={todo.id} todo={todo} />)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;

```

Explanation:

1. Custom Hook (`useTodos`):

- Introduces a `loading` state to track whether the data is still being fetched.
- The `setLoading(false)` is placed in both the successful and error scenarios to handle loading completion.
- Returns an object with both `todos` and `loading` states.

2. Main App Component:

- Destructures the result from the custom hook, including `todos` and `loading`.
- If `loading` is `true`, it renders a loading message. Otherwise, it maps over the `todos` array and renders the `Track` component.

3. Track Component:

- Remains the same, rendering individual todos.

By including a loading parameter in the custom hook, you can provide better user experience by displaying a loading message while the data is being fetched.

Step 3 - Auto refreshing hook

What if you want to keep polling the backend every n seconds? `n` needs to be passed in as an input to the hook

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

// Custom hook for fetching todos with auto-refresh
function useTodos(n) {
  const [loading, setLoading] = useState(true);
  const [todos, setTodos] = useState([]);

  // Function to fetch data from the backend
  function getData() {
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching todos:", error);
        setLoading(false);
      });
  }

  useEffect(() => {
    // Initial data fetch
    getData();

    // Set up interval to fetch data every n seconds
    const intervalId = setInterval(() => {
      getData();
    }, n * 1000);

    // Clean up the interval on component unmount or when n changes
    return () => clearInterval(intervalId);
  }, [n]);
}
```

```
// Return todos and loading state
return {
  todos: todos,
  loading: loading
};

// Main App component
function App() {
  // Using the custom hook to fetch todos with auto-refresh every 5 seconds
  const { todos, loading } = useTodos(5);

  // Rendering loading message if data is still loading
  if (loading) {
    return <div>Loading...</div>;
  }

  // Rendering Track component for each todo
  return (
    <>
      {todos.map(todo => <Track key={todo.id} todo={todo} />)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;
```

Explanation:

1. Custom Hook (`useTodos`):

- Accepts an input `n` representing the interval in seconds for auto-refresh.
- Utilizes the `getData` function to fetch data from the backend.
- In the `useEffect`, sets up an interval to call `getData` every `n` seconds.
- Cleans up the interval when the component unmounts or when `n` changes.

2. Main App Component:

- Utilizes the custom hook with an interval of 5 seconds (`useTodos(5)`).
- Renders a loading message if data is still loading and maps over `todos` otherwise.

3. Track Component:

- Remains the same, rendering individual todos.

This step enhances the hook by adding an auto-refresh feature, ensuring the data is periodically fetched from the backend.

Step 4 - We Clear the Interval

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

// Custom hook for fetching todos with auto-refresh
function useTodos(n) {
  const [todos, setTodos] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Set up interval to fetch data every n seconds
    const intervalId = setInterval(() => {
      axios.get("<https://sum-server.100xdevs.com/todos>")
        .then(res => {
          setTodos(res.data.todos);
          setLoading(false);
        })
        .catch(error => {
          console.error("Error fetching todos:", error);
          setLoading(false);
        });
    }, n * 1000);

    // Initial data fetch
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching todos:", error);
        setLoading(false);
      });
  }, []);
}
```

```

// Clean up the interval on component unmount or when n changes
return () => clearInterval(intervalId);
}, [n]);

// Return todos and loading state
return { todos, loading };
}

// Main App component
function App() {
  // Using the custom hook to fetch todos with auto-refresh every 10 seconds
  const { todos, loading } = useTodos(10);

  // Rendering loading message if data is still loading
  if (loading) {
    return <div>Loading...</div>;
  }

  // Rendering Track component for each todo
  return (
    <>
      {todos.map(todo => <Track key={todo.id} todo={todo} />)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;

```

Explanation:

1. Custom Hook (`useTodos`):

- Continues to have the auto-refresh functionality with a 10-second interval (`useTodos(10)`).
- Clears the interval using `clearInterval` in the cleanup function returned by `useEffect`. This ensures that the interval is cleared when the component unmounts or when `n` changes.

2. Main App Component:

- Remains the same, utilizing the custom hook and rendering todos.

3. Track Component:

- Unchanged, rendering individual todos.

This step enhances the hook by adding a cleanup mechanism to clear the interval when it's no longer needed, preventing potential memory leaks.

SWR Library

The `swr` library is a powerful tool for data fetching in React applications. It simplifies the process of handling data fetching, caching, and re-fetching when needed. Here's an explanation of the provided code snippet:

```
// Import the useSWR hook from the 'swr' library
import useSWR from 'swr';

// Define a fetcher function to handle data fetching
const fetcher = async function(url) {
  // Fetch data from the specified URL
  const data = await fetch(url);

  // Parse the response as JSON
  const json = await data.json();

  // Return the parsed JSON data
  return json;
};

// Example component using the useSWR hook
function Profile() {
  // Use the useSWR hook to fetch data from the specified URL
  const { data, error, isLoading } = useSWR('<https://sum-server.100xdevs.com/todos>', fetcher

  // Handle different states: loading, error, and successful data fetch
  if (error) return <div>Failed to load</div>;
  if (isLoading) return <div>Loading...</div>

  // Render the component with the fetched data
  return <div>Hello, you have {data.todos.length} todos!</div>;
}
```

Explanation:

1. Importing `useSWR`:

- The `useSWR` hook is imported from the 'swr' library. This hook simplifies data fetching by providing caching and re-fetching capabilities.

2. Fetcher Function:

- A `fetcher` function is defined to handle data fetching. It uses the `fetch` API to retrieve data from the specified URL, parses the response as JSON, and returns the parsed data.

3. Usage in `Profile` Component:

- The `useSWR` hook is used in the `Profile` component to fetch data from the specified URL (<https://sum-server.100xdevs.com/todos>). The `fetcher` function is provided as the second argument to `useSWR`.

4. Handling Different States:

- The component checks for different states: `error`, `isLoading`, and successful data fetch. Depending on the state, it renders appropriate content (error message, loading indicator, or the fetched data).

5. Rendering Component:

- If the data is successfully fetched, the component renders a message indicating the number of todos.

Using `swr` can significantly simplify data fetching in React applications, providing a clean and efficient way to manage remote data.

<https://swr.vercel.app/>

2] Browser Functionality Related Hooks

`useOnlineStatus`

The Custom React Hook — `useIsOnline` determines whether the user is currently online or offline. It utilizes the `window.navigator.onLine` property and event listeners to keep track of the online status. Here's a detailed explanation:

1. `useIsOnline` Hook:

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

function useIsOnline() {
  // Initialize state with the current online status
  const [isOnline, setIsOnline] = useState(window.navigator.onLine);

  useEffect(() => {
    // Add event listeners to track online/offline changes
    const handleOnline = () => setIsOnline(true);
    const handleOffline = () => setIsOnline(false);

    // Attach event listeners to the 'online' and 'offline' events
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);

    // Cleanup: Remove event listeners on component unmount
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);

  // Return the current online status
  return isOnline;
}
```

Explanation:

- **Initialization:** The `isOnline` state variable is initialized with the current value of `window.navigator.onLine`. This represents the initial online status.
- **Effect Hook:** The `useEffect` hook is used to add event listeners for the 'online' and 'offline' events when the component mounts. These listeners update the `isOnline` state accordingly.
- **Event Listeners:** Two event listeners, `handleOnline` and `handleOffline`, are defined to update the `isOnline` state based on the user's online or offline status.
- **Cleanup:** The `useEffect` hook also returns a cleanup function. This function removes the event listeners when the component is unmounted, preventing memory leaks.

2. `App` Component:

```
function App() {
  // Use the custom hook to get the current online status
  const isOnline = useIsOnline();

  // Render different messages based on the online status
```

```

return (
  <>
  {isOnline ? "You are online, yay!" : "You are not online"}
</>
);
}

export default App;

```

Explanation:

- The `App` component uses the `useIsOnline` hook to determine the current online status.
- Based on the online status, it renders different messages to inform the user whether they are online or offline.

This custom hook provides a reusable way to track the user's online status throughout the lifecycle of a React component.

useMousePosition

The Custom React hook — `useMousePointer` allows tracking the current position of the mouse pointer. It utilizes the `window.addEventListener` method with the 'mousemove' event to update the mouse position. Here's a detailed explanation:

1. `useMousePointer` Hook:

```

import { useEffect, useState } from 'react';

const useMousePointer = () => {
  // Initialize state with the initial mouse position (0, 0)
  const [position, setPosition] = useState({ x: 0, y: 0 });

  // Event handler to update the mouse position on mouse movement
  const handleMouseMove = (e) => {
    setPosition({ x: e.clientX, y: e.clientY });
  };

  useEffect(() => {
    // Add event listener for 'mousemove' event when the component mounts
    window.addEventListener('mousemove', handleMouseMove);

    // Cleanup: Remove event listener on component unmount
  });
}

```

```

    return () => {
      window.removeEventListener('mousemove', handleMouseMove);
    };
  }, []);
}

// Return the current mouse position
return position;
};

```

Explanation:

- **Initialization:** The `position` state variable is initialized with the initial mouse position (`{ x: 0, y: 0 }`).
- **Event Handler:** The `handleMouseMove` function is defined to update the `position` state with the current mouse coordinates (`e.clientX` and `e.clientY`) when the mouse is moved.
- **Effect Hook:** The `useEffect` hook is used to add an event listener for the 'mousemove' event when the component mounts. This listener triggers the `handleMouseMove` function on mouse movement.
- **Cleanup:** The `useEffect` hook returns a cleanup function that removes the 'mousemove' event listener when the component is unmounted, preventing memory leaks.

2. App Component:

```

function App() {
  // Use the custom hook to get the current mouse position
  const mousePointer = useMousePointer();

  // Render a message displaying the current mouse position
  return (
    <>
      Your mouse position is {mousePointer.x} {mousePointer.y}
    </>
  );
}

export default App;

```

Explanation:

- The `App` component utilizes the `useMousePointer` hook to obtain the current mouse position.

- It renders a message displaying the x and y coordinates of the mouse position in real-time.

This custom hook provides an easy way to track and utilize the mouse pointer position within a React component.

3] Performance/Timer Based

useInterval

The Custom React Hook — `useInterval` facilitates running a callback function at specified intervals. This hook is then utilized in the `App` component to increment a timer every second. Here's an in-depth explanation:

1. `useInterval` Hook:

```
import { useEffect } from 'react';

const useInterval = (callback, delay) => {
  useEffect(() => {
    // Set up an interval and store the interval ID
    const intervalId = setInterval(callback, delay);

    // Cleanup: Clear the interval when the component is unmounted
    return () => clearInterval(intervalId);
  }, [callback, delay]);
};
```

Explanation:

- **Function Signature:** The `useInterval` hook takes two parameters - `callback` (the function to be executed) and `delay` (the interval in milliseconds).
- **Effect Hook:** Inside the `useEffect` hook, `setInterval` is used to repeatedly call the `callback` function at the specified `delay`.
- **Cleanup:** The returned cleanup function ensures that the interval is cleared when the component using this hook is unmounted, preventing memory leaks.

2. `App` Component:

```
import { useState } from 'react';
import useInterval from './useInterval'; // Import the custom useInterval hook

function App() {
  // State to store the count value
  const [count, setCount] = useState(0);

  // Utilize the useInterval hook to increment the count every second
  useInterval(() => {
    setCount(c => c + 1);
  }, 1000);

  // Render the current count value
  return (
    <>
      Timer is at {count}
    </>
  );
}

export default App;
```

Explanation:

- The `App` component utilizes the `useInterval` hook to increment the `count` state value every second.
- The rendered output displays the current value of the timer, which increases every second.

This custom hook simplifies the implementation of interval-based functionality in React components, providing a reusable and clean solution.

useDebounce

The Custom React Hook — `useDebounce` is utilized in a `SearchBar` component to debounce the user input, making it ideal for scenarios such as live search functionality. Below is a detailed explanation:

1. `useDebounce` Hook:

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

const useDebounce = (value, delay) => {
  // State to store the debounced value
```

```

const [debouncedValue, setDebouncedValue] = useState(value);

useEffect(() => {
  // Set up a timer to update the debounced value after the specified delay
  const timerId = setTimeout(() => {
    setDebouncedValue(value);
  }, delay);

  // Clean up the timer if the value changes before the delay has passed
  return () => clearTimeout(timerId);
}, [value, delay]);

return debouncedValue;
};

```

Explanation:

- **Function Signature:** The `useDebounce` hook takes two parameters - `value` (the input value to be debounced) and `delay` (the debounce delay in milliseconds).
- **State:** The `debouncedValue` state holds the debounced value.
- **Effect Hook:** Inside the `useEffect` hook, a timer is set using `setTimeout`. This timer updates the `debouncedValue` with the current input value after the specified delay.
- **Cleanup:** The `clearTimeout` function is used for cleanup to ensure that the timer is cleared if the input value changes before the delay has passed.
- **Dependencies:** The effect hook depends on the `value` and `delay` parameters, ensuring the effect is re-run when they change.

2. SearchBar Component:

```

import React, { useState } from 'react';
import useDebounce from './useDebounce';

const SearchBar = () => {
  // State to manage the user input
  const [inputValue, setInputValue] = useState('');

  // Use the useDebounce hook to get the debounced value
  const debouncedValue = useDebounce(inputValue, 500); // 500 milliseconds debounce delay

  // Integrate the debouncedValue in your component logic (e.g., trigger a search API call via

  return (
    <input
      type="text"

```

```
value={inputValue}
onChange={(e) => setInputValue(e.target.value)}
placeholder="Search..."/>
);
};

export default SearchBar;
```

Explanation:

- The `SearchBar` component uses the `useDebounce` hook to get the debounced value of the user input (`inputValue`).
- The `onChange` handler updates the `inputValue` state as the user types.
- The debounced value (`debouncedValue`) can be further integrated into the component logic, such as triggering a search API call via a `useEffect`.

This custom hook allows for efficient handling of debounced values, reducing user input changes.



Week 9.2

Introduction to Typescript

In this lecture, Harkirat offers a brief introduction to [TypeScript](#), covering language classifications, the importance of strong typing, and an overview of TypeScript's execution. The lecture includes insights into the TypeScript compiler, implementing basic types, and understanding the distinctions between [Interfaces](#) and [Types](#)

[Introduction to Typescript](#)

[Types of Languages](#)

[1\] Loosely Typed Languages](#)

[C++ Code \(Doesn't Work \)](#)

[2\] Strongly Typed Languages](#)

[JavaScript Code \(Does Work \)](#)

[Typescript](#)

[Why Typescript](#)

[What Typescript](#)

[How Typescript](#)

[Execution of TypeScript Code](#)

[TypeScript Compiler \(tsc\)](#)

[Setting up a Typescript Nodejs Application](#)

Basic Types in Typescript

Problems and Code Implementation

[1\] Hello World Greeting](#)

[2\] Sum Function](#)

[3\] Age Verification Function](#)

[4\] Delayed Function Execution](#)

The tsconfig.json File in TypeScript

[1\] Target Option in tsconfig.json:](#)

[2\] rootDir:](#)

[3\] outDir](#)

[4\] noImplicitAny](#)

[5\] removeComments](#)

Interfaces

Understanding Interfaces

[Assignment 1](#)

[Assignment 2](#)

Implementing Interfaces

Types

Features

Interfaces vs Types

Major Differences

[1. Declaration Syntax:](#)

[2. Extension and Merging:](#)

[3. Declaration vs. Implementation:](#)

Other Differences

When to Use Which

Examples

Types of Languages

1] Loosely Typed Languages

1. **Runtime Type Association:** Data types are associated with values at runtime. Unlike strongly typed languages, type information is not strictly bound during compilation but rather at the time of execution.
2. **Dynamic Type Changes:** Variables can change types during execution, offering more adaptability. This flexibility allows for a dynamic approach to variable assignments and operations.
3. **Runtime Error Discovery:** Type errors may be discovered during runtime, potentially leading to unexpected behaviors. This characteristic provides more freedom but requires careful handling.
4. **Examples of Loosely Typed Languages:** JavaScript, Python, Ruby

C++ Code (Doesn't Work ✗)

```
#include <iostream>

int main() {
    int number = 10;
    number = "text"; // Error: Cannot assign a string to an integer variable
    return 0;
}
```

Explanation:

- C++ is a statically-typed language, meaning variable types must be declared and are enforced at compile-time.
- In the given code, `number` is declared as an integer (`int`), and attempting to assign a string ("text") to it results in a compile-time error.
- The type mismatch between the declared type and the assigned value leads to a compilation failure.

2] Strongly Typed Languages

1. **Compile-Time Enforcement:** The data type of a variable is strictly enforced during compilation. This means that the compiler checks and ensures that variables are used in a way that is consistent with their types at compile time.
2. **Type Safety:** The compiler or interpreter guarantees that operations are performed only on compatible types. This ensures that type-related errors are caught early in the development process.

3. Early Error Detection: Type errors are identified and addressed at compile-time, providing early feedback to developers. This leads to increased reliability and reduces the likelihood of runtime errors.

4. Examples of Strongly Typed Languages: Java, C#, TypeScript

JavaScript Code (Does Work ✅)

```
function main() {  
    let number = 10;  
    number = "text"; // Valid: JavaScript allows dynamic typing  
    return number;  
}
```

Explanation:

- JavaScript is a dynamically-typed language, allowing variables to change types during runtime.
- In the provided JavaScript code, `number` is initially assigned the value `10` (a number), and later, it is assigned the value `"text"` (a string).
- JavaScript allows this flexibility, and the code executes without type-related errors.

Considerations:

- Statically-typed languages like C++ provide early error detection during compilation, ensuring type consistency.
- Dynamically-typed languages like JavaScript offer flexibility but may require careful handling to avoid unexpected runtime errors.

The choice between strongly typed and loosely typed languages depends on project requirements, developer preferences, and the balance between early error detection and flexibility during development. Each type has its advantages and considerations, influencing their suitability for specific use cases.

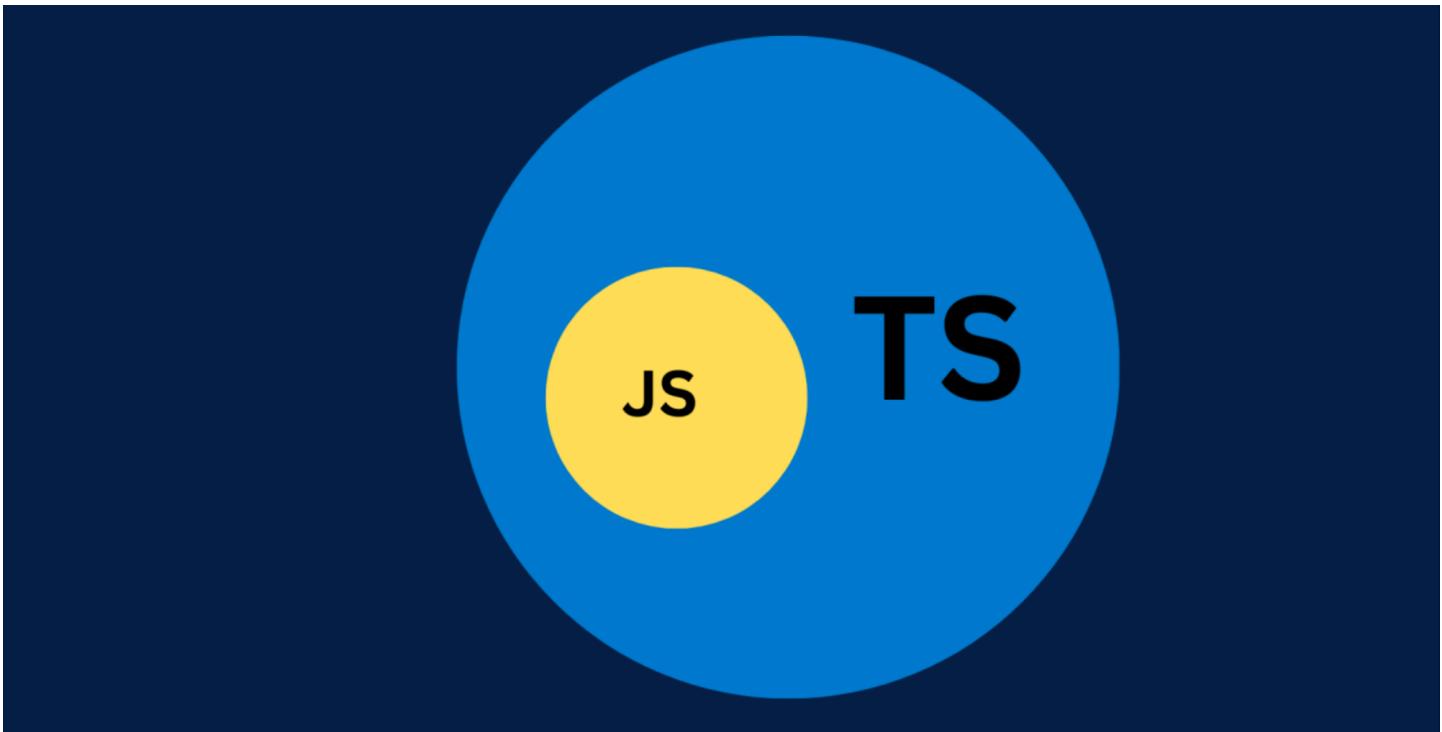
TypeScript

Why TypeScript

JavaScript is a powerful and widely used programming language, but it has a dynamic typing system, which means variable types are determined at runtime. While dynamic typing provides flexibility, it can lead to runtime errors that are challenging to catch during development.

What Typescript

In response to these challenges, Microsoft introduced TypeScript, a superset of JavaScript that adds static typing to the language. TypeScript is designed to address some of the limitations of JavaScript by providing developers with a more robust type system.



How Typescript

1. Static Typing:

- TypeScript introduces static typing, allowing developers to declare the types of variables, parameters, and return values at compile-time.
- Static typing helps catch potential errors during development, offering a level of code safety that may not be achievable in pure JavaScript.

2. Compatibility with JavaScript:

- TypeScript is a superset of JavaScript, meaning that any valid JavaScript code is also valid TypeScript code.
- Developers can gradually adopt TypeScript in existing JavaScript projects without the need for a full rewrite.

3. Tooling Support:

- TypeScript comes with a rich set of tools and features for development, including code editors (like Visual Studio Code) with built-in TypeScript support.
- The TypeScript compiler (tsc) translates TypeScript code into plain JavaScript, allowing it to run in any JavaScript environment.

4. Enhanced IDE Experience:

- IDEs (Integrated Development Environments) that support TypeScript offer improved code navigation, autocompletion, and better refactoring capabilities.
- TypeScript's type information enhances the overall development experience.

5. Interfaces and Type Declarations:

- TypeScript introduces concepts like interfaces and type declarations, enabling developers to define clear contracts for their code.
- Interfaces help document the shape of objects, making it easier to understand and maintain the code.

6. Compilation:

- TypeScript code is transpiled to JavaScript during the compilation process, ensuring that the resulting code is compatible with various JavaScript environments and browsers.

Overall, TypeScript provides developers with the benefits of static typing while preserving the flexibility and features of JavaScript. It has gained popularity in large-scale applications and projects where maintaining code quality and catching errors early are crucial.

Execution of TypeScript Code

TypeScript code doesn't run natively in browsers or JavaScript environments. Instead, it undergoes a compilation process to generate equivalent JavaScript code. Here's an overview of how TypeScript code is executed:

1. Writing TypeScript Code:

- Developers write TypeScript code using `.ts` or `.tsx` files, employing TypeScript's syntax with features like static typing, interfaces, and type annotations.

1. TypeScript Compiler (tsc):

- The TypeScript Compiler (`tsc`) is a command-line tool that processes TypeScript code.
- Developers run `tsc` to initiate the compilation process.

1. Compilation Process:

- The TypeScript Compiler parses and analyzes the TypeScript code, checking for syntax errors and type-related issues.
- It generates equivalent JavaScript code, typically in one or more `.js` or `.jsx` files.

1. Generated JavaScript Code:

- The output JavaScript code closely resembles the original TypeScript code but lacks TypeScript-specific constructs like type annotations.
- TypeScript features that aren't present in JavaScript (e.g., interfaces) are often transpiled or emitted in a way that doesn't affect runtime behavior.

1. JavaScript Execution:

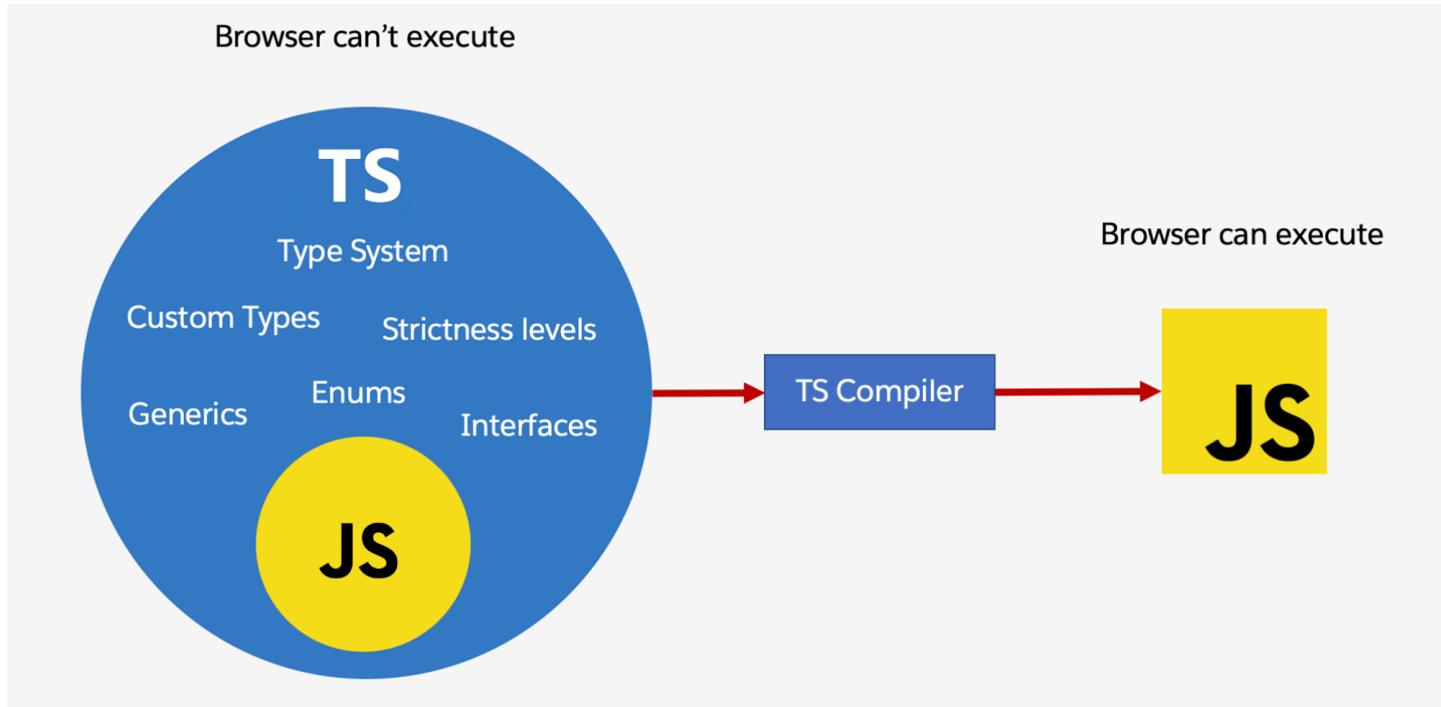
- The generated JavaScript code can now be executed by any JavaScript runtime or browser.
- Developers can include the resulting JavaScript files in HTML documents or use them in Node.js environments.

1. Runtime Environment:

- In the chosen runtime environment, the JavaScript code is interpreted or compiled by the JavaScript engine (e.g., V8 in Chrome, SpiderMonkey in Firefox).
- Just-in-time (JIT) compilation or interpretation occurs to convert the code into machine code that the computer's processor can execute.

1. Interacting with the DOM (Browser Environments):

- In browser environments, the JavaScript code, generated from TypeScript, may interact with the Document Object Model (DOM) to manipulate web page structure and behavior.



TypeScript Compiler (`tsc`)

- The TypeScript Compiler (`tsc`) is responsible for transpiling TypeScript code into JavaScript.
- It is a part of the official TypeScript distribution and can be installed using tools like npm.
- Developers run `tsc` from the command line, specifying the TypeScript file(s) they want to compile.
- Configuration for the compilation process can be provided via a `tsconfig.json` file.
- The compiler performs type checking, emits JavaScript files, and allows customization of compilation options.

In summary, TypeScript code is transformed into JavaScript through the TypeScript Compiler (`tsc`). This compilation process ensures that TypeScript's features are compatible with existing JavaScript environments, enabling developers to benefit from static typing during development while still producing standard JavaScript for execution.

In addition to the TypeScript Compiler (`tsc`), several alternative tools have gained popularity for their efficiency, speed, and additional features when transpiling TypeScript to JavaScript. Here are a couple of noteworthy ones:

1. esbuild: a highly performant JavaScript bundler and minifier, but it also supports TypeScript.

2. swc (Speedy Web Compiler): a fast and low-level JavaScript/TypeScript compiler.

Setting up a Typescript Nodejs Application

Let's walk through the process of setting up a simple TypeScript Node.js application locally on your machine.

Step 1 - Install TypeScript Globally:

```
npm install -g typescript
```

This command installs TypeScript globally on your machine, allowing you to use the `tsc` command anywhere.

Step 2 - Initialize a Node.js Project with TypeScript:

```
mkdir node-app
cd node-app
npm init -y
npx tsc --init
```

These commands create a new directory (`node-app`), initialize a Node.js project with default settings (`npm init -y`), and then generate a `tsconfig.json` file using `npx tsc --init`.

Step 3 - Create a TypeScript File (a.ts):

```
// a.ts
const x: number = 1;
console.log(x);
```

Step 4 - Compile the TypeScript File to JavaScript:

```
tsc -b
```

The `-b` flag tells TypeScript to build the project based on the configuration in `tsconfig.json`. This generates a JavaScript file (`index.js`) from the TypeScript source (`a.ts`).

Step 5 - Explore the Generated JavaScript File (`index.js`):

```
// index.js
const x = 1;
console.log(x);
```

Note that the generated JavaScript file doesn't include TypeScript-specific code. It's standard JavaScript without types.

Step 6 - Attempt to Assign a String to a Number:

```
// a.ts
let x: number = 1;
x = "harkirat";
console.log(x);
```

Step 7 - Try Compiling the Code Again:

```
tsc -b
```

Upon compiling, TypeScript detects the type error (`x` being assigned a string) and reports it in the console. Additionally, no `index.js` file is generated due to the type error.

This example illustrates one of TypeScript's key benefits: catching type errors at compile time. By providing static typing, TypeScript enhances code reliability and helps identify potential issues before runtime. This is particularly valuable in large codebases where early error detection can save time and prevent bugs.

Basic Types in Typescript

In TypeScript, basic types serve as the building blocks for defining the data types of variables. Here's an overview of some fundamental types provided by TypeScript:

1. Number:

- Represents numeric values.
- Example:

```
let age: number = 25;
```

2. String:

- Represents textual data (sequences of characters).
- Example:

```
let name: string = "John";
```

3. Boolean:

- Represents true or false values.
- Example:

```
let isStudent: boolean = true;
```

4. Null:

- Represents the absence of a value.
- Example:

```
let myVar: null = null;
```

5. Undefined:

- Represents a variable that has been declared but not assigned a value.
- Example:

```
let myVar: undefined = undefined;
```

Problems and Code Implementation

1] Hello World Greeting

Objective:

Learn how to give types to function arguments in TypeScript.

Task:

Write a TypeScript function named `greet` that takes a user's first name as an argument and logs a greeting message to the console.

Function Signature:

```
function greet(firstName: string): void {  
    // Implementation goes here  
}
```

Solution:

```
function greet(firstName: string): void {  
    console.log("Hello " + firstName);  
}  
  
// Example Usage  
greet("harkirat");
```

Explanation:

1. Function Definition (`function greet(firstName: string): void`):

- The `greet` function is declared with a parameter named `firstName`.
- `: string` indicates that the `firstName` parameter must be of type `string`.
- `: void` specifies that the function does not return any value.

2. Function Body (`console.log("Hello " + firstName);`):

- Inside the function body, a `console.log` statement prints a greeting message to the console.
- The message includes the provided `firstName` parameter.

3. Function Invocation (`greet("harkirat");`):

- The function is called with the argument `"harkirat"`.

- The provided argument must be a string, aligning with the specified type in the function definition.

This example demonstrates the basic usage of TypeScript types in function parameters, ensuring that the expected data type is enforced and catching errors related to type mismatches during development.

2] Sum Function

Objective:

Learn how to assign a return type to a function in TypeScript.

Task:

Write a TypeScript function named `sum` that takes two numbers as arguments and returns their sum. Additionally, invoke the function with an example.

Function Signature:

```
function sum(a: number, b: number): number {  
    // Implementation goes here  
}
```

Solution:

```
function sum(a: number, b: number): number {  
    return a + b;  
}  
  
// Example Usage  
console.log(sum(2, 3));
```

Explanation:

1. Function Definition (`function sum(a: number, b: number): number`):

- The `sum` function is declared with two parameters, `a` and `b`, both of type number.
- `: number` indicates that the function returns a value of type number.

2. Function Body (`return a + b;`):

- Inside the function body, the sum of `a` and `b` is calculated using the `+` operator.
- The result is then returned.

3. Function Invocation (`console.log(sum(2, 3));`):

- The function is called with the arguments `2` and `3`.
- The result is logged to the console using `console.log`.

This example showcases how to specify the return type of a function in TypeScript, ensuring that the function returns the expected data type. In this case, the `sum` function returns a number.

3] Age Verification Function

Objective:

Understand Type Inference in TypeScript.

Task:

Write a TypeScript function named `isLegal` that takes an `age` as a parameter and returns `true` if the user is 18 or older, and `false` otherwise. Also, invoke the function with an example.

Function Signature:

```
function isLegal(age: number): boolean {
    // Implementation goes here
}
```

Solution:

```
function isLegal(age: number): boolean {
    if (age > 18) {
        return true;
    }
}
```

```
    } else {
        return false;
    }
}

// Example Usage
console.log(isLegal(22)); // Output: true
```

Explanation:

1. Function Definition (`function isLegal(age: number): boolean`):

- The `isLegal` function is declared with a parameter `age` of type `number`.
- `: boolean` indicates that the function returns a boolean value.

2. Function Body (`if (age > 18) {...}`):

- Inside the function body, an `if` statement checks if the provided `age` is greater than 18.
- If true, the function returns `true`; otherwise, it returns `false`.

3. Function Invocation (`console.log(isLegal(22))`):

- The function is called with the argument `22`.
- The result (`true`) is logged to the console using `console.log`.

This example demonstrates how TypeScript's type inference can be leveraged. The return type (`boolean`) is implicitly inferred based on the conditions within the function. The `isLegal` function is designed to return a boolean value indicating whether the provided age is 18 or older.

4] Delayed Function Execution

Objective:

Learn to work with functions as parameters in TypeScript.

Task:

Write a TypeScript function named `delayedCall` that takes another function (`fn`) as input and executes it after a delay of 1 second. Also, invoke the `delayedCall` function with an example.

Function Signature:

```
function delayedCall(fn: () => void): void {
    // Implementation goes here
}
```

Solution:

```
function delayedCall(fn: () => void): void {
    setTimeout(fn, 1000);
}

// Example Usage
delayedCall(function() {
    console.log("hi there");
});
```

Explanation:

1. Function Definition (`function delayedCall(fn: () => void): void`):

- The `delayedCall` function is declared with a parameter `fn` of type function that takes no arguments and returns `void`.
- `: void` indicates that the function doesn't return any value.

2. Function Body (`setTimeout(fn, 1000);`):

- Inside the function body, `setTimeout` is used to delay the execution of the provided function (`fn`) by 1000 milliseconds (1 second).

3. Function Invocation (`delayedCall(function() {...});`):

- The `delayedCall` function is invoked with an anonymous function as an argument.
- The provided function logs "hi there" to the console after a 1-second delay.

This example illustrates how TypeScript handles functions as first-class citizens, allowing them to be passed as arguments to other functions. The `delayedCall` function provides a way to execute a given function after a specified delay.

The `tsconfig.json` File in TypeScript

The `tsconfig.json` file in TypeScript is a configuration file that provides settings for the TypeScript compiler (`tsc`). It allows you to customize various aspects of the compilation process and define how your TypeScript code should be transpiled into JavaScript.

Below are a bunch of options that you can change to change the compilation process in the `tsconfig.json` file:

1] Target Option in `tsconfig.json` :

The `target` option in a `tsconfig.json` file specifies the ECMAScript target version to which the TypeScript compiler (`tsc`) will compile the TypeScript code. It allows you to define the lowest version of ECMAScript that your code should be compatible with. Here's an explanation and example usage:

- **ES5 (ECMAScript 5):**

- When `target` is set to `"es5"`, the TypeScript compiler generates code compatible with ECMAScript 5, which is widely supported across browsers.
- Example:

```
{
  "compilerOptions": {
    "target": "es5",
    // Other options...
  }
}
```

- **TypeScript Code:**

```
const greet = (name: string) => `Hello, ${name}!`;
```

- **Output:**

```
"use strict";
var greet = function (name) { return "Hello, ".concat(name, "!"); };
```

- **ES2020 (ECMAScript 2020):**

- When `target` is set to `"es2020"`, the TypeScript compiler generates code compatible with ECMAScript 2020, incorporating the latest features.
- Example:

```
{
  "compilerOptions": {
    "target": "es2020",
    // Other options...
  }
}
```

- TypeScript Code:

```
const greet = (name: string) => `Hello, ${name}!`;
```

- Output:

```
"use strict";
const greet = (name) => `Hello, ${name}!`;
```

By setting the `target` option, you ensure that the generated JavaScript code adheres to the specified ECMAScript version, allowing you to control the level of compatibility and take advantage of the features available in newer ECMAScript versions.

2] `rootDir`:

- The `rootDir` option in a `tsconfig.json` file specifies the root directory where the TypeScript compiler (`tsc`) should look for `.ts` files.
- It is considered a good practice to set `rootDir` to the source folder (`src`), indicating the starting point for TypeScript file discovery.
- Example:

```
{
  "compilerOptions": {
    "rootDir": "src",
    // Other options...
  }
}
```

```
    }  
}
```

3] outDir

- The `outDir` option defines the output directory where the TypeScript compiler will place the generated `.js` files.
- It determines the structure of the output directory relative to the `rootDir`.
- Example:

```
{  
  "compilerOptions": {  
    "outDir": "dist",  
    // Other options...  
  }  
}
```

If `rootDir` is set to `"src"` and `outDir` is set to `"dist"`, the compiled files will be placed in the `dist` folder, mirroring the structure of the `src` folder.

4] noImplicitAny

- The `noImplicitAny` option in a `tsconfig.json` file determines whether TypeScript should issue an error when it encounters a variable with an implicit `any` type.
- Enabled (`"noImplicitAny": true`):

```
{  
  "compilerOptions": {  
    "noImplicitAny": true,  
    // Other options...  
  }  
}
```

Example:

```
// Compilation Error: Implicit any type
const greet = (name) => `Hello, ${name}!`;
```

- **Disabled ("noImplicitAny": false):**

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    // Other options...
  }
}
```

No error will be issued for implicit `any` types.

5] removeComments

- The `removeComments` option in a `tsconfig.json` file determines whether comments should be included in the final JavaScript output.
- **Enabled ("removeComments": true):**

```
{
  "compilerOptions": {
    "removeComments": true,
    // Other options...
  }
}
```

Comments will be stripped from the generated JavaScript files.

- **Disabled ("removeComments": false):**

```
{
  "compilerOptions": {
    "removeComments": false,
    // Other options...
  }
}
```

Comments will be retained in the generated JavaScript files.

These options provide flexibility and control over the compilation process, allowing you to structure your project and handle type-related scenarios according to your preferences.

Interfaces

In TypeScript, an interface is a way to define a contract for the shape of an object. It allows you to specify the expected properties, their types, and whether they are optional or required. Interfaces are powerful tools for enforcing a specific structure in your code.

Understanding Interfaces

Suppose you have an object representing a user:

```
const user = {
  firstName: "harkirat",
  lastName: "singh",
  email: "email@gmail.com",
  age: 21,
};
```

To assign a type to the `user` object using an interface, you can create an interface named `User`:

```
interface User {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
}
```

Now, you can explicitly specify that the `user` object adheres to the `User` interface:

```
const user: User = {
  firstName: "harkirat",
  lastName: "singh",
  email: "email@gmail.com",
  age: 21,
};
```

Explanation:

1. Interface Declaration (`interface User`):

- The `interface` keyword is used to declare an interface named `User`.
- Inside the interface, you define the expected properties (`firstName`, `lastName`, `email`, `age`) along with their types.

2. Assigning Type to Object (`const user: User = { /* ... */ };`):

- By stating `const user: User`, you are explicitly indicating that the `user` object must adhere to the structure defined by the `User` interface.
- If the `user` object deviates from the defined structure or misses any required property, TypeScript will raise a compilation error.

Assignment 1

Problem: Create a function `isLegal` that returns true or false if a user is above 18. It takes a user as an input.

Solution:

```
// Define an interface to specify the structure of a user object
interface User {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
}

// Create a function 'isLegal' that checks if a user is above 18
function isLegal(user: User): boolean {
  // Check if the user's age is greater than 18
  if (user.age > 18) {
    return true; // Return true if the user is legal
  } else {
    return false; // Return false if the user is not legal
  }
}
```

Code Explanation:

- An interface "User" is defined to enforce the structure of a user object with properties: firstName, lastName, email, and age.
- The function "isLegal" takes a user object as input and checks if the user's age is greater than 18.
- It returns true if the user is legal (age > 18) and false otherwise.

Assignment 2

Problem: Create a React component that takes todos as an input and renders them.

Solution:

```
// Define an interface to specify the structure of a todo object
interface TodoType {
  title: string;
  description: string;
  done: boolean;
}

// Define the input prop for the Todo component
interface TodoInput {
  todo: TodoType;
}

// Create a React component 'Todo' that takes a 'todo' prop and renders it
function Todo({ todo }: TodoInput): JSX.Element {
  return (
    <div>
      <h1>{todo.title}</h1>
      <h2>{todo.description}</h2>
      {/* Additional rendering logic can be added for other properties */}
    </div>
  );
}
```

Code Explanation:

- An interface "TodoType" is defined to specify the structure of a todo with properties: title, description, and done.
- An interface "TodoInput" is defined to specify the input prop for the Todo component.

- The React component "Todo" takes a prop "todo" of type "TodoType" and renders its properties (title and description).

Implementing Interfaces

In TypeScript, you can implement interfaces using classes. This provides a way to define a blueprint for the structure and behavior of a class. Let's take an example:

Assume you have a `Person` interface:

```
interface Person {  
    name: string;  
    age: number;  
    greet(phrase: string): void;  
}
```

Now, you can create a class that adheres to this interface:

```
class Employee implements Person {  
    name: string;  
    age: number;  
  
    constructor(n: string, a: number) {  
        this.name = n;  
        this.age = a;  
    }  
  
    greet(phrase: string) {  
        console.log(` ${phrase} ${this.name}`);  
    }  
}
```

Here's what's happening:

- The `Employee` class implements the `Person` interface.
- It has properties (`name` and `age`) matching the structure defined in the interface.
- The `greet` method is implemented as required by the interface.

This approach is handy when creating various types of persons (like Manager, CEO), ensuring they all adhere to the same interface contract. It maintains consistency in the structure and behavior across different classes.

Types

In TypeScript, **types** allow you to aggregate data together in a manner very similar to interfaces. They provide a way to define the structure of an object, similar to how interfaces do. Here's an example:

```
type User = {
    firstName: string;
    lastName: string;
    age: number;
};
```

Features

1. Unions:

Unions allow you to define a type that can be one of several types. This is useful when dealing with values that could have different types. For instance, imagine you want to print the ID of a user, which can be either a number or a string:

```
type StringOrNumber = string | number;

function printId(id: StringOrNumber) {
    console.log(`ID: ${id}`);
}

printId(101);      // ID: 101
printId("202");   // ID: 202
```

Unions provide flexibility in handling different types within a single type definition.

1. Intersection:

Intersections allow you to create a type that has every property of multiple types or interfaces. If you have types like `Employee` and `Manager`, and you want to create a `TeamLead` type that combines properties of both:

```
type Employee = {
    name: string;
    startDate: Date;
};
```

```
type Manager = {  
    name: string;  
    department: string;  
};  
  
type TeamLead = Employee & Manager;  
  
const teamLead: TeamLead = {  
    name: "harkirat",  
    startDate: new Date(),  
    department: "Software Developer"  
};
```

Intersections provide a way to create a new type that inherits properties from multiple existing types.

In summary, while types and interfaces are similar in defining object structures, types in TypeScript offer additional features like unions and intersections, making them more versatile in certain scenarios.

Interfaces vs Types

Major Differences

1. Declaration Syntax:

- **Type:**
 - Uses the `type` keyword.
 - More flexible syntax, can represent primitive types, unions, intersections, and more.
- **Interface:**
 - Uses the `interface` keyword.
 - Typically used for defining the structure of objects.

2. Extension and Merging:

- **Type:**

- Supports extending types.
- Can't be merged; if you define another type with the same name, it will override the previous one.
- **Interface:**
 - Supports extending interfaces using the `extends` keyword.
 - Automatically merges with the same-name interfaces, combining their declarations.

3. Declaration vs. Implementation:

- **Type:**
 - Can represent any type, including primitives, unions, intersections, etc.
 - Suitable for describing the shape of data.
- **Interface:**
 - Mainly used for describing the shape of objects.
 - Can also be used to define contracts for classes.

Other Differences

- **Type Overriding:**
 - Types cannot be overridden or merged. Redefining a type with the same name replaces the previous one.
 - Interfaces automatically merge if declared with the same name.
- **Object Literal Strictness:**
 - Types are more lenient when dealing with object literal assignments.
 - Interfaces enforce strict object literal shapes.
- **Implementation for Classes:**
 - Interfaces can be used to define contracts for class implementations.
 - Types are more versatile for creating complex types and reusable utility types.

When to Use Which

- **Use Types:**

- For advanced scenarios requiring union types, intersections, or mapped types.
- When dealing with primitive types, tuples, or non-object-related types.
- Creating utility types using advanced features like conditional types.

- **Use Interfaces:**

- When defining the structure of objects or contracts for class implementations.
- Extending or implementing other interfaces.
- When consistency in object shape is a priority.

Examples

Type Example:

```
type StringOrNumber = string | number;

function printId(id: StringOrNumber) {
  console.log(`ID: ${id}`);
}

printId(101);      // ID: 101
printId("202");    // ID: 202
```

Interface Example:

```
interface Employee {
  name: string;
  startDate: Date;
}

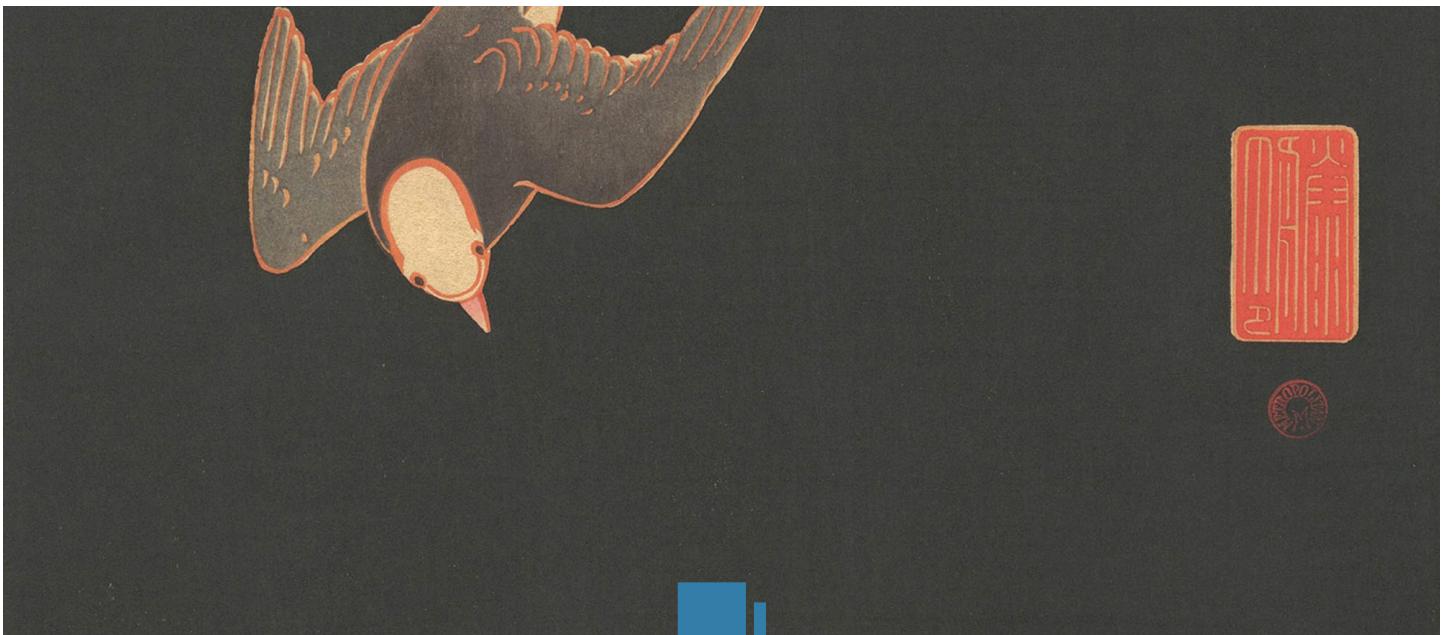
interface Manager {
  name: string;
  department: string;
}

type TeamLead = Employee & Manager;

const teamLead: TeamLead = {
  name: "Harkirat",
  startDate: new Date(),
```

```
department: "Software Developer",  
};
```

In summary, choose types for flexibility and advanced type features, and use interfaces for object shapes, contracts, and class implementations, ensuring a consistent and readable codebase.



Week 10.1

Understanding Postgres

In today's lecture, Harkirat introduces the PostgreSQL database, beginning with an overview of [different database types](#) and the limitations of NoSQL databases. The discussion then pivots to the [necessity of SQL databases](#), highlighting their advantages in certain scenarios.

We further look into the process of [creating our first PostgreSQL database](#), exploring how to [interact with it and the various operations](#) that can be executed. This provides a solid foundation for understanding the principles of relational database management systems.

[Understanding Postgres](#)

[What We'll Learn Today](#)

[Types of Databases](#)

[NoSQL Databases](#)

[Graph Databases](#)

[Vector Databases](#)

[SQL Databases](#)

[Why Not NoSQL](#)

[What is Schemaless?](#)

[Problems with Schemaless Databases](#)

[Upsides of Schemaless Databases](#)

[Mongoose and Schema Enforcement](#)

[Why SQL?](#)

[1. Strict Schema](#)

[2. Running the Database](#)

[3. Connecting and Manipulating Data](#)

[Benefits of SQL Databases](#)

[Creating a PostgreSQL Database](#)

[1\] Using Neon](#)[2\] Using Docker Locally](#)[3\] Using Docker on Windows](#)[Connection String](#)[Understanding the Connection String Components](#)[Understanding Vector Databases:](#)[Interact with PostgreSQL](#)[1. psql](#)[2. pg \(node-postgres\)](#)[Creating a table schema](#)[Creating a Table in SQL](#)[1. Initiate Table Creation](#)[2. Define Columns and Constraints](#)[Practical Steps](#)[Interacting with the database](#)[1. INSERT \(Create\)](#)[2. UPDATE](#)[3. DELETE](#)[4. SELECT \(Read\)](#)[Practical Tips](#)[Database Operations](#)[Installing the pg Library](#)[Connecting to the Database](#)[Querying the Database](#)[INSERT](#)[UPDATE](#)[DELETE](#)[SELECT](#)[Creating a Table](#)[Conclusion](#)[Creating a Simple Node.js App](#)[Step 1: Initialize a TypeScript Project](#)[Step 2: Install Dependencies](#)[Step 3: Create a Simple Node.js App](#)[Insert Data Function](#)[Fetch Data Function](#)

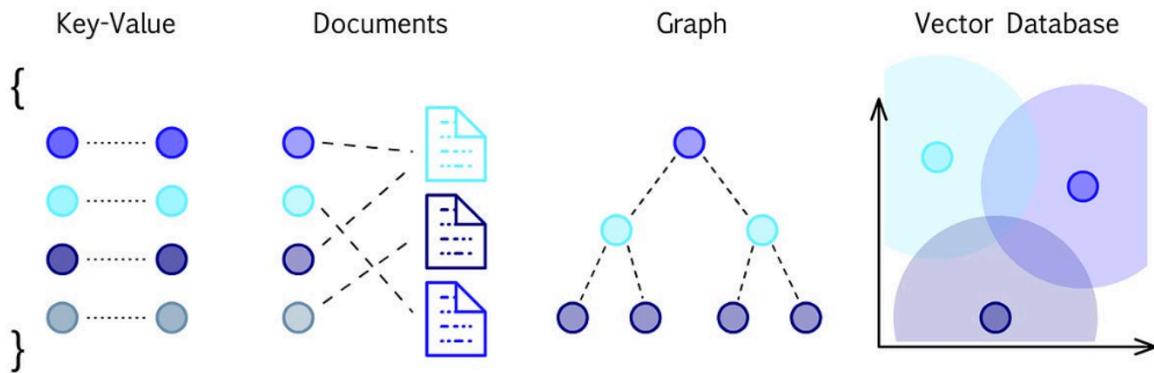
What We'll Learn Today

- Simple Topics:

- **SQL vs NoSQL:** Understanding the differences between structured SQL databases and flexible NoSQL databases.
- **Creating Postgres Databases:** Learning how to set up and configure PostgreSQL databases.
- **CRUD Operations:** Performing Create, Read, Update, and Delete operations on database records.

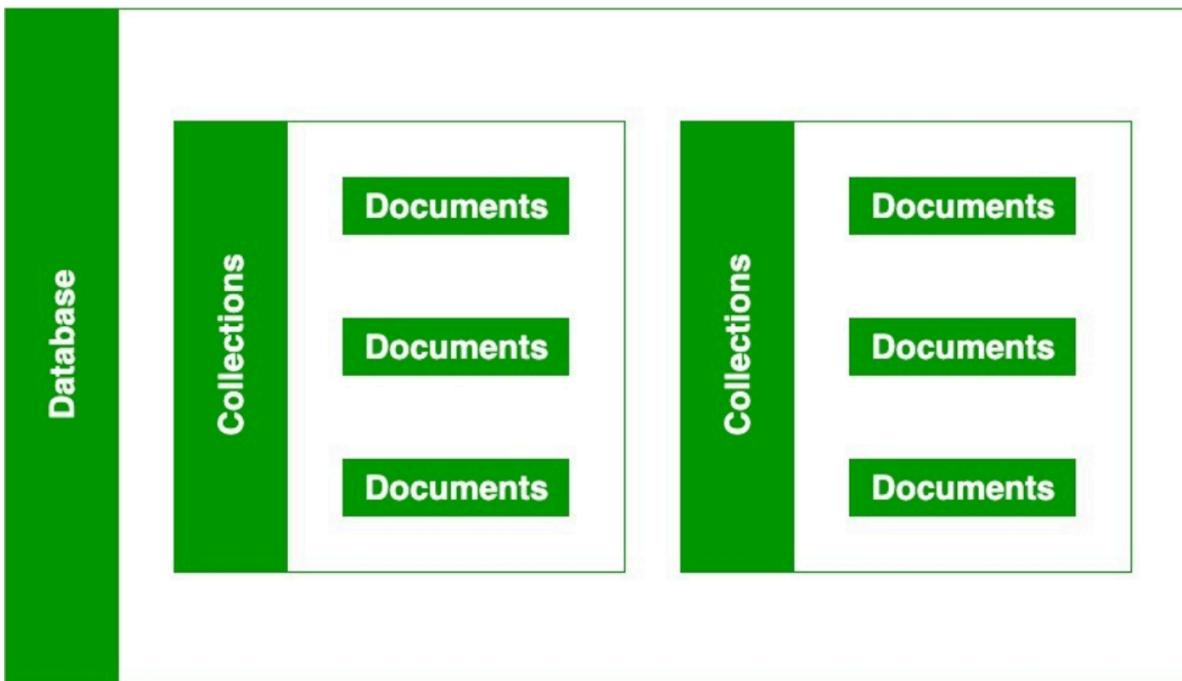
Types of Databases

Databases are an essential component of many applications, serving as the backbone for data storage and retrieval. There are several types of databases, each designed to serve specific use cases and data management needs. Below is an elaboration on the types of databases:



NoSQL Databases

- **Definition:** NoSQL databases are designed to store, retrieve, and manage large volumes of unstructured or semi-structured data. They are known for their flexibility, scalability, and high performance.
- **Schema-less:** Unlike SQL databases, NoSQL databases do not require a predefined schema, allowing for the storage of data in various formats.
- **Use Cases:** Ideal for big data applications, real-time web apps, and for handling large volumes of data that may not fit neatly into a relational model.
- **Examples:** MongoDB, Cassandra, Redis, Couchbase.

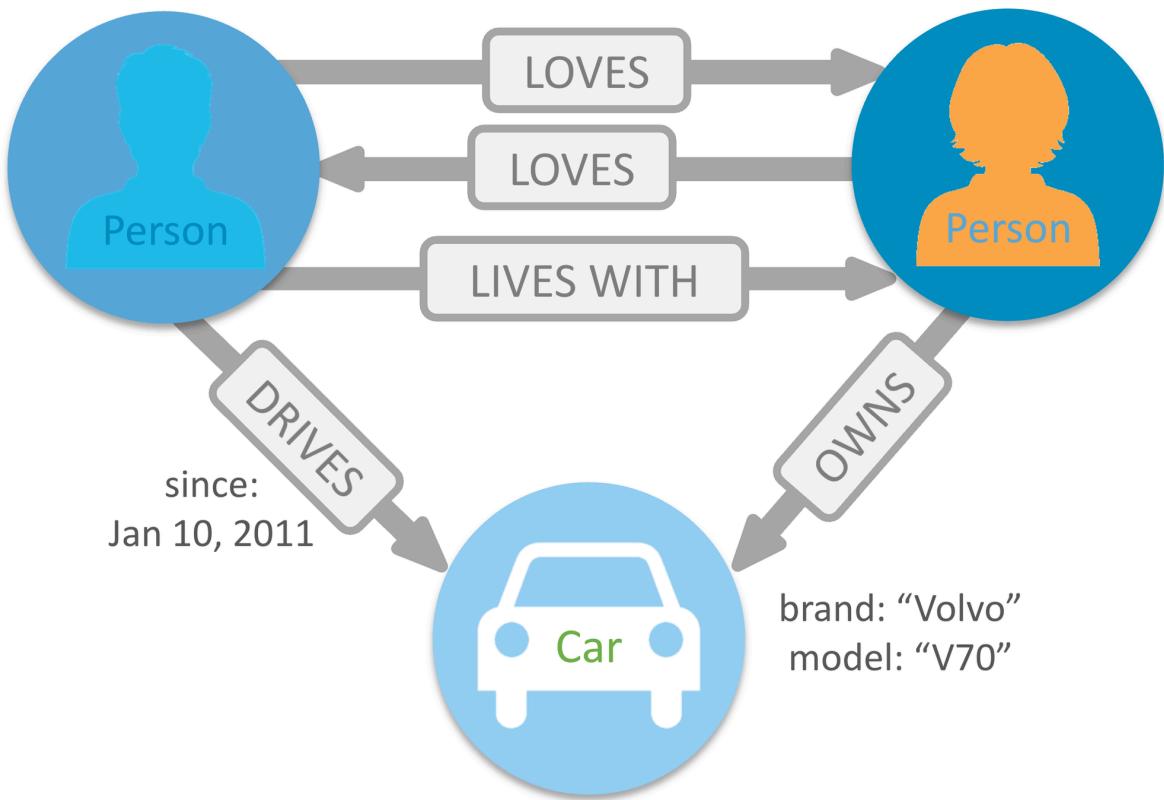


Graph Databases

- **Definition:** Graph databases are designed to store and navigate relationships. They treat the relationships between data as equally important as the data itself.
- **Data Storage:** Data is stored in nodes (entities) and edges (relationships), which makes them highly efficient for traversing and querying complex relationships.
- **Use Cases:** Particularly useful for social networks, recommendation engines, fraud detection, and any domain where relationships are key.
- **Examples:** Neo4j, Amazon Neptune, OrientDB.

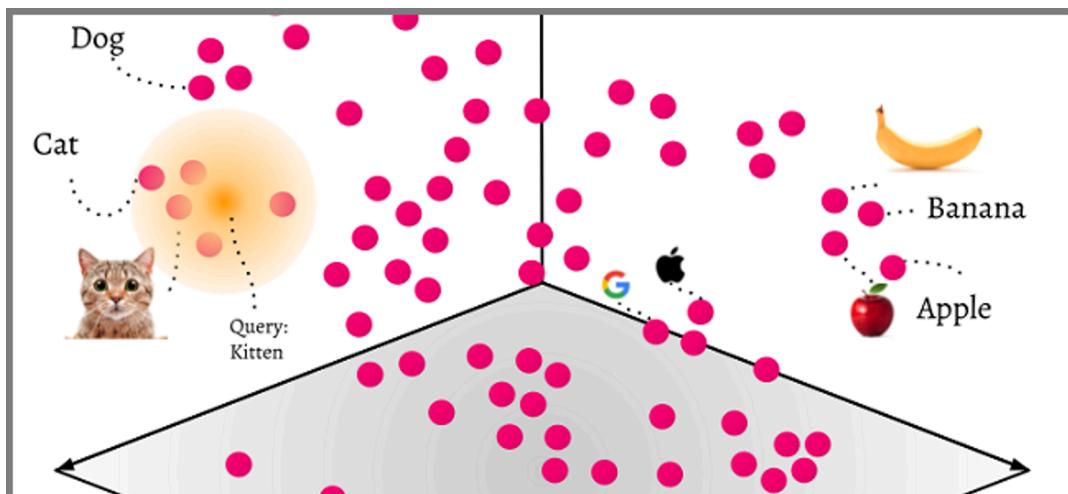
name: "Dan"
 born: May 29, 1970
 twitter: "@dan"

name: "Ann"
 born: Dec 5, 1975



Vector Databases

- Definition:** Vector databases are specialized databases optimized for vector similarity searching. They are used to store and process vector embeddings typically generated by machine learning models.
- Data Storage:** Data is stored in the form of vectors, which are arrays of numbers that represent data in a high-dimensional space.
- Use Cases:** Useful in machine learning applications, such as image recognition, natural language processing, and recommendation systems.
- Examples:** Pinecone, Milvus, Faiss.



SQL Databases

- **Definition:** SQL databases, also known as relational databases, store data in predefined schemas and tables with rows and columns.
- **Data Storage:** Data is organized into tables, and each row in a table represents a record with a unique identifier called the primary key.
- **Use Cases:** Most full-stack applications use SQL databases for their ability to maintain ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring reliable transaction processing.
- **Examples:** MySQL, PostgreSQL, Oracle, SQL Server.

user_id	first_name	last_name	age
1	Joe	Doe	29
2	Jane	Dan	31
3	Potter	Paul	39
4	Pil	Passot	41

Table: Users

order_id	name	price	user_id
1	Wristwatch	\$10	4
2	Keyboard	\$42	2
3	Chair	\$120	4
4	Phone	\$310	1

Table: Orders

Why Not NoSQL

While NoSQL databases like MongoDB offer significant advantages, particularly in terms of flexibility and speed of development, they also come with potential drawbacks that can become more pronounced as an application scales. Here's an elaboration on the points mentioned:

What is Schemaless?

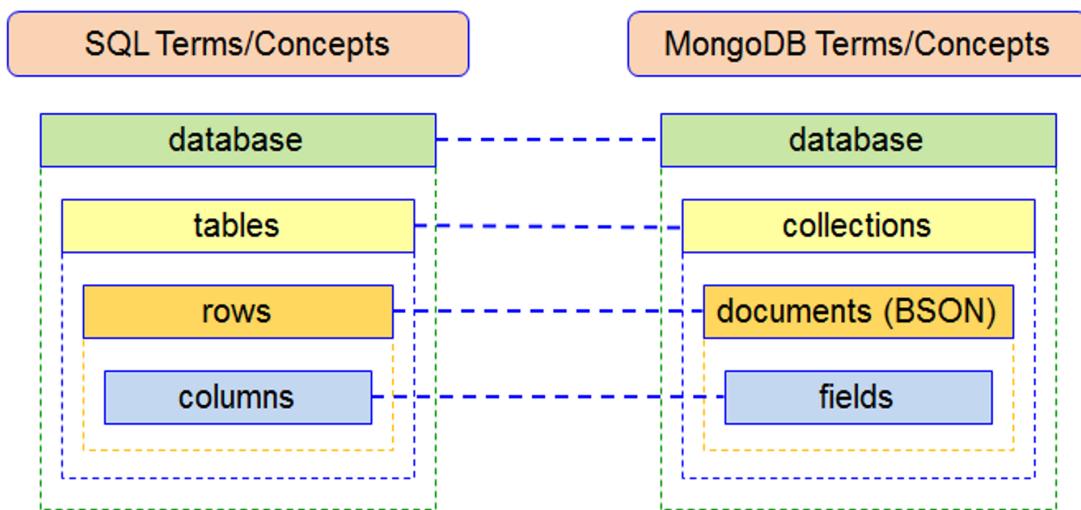
- **Definition:** In a schemaless database, the structure of the data is not defined beforehand. This means that each 'row' or document can have a different set of fields (keys) and data types (values).
- **Flexibility:** This allows for the storage of heterogeneous data and can accommodate changes in the data model without requiring migrations or alterations to the existing data structure.

Problems with Schemaless Databases

- **Inconsistent Database:** Without a uniform structure, data can become inconsistent. For example, one document might have a field that others don't, leading to unpredictable query results.
- **Runtime Errors:** Applications may expect certain fields or data types that are not present in all documents, leading to errors at runtime when the code tries to access or manipulate non-existent fields.
- **Too Flexible:** The flexibility that makes NoSQL databases appealing for rapid development can become a liability for applications that require strict data integrity and consistency.

Upsides of Schemaless Databases

- **Speed:** Developers can iterate quickly without being constrained by a rigid database schema. This is particularly useful in the early stages of a project or when requirements are rapidly evolving.
- **Schema Evolution:** It's easier to adapt to changes in the application's data requirements, as there's no need to perform complex migrations or updates to the database schema.



Mongoose and Schema Enforcement

- **Mongoose Schemas:** Mongoose is an Object Data Modeling (ODM) library for MongoDB that allows developers to define schemas at the application level. It provides a layer of data validation and structure in a Node.js environment.
- **Application-Level Strictness:** While Mongoose enforces schema rules in the application code, it does not impose these constraints at the database level. MongoDB itself remains schemaless.
- **Potential for Erroneous Data:** Even with Mongoose schemas, it's still possible to insert data that doesn't conform to the defined schema directly into the database, bypassing the application's validation logic.

Why SQL?

SQL (Structured Query Language) databases, also known as relational databases, have been the cornerstone of data storage and management in software applications for decades. Their approach to

data management offers several advantages, particularly in terms of data integrity, consistency, and reliability. Here's an elaboration on the structured approach of SQL databases and the four key aspects of using them:

1. Strict Schema

- **Define Your Schema:** Before inserting data into an SQL database, you must define a schema. This schema dictates the structure of the data, including the tables, columns, data types, and relationships between tables.
- **Data Integrity:** By requiring all data to adhere to the predefined schema, SQL databases ensure a high level of data integrity. Each column in a table is designed to hold data of a specific type, and relationships between tables are strictly enforced.
- **Schema Updates and Migrations:** As your application evolves, you may need to update the database schema. This typically involves performing migrations—carefully managed changes that may add, remove, or alter tables and columns without losing data.

2. Running the Database

- **Database Server:** An SQL database runs on a database server, which can be hosted locally on your development machine, on-premises in your data center, or in the cloud. Running the database involves setting it up, configuring it for performance and security, and ensuring it's accessible for data operations.

3. Connecting and Manipulating Data

- **Using a Library:** To interact with an SQL database, you typically use a library or an ORM (Object-Relational Mapping) tool that facilitates the connection and allows you to perform data operations in a more abstracted way, often using the programming language of your application.
- **Creating Tables and Defining Schemas:** Before you can store data, you need to create tables and define their schema. This includes specifying the columns, data types, primary keys, foreign keys, and any constraints to enforce data integrity.
- **Running Queries:** SQL databases are interacted with through SQL queries. These queries allow you to perform a variety of data operations, including:
 - **Insert:** Adding new records to a table.
 - **Update:** Modifying existing records based on specific criteria.
 - **Delete:** Removing records from a table.
 - **Select:** Retrieving data from one or more tables, often involving complex filtering, sorting, and joining operations.

Benefits of SQL Databases

- **Data Integrity and Consistency:** The strict schema and relational model of SQL databases ensure that data is stored in a consistent and reliable manner.
- **ACID Properties:** SQL databases are designed to guarantee ACID (Atomicity, Consistency, Isolation, Durability) properties, making them ideal for applications that require transactions to be processed reliably.
- **Complex Queries:** The SQL language provides powerful querying capabilities, allowing for complex data retrieval that can involve multiple tables and conditions.

- **Mature Ecosystem:** SQL databases have been around for a long time, resulting in a mature ecosystem of tools, libraries, and best practices.

Creating a PostgreSQL Database

Creating a PostgreSQL database can be done in several ways, depending on your environment and preferences. Here are a few methods:

1] Using Neon

- **Neon:** Neon is a cloud service that allows you to create and manage PostgreSQL databases without the need to handle the underlying infrastructure.
- **Steps:**
 1. Visit [Neon's website](#) and sign up for an account.
 2. Follow the instructions to create a new PostgreSQL server.
 3. Once created, you will be provided with a connection string that you can use to connect to your database from your application.
- **Connection String Example:**

```
postgresql://username:password@ep-broken-frost-69135494.us-east-2.aws.neon.tech/calm-gobbler-41_dt
```

2] Using Docker Locally

- **Docker:** Docker allows you to run PostgreSQL in an isolated container on your local machine.
- **Steps:**
 1. Install Docker if you haven't already.
 2. Run the following command to start a PostgreSQL container:

```
docker run --name my-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres
```
- **Connection String Example:**

```
postgresql://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disable
```

3] Using Docker on Windows

- **Windows Terminal with Docker:**
 1. Ensure Docker Desktop for Windows is installed and running.
 2. Open Windows Terminal or any command-line interface.
 3. To download and run the PostgreSQL image for the first time, use:

```
docker run --name my-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres
```

```
docker run --name my-postgres1 -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres
```

4. If the image is already downloaded, you can start the container with:

```
docker start my-postgres1
```

5. To connect to the PostgreSQL instance, use:

```
docker exec -it my-postgres1 psql -U postgres -d postgres
```

6. Enter the password when prompted to access the PostgreSQL command line interface.

7. Inside the PostgreSQL CLI, you can list all tables with:

```
\dt
```

Connection String

- The connection string is a vital piece of information that your application uses to connect to the database. It includes the username, password, host, port, and database name.
- Format:**

```
postgresql://[:user[:password]@[host]:[port]/[database]?[options]
```

- This format is similar to what you might have seen with MongoDB and Mongoose, where the connection string is used to establish a connection to the database from your application code.

Understanding the Connection String Components

- postgresql://** This is the protocol indicating that you are connecting to a PostgreSQL database.
- username:password** Credentials for authenticating with the database.
- host** The server where the database is hosted (e.g., localhost, a remote server, or a cloud service like Neon).
- port** The port number on which the PostgreSQL server is listening (default is 5432).
- database** The specific database you want to connect to.
- options** Additional connection options such as SSL mode.

Understanding Vector Databases:

Let's take an example to understand vector databases more effectively, consider the following

Harkirat lives in India ⇒ [1, 2, 2, 2, 2, 2]

Harkirat is from Chandigarh ⇒ [1, 2, 2, 2, 3]

Harkirat has been living in India, Chandigarh ⇒ [1, 2, 2, 2, 2, 3]

```
The world is round ⇒ [1, 2, 10001, 1001, 001001]
```

```
Pacman is such a good game ⇒ [100, 10001, 20020, 1-001, 100]
```

In the examples provided, the vectors for statements about "Harkirat" and "India" have similar coordinates because they contain similar words or concepts. The presence of identical numbers in different vectors indicates that those vectors represent statements with shared words or meanings. For instance, the repeated '2' in the vectors might indicate common words or a common structure in the statements, while unique identifiers like '3001' for "Harkirat" or '3' for "Chandigarh" show up in vectors representing statements about those specific entities.

Vector databases leverage this property to perform efficient similarity searches. When a query vector is provided, the database can quickly find other vectors with similar coordinates, which correspond to records containing similar words or concepts, thus retrieving relevant information based on semantic similarity.

Interact with PostgreSQL

When working with PostgreSQL databases, especially in the context of application development, it's common to use libraries or tools that facilitate connecting to, interacting with, and visualizing the data within these databases. Two such tools are `psql` and `pg`

Each serves different purposes and fits into different parts of the development workflow.

1. psql

- What is `psql`?
 - `psql` is a command-line interface (CLI) tool that allows you to interact with a PostgreSQL database server. It provides a terminal-based front-end to PostgreSQL, enabling users to execute SQL queries directly, inspect the database schema, and manage the database.
- How to Connect to Your Database with `psql`?
 - To connect to a PostgreSQL database using `psql`, you can use a command in the following format:

```
psql -h [host] -d [database] -U [user]
```

- For example, based on the provided information:

```
psql -h p-broken-frost-69135494.us-east-2.aws.neon.tech -d database1 -U 100xdevs
```

- Here, `h` specifies the host of the database server, `d` specifies the name of the database you want to connect to, and `u` specifies the username.

- Installation and Usage:

- `psql` comes bundled with the PostgreSQL installation package. If you have PostgreSQL installed on your system, you likely already have `psql` available.
- While not necessary for direct application development, `psql` is invaluable for database administration, debugging, and manual data inspection.

2. pg (node-postgres)

- What is pg?

- `pg`, also known as node-postgres, is a collection of Node.js modules for interfacing with your PostgreSQL database. It is non-blocking and designed specifically for use with Node.js. Similar to how `mongoose` is used for MongoDB, `pg` allows for interaction with PostgreSQL databases within a Node.js application.

- How to Use pg in Your Application?

- To use `pg` in your Node.js application, you first need to install it via npm or yarn:

```
npm install pg
yarn add pg
```

- After installing, you can connect to your PostgreSQL database using the `pg` library by creating a client and connecting it with your database's connection string:

```
javascriptconst { Client } = require('pg');

const client = new Client({
  connectionString: 'YourDatabaseConnectionStringHere'
});

client.connect();
```

- You can then use this client to execute queries against your database.

- Why Use pg?

- `pg` provides a programmatic way to connect to and interact with your PostgreSQL database directly from your Node.js application. It supports features like connection pooling, transactions, and streaming results. It's essential for building applications that require data persistence in a PostgreSQL database.

Creating a table schema

Creating a table and defining its schema is a fundamental step in working with SQL databases. This process involves specifying the structure of the data each table will hold. Let's break down the process and the components of a SQL statement used to create a table in PostgreSQL, using the `table` as an example.

```
users
```

Creating a Table in SQL

1. Initiate Table Creation

- **Syntax:** `CREATE TABLE users`
- **Description:** This command starts the creation of a new table named `users` in the database.

2. Define Columns and Constraints

- **Column Definition:** Each column in the table is defined with a name, data type, and possibly one or more constraints.
 - **`id SERIAL PRIMARY KEY :`**
 - `id` : Column name, typically used as a unique identifier for each row.
 - `SERIAL` : A PostgreSQL data type for auto-incrementing integers, ensuring each row has a unique ID.
 - `PRIMARY KEY` : A constraint that specifies the `id` column as the primary key, ensuring uniqueness and non-null values.
 - **`username VARCHAR(50) UNIQUE NOT NULL :`**
 - `username` : Column for storing the user's username.
 - `VARCHAR(50)` : Data type specifying a variable-length string of up to 50 characters.
 - `UNIQUE` : Ensures all values in this column are unique.
 - `NOT NULL` : Prevents null values, requiring every row to have a username.
 - **`email VARCHAR(255) UNIQUE NOT NULL :`**
 - Similar to `username`, but intended for the user's email address, allowing up to 255 characters.
 - **`password VARCHAR(255) NOT NULL :`**
 - Stores the user's password with the same data type as `email`, but without the `UNIQUE` constraint, as passwords can be non-unique.
 - **`created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP :`**
 - `created_at` : Column for storing the timestamp of when the user was created.
 - `TIMESTAMP WITH TIME ZONE` : Stores both a timestamp and a time zone.
 - `DEFAULT CURRENT_TIMESTAMP` : Automatically sets the value to the current timestamp when a new row is inserted.

Practical Steps

1. Execute the **CREATE TABLE Command**: Use the provided SQL statement to create the `users` table in your PostgreSQL database.

```
sqlCREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

2. Verify Table Creation: After running the command, you can verify the creation of the table by executing `\dt;` in the `psql` command-line interface. This command lists all tables in the current database, and you should see the `users` table listed among them.

Interacting with the database

Interacting with a database typically involves performing four fundamental operations, collectively known as CRUD operations: Create, Read, Update, and Delete. In the context of SQL databases, these operations are executed using SQL commands. Let's elaborate on how each of these operations is carried out in PostgreSQL, using the table as an example `users`

1. INSERT (Create)

- **Purpose:** To add new records to a table.
- **SQL Command:**

```
sqlINSERT INTO users (username, email, password)
VALUES ('username_here', 'user@example.com', 'user_password');
```

- **Explanation:** This command inserts a new row into the `users` table with values for `username`, `email`, and `password`. The `id` column is not specified because it's an auto-incrementing field (`SERIAL`), meaning PostgreSQL will automatically assign a unique `id` to each new row.

2. UPDATE

- **Purpose:** To modify existing records in a table.
- **SQL Command:**

```
sqlUPDATE users
SET password = 'new_password'
WHERE email = 'user@example.com';
```

- **Explanation:** This command updates the `password` for the user with the specified `email`. The `WHERE` clause is crucial as it determines which records are updated. Without it, all records in the table would be updated.

3. DELETE

- **Purpose:** To remove records from a table.
- **SQL Command:**

```
sqlDELETE FROM users  
WHERE id = 1;
```

- **Explanation:** This command deletes the row from the `users` table where the `id` is 1. Like with `UPDATE`, the `WHERE` clause specifies which records to delete. Omitting the `WHERE` clause would result in deleting all records in the table, which is rarely intended.

4. SELECT (Read)

- **Purpose:** To retrieve records from a table.
- **SQL Command:**

```
sqlSELECT * FROM users  
WHERE id = 1;
```

- **Explanation:** This command selects all columns (`*`) for the row(s) in the `users` table where the `id` is 1. The `SELECT` statement is highly versatile, allowing for complex queries with various conditions, sorting, and joining multiple tables.

Practical Tips

- **Running Commands:** If you have `psql` installed locally, you can run these commands directly in your terminal to interact with your PostgreSQL database. This is a great way to practice and see the immediate effect of each operation.
- **Using pg Library:** For application development, especially in a Node.js environment, you'll eventually use the `pg` library to execute these operations programmatically. This allows your application to dynamically interact with the database based on user input or other logic.

Understanding and being able to execute these four basic database operations is foundational for working with SQL databases. Whether you're manually testing commands in `psql` or integrating database operations into an application with the `pg` library, these CRUD operations form the basis of data manipulation and retrieval in relational databases.

Database Operations

To perform database operations from a Node.js application, you can use the `pg` library, a non-blocking PostgreSQL client for Node.js. This library allows you to connect to your PostgreSQL

database and execute queries programmatically. Here's a step-by-step guide on how to connect to the database and perform basic operations using the library [pg](#)

Installing the [pg](#) Library

First, you need to install the [pg](#) library in your Node.js project. Run the following command in your project directory:

```
bashnpm install pg
```

Connecting to the Database

To connect to your PostgreSQL database, you need to create a [Client](#) instance with your database connection details and then call the [connect](#) method.

```
javascriptimport { Client } from 'pg';

const client = new Client({
  host: 'my.database-server.com',
  port: 5334,
  database: 'database-name',
  user: 'database-user',
  password: 'secretpassword!!',
});

client.connect();
```

Querying the Database

Once connected, you can execute queries using the [query](#) method of the [Client](#) instance. Here's how to perform the four basic database operations:

INSERT

To insert data into your table:

```
javascriptconst insertResult = await client.query(
  "INSERT INTO users (username, email, password) VALUES ('username_here', 'user@example.com', '')");
console.log(insertResult);
```



UPDATE

To update existing data:

```
javascriptconst updateResult = await client.query(
  "UPDATE users SET password = 'new_password' WHERE email = 'user@example.com';");
console.log(updateResult);
```

DELETE

To delete data from your table:

```
javascriptconst deleteResult = await client.query(
  "DELETE FROM users WHERE id = 1;"
);
console.log(deleteResult);
```

SELECT

To retrieve data:

```
javascriptconst selectResult = await client.query(
  "SELECT * FROM users WHERE id = 1;"
);
console.log(selectResult.rows);
```

Creating a Table

You can also use the `pg` library to create tables. Here's an example function that creates a `users` table:

```
javascripasync function createUsersTable() {
  await client.connect();
  const result = await client.query(` 
    CREATE TABLE users (
      id SERIAL PRIMARY KEY,
      username VARCHAR(50) UNIQUE NOT NULL,
      email VARCHAR(255) UNIQUE NOT NULL,
      password VARCHAR(255) NOT NULL,
      created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
    );
  `);
  console.log(result);
  await client.end();
}

createUsersTable();
```

Conclusion

The `pg` library provides a powerful interface for interacting with PostgreSQL databases from Node.js applications. By following the steps outlined above, you can connect to your database, execute queries, and manage your data effectively. For more advanced use cases, refer to the [pg library documentation](#)

Creating a Simple Node.js App

Creating a simple Node.js application that interacts with a PostgreSQL database using TypeScript involves several steps, from initializing the project to writing secure database interaction functions. Here's a concise guide to setting up your project and implementing basic database operations securely.

Step 1: Initialize a TypeScript Project

1. Create a new directory for your project and navigate into it.

2. Initialize a new npm project:

```
npm init -y
```

3. Initialize a TypeScript project:

```
npx tsc --init
```

4. Configure TypeScript by editing `tsconfig.json`:

- Set `"rootDir": "./src"` to specify the source directory.
- Set `"outDir": "./dist"` to specify the output directory for compiled JavaScript files.

Step 2: Install Dependencies

1. Install the `pg` library to interact with PostgreSQL:

```
npm install pg
```

2. Install TypeScript types for `pg`:

```
npm install @types/pg --save-dev
```

Step 3: Create a Simple Node.js App

Insert Data Function

Create a function to insert data into a table securely, using parameterized queries to prevent SQL injection.

```
import { Client } from 'pg';

// Async function to insert data into a table
async function insertData(username: string, email: string, password: string) {
  const client = new Client({
    host: 'localhost',
    port: 5432,
    database: 'postgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });

  try {
    await client.connect();
    const insertQuery = "INSERT INTO users (username, email, password) VALUES ($1, $2, $3)";
    const values = [username, email, password];
    const res = await client.query(insertQuery, values);
    console.log('Insertion success:', res);
  } catch (err) {
    console.error('Error during the insertion:', err);
  } finally {
    await client.end();
  }
}
```

```
// Example usageinsertData('username5', 'user5@example.com', 'user_password').catch(console.error)
```

Fetch Data Function

Implement a function to fetch user data from the database given an email, using parameterized queries for security.

```
import { Client } from 'pg';

// Async function to fetch user data from the database given an emailasync function getUser(email) {
  const client = new Client({
    host: 'localhost',
    port: 5432,
    database: 'postgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });

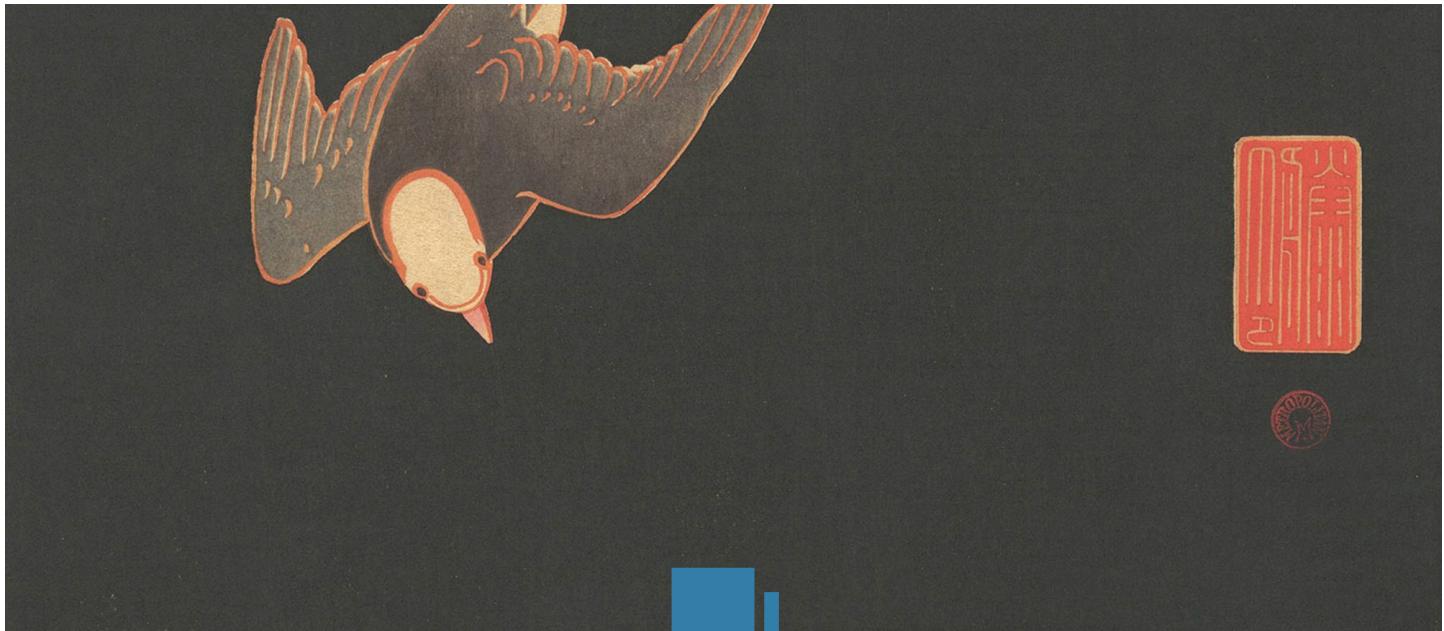
  try {
    await client.connect();
    const query = 'SELECT * FROM users WHERE email = $1';
    const values = [email];
    const result = await client.query(query, values);

    if (result.rows.length > 0) {
      console.log('User found:', result.rows[0]);
      return result.rows[0];
    } else {
      console.log('No user found with the given email.');
      return null;
    }
  } catch (err) {
    console.error('Error during fetching user:', err);
    throw err;
  } finally {
    await client.end();
  }
}

// Example usagegetUser('user5@example.com').catch(console.error);
```



This demonstrates how to set up a TypeScript project for a Node.js application that interacts with a PostgreSQL database using the `pg` library. By using parameterized queries, you can securely insert and fetch data, protecting your application from SQL injection attacks.



Week 10.2

Understanding Prisma

In today's lecture, Harkirat provides an insightful overview of [Prisma](#), discussing the role of [Object-Relational Mapping \(ORM\)](#) systems and how Prisma serves as a powerful ORM for modern application development.

The session later explores the use of Prisma with various databases, the functionality of the [Prisma Client](#), and culminated in a hands-on demonstration of [creating a first application using Prisma](#) showcasing its ease of use and versatility in managing database operations.

Understanding Prisma

ORMs

[What are ORMs?](#)

[Official Definition](#)

[Simplified Definition](#)

[Prisma as an ORM](#)

Why ORMs?

[1. Simpler Syntax](#)

[2. Database Abstraction](#)

[3. Type Safety/Auto-completion](#)

[4. Automatic Migrations](#)

Prisma

[1. Data Model](#)

[2. Automated Migrations](#)

[3. Type Safety](#)

[4. Auto-Completion](#)

[Installing Prisma](#)

[Step 1: Initialize an Empty Node.js Project](#)

[Step 2: Add Dependencies](#)

[Step 3: Initialize TypeScript](#)

[Step 4: Configure TypeScript](#)

[Step 5: Initialize a Fresh Prisma Project](#)

[Next Steps](#)

[Selecting Your Database](#)

[Step 1: Update schema.prisma for Your Database](#)

[For PostgreSQL](#)

[For MySQL](#)

[For MongoDB \(Preview\)](#)

[Step 2: Set Your Database Connection URL](#)

[Example .env Content for PostgreSQL](#)

[Example .env Content for MySQL](#)

[Example .env Content for MongoDB](#)

[Step 3: Install the Prisma VSCode Extension](#)

[Defining Models](#)

[Step 1: Define Your Data Model in schema.prisma](#)

[Step 2: Generate Migrations](#)

[Step 3: Inspect the Migrations Folder](#)

[Exploring Your Databases](#)

[Step 1: Connect to Your Database](#)

[Step 2: List Tables](#)

[Step 3: Describe Table Structure](#)

[Step 4: Query Data](#)

[Step 5: Exit psql](#)

[Prisma Client](#)

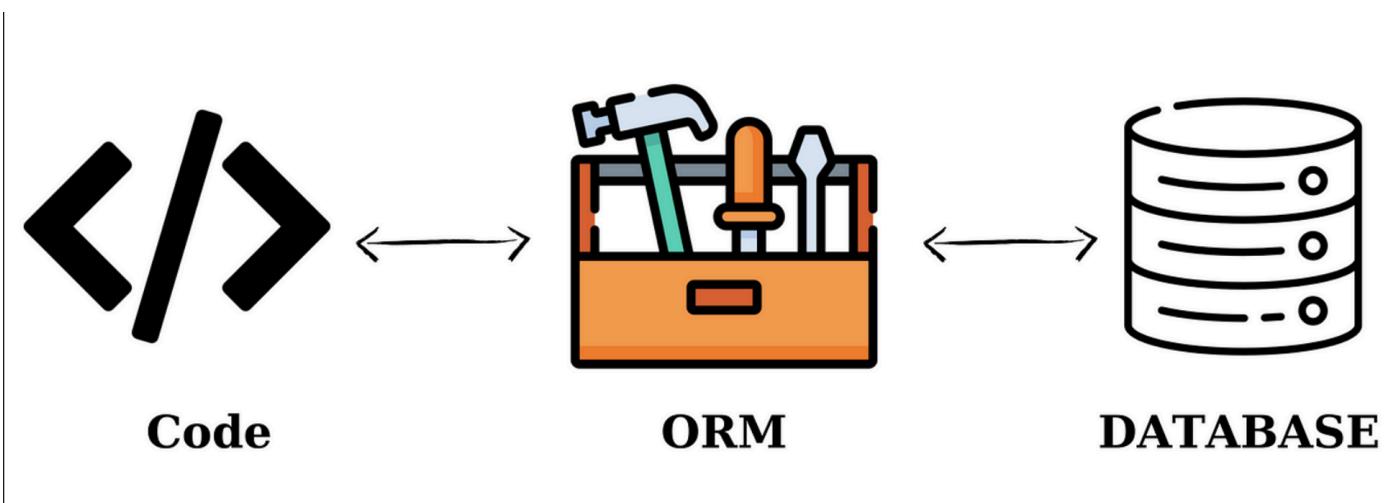
[What is a Prisma Client?](#)

[How Does the Prisma Client Work?](#)

[How to Generate the Prisma Client?](#)[Using the Generated Prisma Client](#)[Creating Your First Application](#)[1\] Insert](#)[2\] Update](#)[3\] Get a User's Detail](#)

ORMs

Object-Relational Mapping (ORM) is a crucial concept in modern software development, particularly when dealing with databases in object-oriented programming languages. Prisma is an ORM that exemplifies the use of this technique. Here's an elaboration on ORMs, with a focus on how Prisma fits into this context:



What are ORMs?

Official Definition

- **ORM:** Object-Relational Mapping is a programming technique for converting data between incompatible systems using object-oriented programming languages. It creates a "virtual object database" that developers can interact with using their programming language instead of direct database queries.
- **Abstraction:** ORMs abstract the complexities of the database, allowing developers to work with database records as if they were objects in their code. This includes handling CRUD operations (Create, Read, Update, Delete) and managing database connections and transactions.

Simplified Definition

- **Ease of Use:** ORMs simplify database interactions by letting developers use the syntax and paradigms of their programming language rather than writing SQL queries. This can make code more readable and maintainable.

Prisma as an ORM

Prisma is a next-generation ORM that takes the concept of ORMs further by providing additional tools and features that enhance the developer experience:

- **Schema Definition:** Prisma uses a declarative Prisma schema to define the application's data model. This schema is used to generate a Prisma Client that provides type-safe database access.
- **Migrations:** Prisma Migrate allows developers to define and perform database schema migrations in a controlled and versioned manner.
- **Type Safety:** Prisma ensures type safety by generating a client that is tailored to the schema, reducing the risk of runtime errors due to mismatched data types.
- **Query Building:** Prisma Client provides a fluent API for building queries, which can be more intuitive than writing raw SQL, especially for complex queries.
- **Performance:** Prisma is designed to be performant and efficient, with a focus on minimizing the overhead typically associated with ORMs.

ORMs, including Prisma, offer a high-level abstraction over database interactions, making it easier for developers to work with data in the context of their applications.

Why ORMs?

Object-Relational Mapping (ORM) frameworks provide a bridge between the object-oriented world of application development and the relational world of databases. They offer several advantages that make them an attractive choice for developers. Let's delve into these benefits with explanations and code snippets to illustrate their impact.

1. Simpler Syntax

ORMs allow developers to work with high-level programming constructs instead of writing SQL queries directly. This means you can manipulate database entries using objects and methods in your programming language.

Without ORM: SQL Query

```
INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com');
```

With ORM: Object Manipulation (Example in JavaScript using Prisma)

```
await prisma.users.create({
  data: {
    name: 'John Doe',
    email: 'john@example.com',
  },
});
```

2. Database Abstraction

ORMs provide a unified API to interact with different databases, making it easier to switch databases if needed without rewriting your data access layer.

Prisma Example: The same Prisma client code works across different databases. Switching from PostgreSQL to MySQL, for instance, primarily requires changes in the configuration, not in the code that interacts with the database.

Prisma Schema for PostgreSQL

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

Prisma Schema for MySQL

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}
```

3. Type Safety/Auto-completion

Modern ORMs, especially those used in statically typed languages or with TypeScript support, offer type safety and auto-completion, reducing runtime errors and improving developer productivity.

TypeScript Example with Prisma: When you query the database, the Prisma client provides auto-completion for table names and columns, and the returned data is automatically typed.

```
// TypeScript understands the structure of the expected result,
// providing auto-completion and type checking
const user = await prisma.user.findUnique({
  where: {
    email: 'john@example.com',
```

```
},  
});
```

4. Automatic Migrations

ORMs can automate the process of generating and applying database schema migrations, making it easier to evolve your database schema as your application grows.

Prisma Migration Example: Prisma Migrate generates SQL migration files for your schema changes, which can be applied to update the database schema.

Generate Migration

```
npx prisma migrate dev --name add_phone_number
```

This command might generate a SQL file similar to:

```
-- Migration SQL generated by Prisma Migrate  
ALTER TABLE "users"  
ADD COLUMN "phone_number" VARCHAR(15);
```

Apply Migration

Applying migrations is handled by Prisma Migrate when you run the above command, keeping your database schema in sync with your Prisma schema.

Prisma

Prisma is a next-generation ORM (Object-Relational Mapping) tool for Node.js and TypeScript applications. It simplifies database workflows by providing a robust set of features that enhance developer productivity and code quality. Let's delve into the core components that make Prisma a powerful tool for modern application development.

Next-generation Node.js and TypeScript ORM

Prisma unlocks a new level of **developer experience** when working with databases thanks to its intuitive **data model**, **automated migrations**, **type-safety** & **auto-completion**.

1. Data Model

Prisma uses a Prisma Schema file to define the data model of your application. This schema acts as a single source of truth for your database structure, including tables, columns, relationships, and more. The Prisma Schema Language (PSL) is intuitive yet powerful, allowing you to define your database schema in a clear and concise manner.

Example Prisma Schema:

```
// schema.prisma

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id      Int      @id @default(autoincrement())
  name    String
  email   String   @unique
  posts   Post[]
}

model Post {
  id      Int      @id @default(autoincrement())
  title  String
  content String?
  author  User     @relation(fields: [authorId], references: [id])
  authorId Int
}
```

This schema defines two models, `User` and `Post`, representing tables in the database. It specifies fields for each table, their types, and the relationship between users and posts.

2. Automated Migrations

Prisma Migrate is a feature that automatically generates and runs database migrations based on changes to your Prisma schema. This means that when you modify your data model, Prisma Migrate can automatically update your database schema to match, without the need for manual SQL migration scripts.

Generating and Applying Migrations:

```
npx prisma migrate dev --name init
```

This command generates SQL migration files for the current state of your Prisma schema and applies them to your database, creating or altering tables and relationships as defined.

3. Type Safety

Prisma Client is a type-safe database client generated based on your Prisma schema. This means that every database query you write is checked against the schema, significantly reducing the risk of runtime errors due to data type mismatches. The client provides full auto-completion and type checking in supported editors, making it easier to write and refactor code confidently.

Example Usage of Prisma Client:

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

async function main() {
  const newUser = await prisma.user.create({
    data: {
      name: 'Alice',
      email: 'alice@example.com',
    },
  });
  console.log(newUser);
}

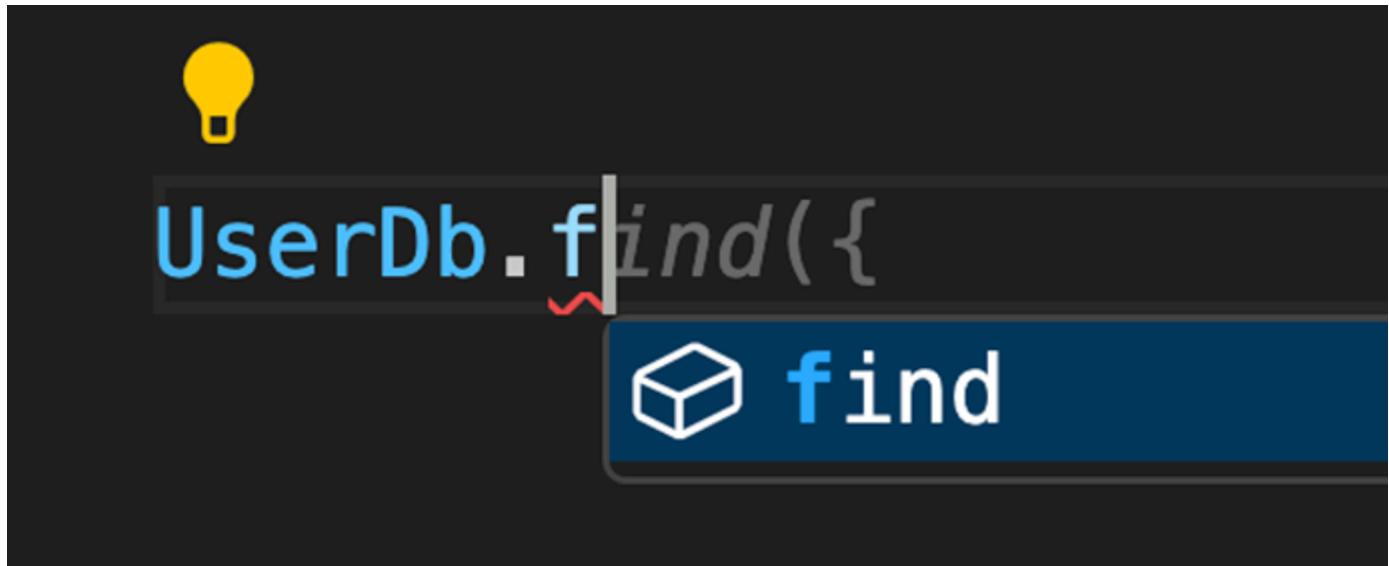
main();
```

4. Auto-Completion

Prisma's integration with code editors (such as VSCode) provides auto-completion for model fields, operations, and even potential query results. This feature not only speeds up development but also helps prevent errors by ensuring that your queries align with the defined schema.

Auto-Completion Example:

When writing queries with Prisma Client, your editor can suggest model fields, available methods, and more, based on the Prisma schema.



Installing Prisma

Creating a simple TODO app with Prisma in a Node.js environment involves several steps, from initializing your Node.js project to setting up Prisma. Here's a detailed guide to get you started:

Step 1: Initialize an Empty Node.js Project

First, create a new directory for your project and navigate into it. Then, initialize a new Node.js project:

```
mkdir todo-app
cd todo-app
npm init -y
```

This command creates a `package.json` file with default values.

Step 2: Add Dependencies

Install Prisma, TypeScript, ts-node (for running TypeScript files directly), and @types/node (for Node.js type definitions) as development dependencies:

```
npm install prisma typescript ts-node @types/node --save-dev
```

Step 3: Initialize TypeScript

Set up TypeScript in your project:

```
npx tsc --init
```

This command creates a `tsconfig.json` file, which configures TypeScript options.

Step 4: Configure TypeScript

Edit the `tsconfig.json` file to specify the root directory and the output directory for the compiled JavaScript files:

- Change `"rootDir"` to `".src"`. This tells TypeScript to look for `.ts` files in the `src` directory.
- Change `"outDir"` to `".dist"`. Compiled `.js` files will be output to the `dist` directory.

Your `tsconfig.json` should include these changes like so:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

Step 5: Initialize a Fresh Prisma Project

Initialize Prisma in your project:

```
npx prisma init
```

```
✓ Your Prisma schema was created at prisma/schema.prisma
You can now open it in your favorite editor.

Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing database. If your database has no tables yet, read https://pris.ly/d/getting-started
2. Set the provider of the datasource block in schema.prisma to match your database: postgresql, mysql, sqlite, sqlserver, mongodb or cockroachdb.
3. Run prisma db pull to turn your database schema into a Prisma schema.
4. Run prisma generate to generate the Prisma Client. You can then start querying your database.

More information in our documentation:
https://pris.ly/d/getting-started
```

This command performs several actions:

- Creates a `prisma` directory with a `schema.prisma` file inside. This is where you define your database schema.
- Creates a `.env` file in the root of your project. This is where you set environment variables, such as your database connection string.

Next Steps

After initializing Prisma, you can proceed to define your database schema in the `schema.prisma` file. For a TODO app, you might define a `Todo` model like this:

```
// prisma/schema.prisma

datasource db {
  provider = "postgresql" // Or another database provider like "mysql", "sqlite", etc.
  url      = env("DATABASE_URL")
}

model Todo {
  id      Int      @id @default(autoincrement())
  title   String
  completed Boolean @default(false)
}
```

Remember to replace `"postgresql"` with your database provider and set your database connection string in the `.env` file:

```
DATABASE_URL="postgresql://user:password@localhost:5432/mydb?schema=public"
```

Finally, to create the `Todo` table in your database, run Prisma Migrate:

```
npx prisma migrate dev --name init
```

This command generates and applies a migration based on your schema changes.

You can start adding functionality to your TODO app, such as creating, reading, updating, and deleting TODO items using Prisma Client in your application code.

Selecting Your Database

Prisma supports a variety of databases, including relational databases like MySQL and PostgreSQL, as well as the document-oriented database MongoDB (in Preview). Selecting and configuring your

database with Prisma involves updating the `schema.prisma` file and setting the database connection URL in your environment variables. Here's how you can do it:

Step 1: Update `schema.prisma` for Your Database

The `schema.prisma` file contains a `datasource` block where you specify your database connection. You need to update the `provider` field according to the database you want to use.

For PostgreSQL

```
datasource db {  
  provider = "postgresql"  
  url      = env("DATABASE_URL")  
}
```

For MySQL

```
datasource db {  
  provider = "mysql"  
  url      = env("DATABASE_URL")  
}
```

For MongoDB (Preview)

```
datasource db {  
  provider = "mongodb"  
  url      = env("DATABASE_URL")  
}
```

Step 2: Set Your Database Connection URL

The `url` field in the `datasource` block references an environment variable `DATABASE_URL` where you should set your database connection string. This is done in the `.env` file in the root of your project.

Example `.env` Content for PostgreSQL

```
DATABASE_URL="postgresql://username:password@localhost:5432/mydatabase"
```

Replace `username`, `password`, `localhost`, `5432`, and `mydatabase` with your actual database credentials and details.

Example `.env` Content for MySQL

```
DATABASE_URL="mysql://username:password@localhost:3306/mydatabase"
```

Example `.env` Content for MongoDB

```
DATABASE_URL="mongodb+srv://username:password@cluster0.example.mongodb.net/mydatabase"
```

Step 3: Install the Prisma VSCode Extension

To enhance your development experience with Prisma, it's highly recommended to install the Prisma VSCode extension. This extension provides features like syntax highlighting, formatting, auto-completion, and a visual overview of your Prisma schema.

You can install the Prisma extension directly from the Visual Studio Code Marketplace:

1. Open VSCode.
2. Go to the Extensions view by clicking on the square icon on the sidebar or pressing `Ctrl+Shift+X`
3. Search for "Prisma".
4. Find the Prisma extension by Prisma and click "Install".

Remember, when using MongoDB with Prisma, it's currently in Preview, so it's a good idea to keep an eye on the official Prisma documentation for any updates or changes.

Defining Models

Defining your data model in Prisma involves specifying the structure of your database tables and their relationships directly in the `schema.prisma` file. This schema acts as a blueprint for your database, allowing Prisma to generate the necessary code to interact with your data in a type-safe manner. Let's walk through the process of defining a data model for a simple application with Users and Todos tables, and then generating the corresponding migrations to update your database schema.

Step 1: Define Your Data Model in `schema.prisma`

To add a Users and a Todos table to your application, you'll define two models in your `schema.prisma` file: `User` and `Todo`. At this stage, we won't worry about defining relationships between these tables.

```
// schema.prisma

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql" // Or your database provider of choice
  url      = env("DATABASE_URL")
}

model User {
  id      Int      @id @default(autoincrement())
  username String  @unique
  password String
  firstName String
  lastName String
}

model Todo {
  id      Int      @id @default(autoincrement())
  title   String
  description String
  done    Boolean @default(false)
  userId  Int      // This will later be used to establish a relationship
}
```

In this schema:

- Each `model` keyword defines a table in your database.
- The fields within each model represent columns in the table.
- Attributes like `@id`, `@default(autoincrement())`, and `@unique` specify column constraints and behaviors.

Step 2: Generate Migrations

After defining your data model, the next step is to generate migrations to create these tables in your database. Prisma Migrate translates changes in your Prisma schema into SQL migration files, which are then applied to your database to update its schema.

Run the following command to generate and apply migrations:

```
npx prisma migrate dev --name initialize_schema
```

This command does a few things:

- It generates SQL migration files that represent the schema changes (in this case, creating the

User and Todo tables).

- It applies these migrations to your database, creating the tables.
- It generates or updates the Prisma Client, which you use to interact with your database in your application code.

Step 3: Inspect the Migrations Folder

After running the migrations, you can find them in the `prisma/migrations` folder in your project directory. Each migration is stored in a separate folder, named with a timestamp and the name you provided (`initialize_schema` in this case). Inside, you'll find:

- A `migration.sql` file containing the SQL commands that were run against your database.
- A `README.md` file with information about the migration.

Inspecting these files can give you insights into the exact changes made to your database schema.

By defining your data model in the `schema.prisma` file and using Prisma Migrate, you've successfully created a Users and a Todos table in your database without manually writing any SQL.

This process not only simplifies database schema management but also ensures that your application's data model is version-controlled and easily reproducible across different environments.

Exploring Your Databases

Exploring your database with `psql` after Prisma has created tables for you is a straightforward process. `psql` is a command-line interface for interacting with PostgreSQL that allows you to execute queries, view table structures, and manage your database. Here's how you can use `psql` to explore the tables created by Prisma:

Step 1: Connect to Your Database

Open your terminal and use the `psql` command to connect to your PostgreSQL database. Replace `localhost`, `postgres`, and `postgres` with your database's host, database name, and user, respectively, if they are different.

```
psql -h localhost -d postgres -U postgres
```

You might be prompted to enter the password for the PostgreSQL user.

Step 2: List Tables

Once connected, you can list all the tables in your database using the `\dt` command.

```
\dt
```

This command will display a list of tables, including those created by Prisma. You should see the `User` and `Todo` tables listed if you followed the previous steps to define your data model and run migrations.

Step 3: Describe Table Structure

To get detailed information about the structure of a specific table, use the `\d` command followed by the table name. For example, to describe the `User` table:

```
\d "User"
```

This will show you the columns, their data types, and any constraints or indexes associated with the table.

Step 4: Query Data

You can also execute SQL queries directly to retrieve data from your tables. For instance, to select all records from the `User` table:

```
SELECT * FROM "User";
```

Step 5: Exit `psql`

When you're done exploring, you can exit `psql` by typing `\q` and pressing Enter.

```
\q
```

Using `psql` to explore your database gives you a direct view of the underlying structure and data. It's a powerful tool for database administration and can be particularly useful for verifying the results of ORM operations, such as those performed by Prisma. Whether you're developing, debugging, or simply curious about the state of your database, `psql` provides the necessary commands to interact with and inspect your PostgreSQL database effectively.

Prisma Client

What is a Prisma Client?

The Prisma Client is an auto-generated and type-safe query builder that's tailored to your data model. It provides a fluent API that lets you compose queries in a way that is both intuitive and close to natural language. The client abstracts away the SQL layer, offering methods that correspond to various database operations, such as creating, reading, updating, and deleting records.

How Does the Prisma Client Work?

When you use Prisma Client in your application, you write code like this:

```
const newUser = await prisma.user.create({  
  data: {  
    email: "harkirat@gmail.com",  
  },  
});
```

Under the hood, Prisma Client converts this operation into an SQL query similar to:

```
INSERT INTO users (email) VALUES ('harkirat@gmail.com');
```

This conversion is handled automatically, so you don't need to write SQL queries manually.

How to Generate the Prisma Client?

After defining your data model in the `schema.prisma` file, you can generate the Prisma Client by running the following command in your terminal:

```
npx prisma generate
```

This command reads your Prisma schema and generates the client code accordingly. The generated client includes all the necessary functions to interact with your database based on the models you've defined.

Using the Generated Prisma Client

Once generated, you can import and use the Prisma Client in your Node.js application to perform database operations. Here's an example of how you might use the client in an async function:

```
import { PrismaClient } from '@prisma/client';
```

```
const prisma = new PrismaClient();

async function main() {
  const newUser = await prisma.user.create({
    data: {
      email: "harkirat@gmail.com",
    },
  });
  console.log(newUser);
}

main()
  .catch(e => {
    throw e;
  })
  .finally(async () => {
    await prisma.$disconnect();
  });

```

In this example, we're using the `create` method on the `user` model to insert a new user into the database. The Prisma Client provides similar methods for other CRUD operations and supports complex queries, including filtering, sorting, and joining data across tables.

Creating Your First Application

1] Insert

Creating a function to insert data into the `Users` table using Prisma in a TypeScript application

The provided solution defines an asynchronous function `insertUser` that takes four parameters: `username`, `password`, `firstName`, and `lastName`. This function uses the Prisma Client to insert a new user into the `Users` table with the provided details.

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function insertUser(username: string, password: string, firstName: string, lastName: string) {
  const res = await prisma.user.create({
    data: {
      username,
      password,
      firstName,
      lastName
    }
  });
}
```

```

    console.log(res);
}

insertUser("admin1", "123456", "harkirat", "singh");

```

Here's a breakdown of the key parts of this function:

- **Prisma Client Initialization:** `const prisma = new PrismaClient();` creates a new instance of the Prisma Client. This instance is used to perform operations on your database.
- **The `insertUser` Function:** This is an asynchronous function, indicated by the `async` keyword. It's designed to insert a new user into the database.
- **Inserting Data:** `prisma.user.create()` is a method provided by the Prisma Client to create (or insert) a new record in the `user` table. The method takes an object with a `data` property, which itself is an object containing the fields to be inserted into the table.
- **Awaiting the Promise:** The `await` keyword is used to wait for the promise returned by `prisma.user.create()` to resolve. This is necessary because database operations are asynchronous.
- **Logging the Result:** The result of the insert operation (the newly created user record) is stored in the `res` variable and then logged to the console.
- **Executing the Function:** Finally, `insertUser("admin1", "123456", "harkirat", "singh");` calls the function with sample data. In a real application, you would likely call this function in response to user input, such as a form submission.

2] Update

Updating data in the `Users` table using Prisma in a TypeScript application involves specifying the criteria for selecting the user to update and providing the new data for the selected fields.

The solution defines an asynchronous function `updateUser` that takes a `username` and an object containing the new `firstName` and `lastName` values. This function uses the Prisma Client to update the specified user in the database.

```

import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

interface UpdateParams {
  firstName: string;
  lastName: string;
}

async function updateUser(username: string, {

```

```
firstName,  
lastName  
}: UpdateParams) {  
  const res = await prisma.user.update({  
    where: { username },  
    data: {  
      firstName,  
      lastName  
    }  
  });  
  console.log(res);  
}  
  
updateUser("admin1", {  
  firstName: "new name",  
  lastName: "singh"  
});
```

Here's a breakdown of the key components:

- **Prisma Client Initialization:** `const prisma = new PrismaClient();` creates a new instance of the Prisma Client, which is used to interact with your database.
- **The `UpdateParams` Interface:** This TypeScript interface defines the shape of the object expected by the `updateUser` function for the update operation. It specifies that both `firstName` and `lastName` should be strings.
- **The `updateUser` Function:** This asynchronous function is designed to update a user's `firstName` and `lastName` in the database. It takes a `username` to identify the user to update and an `UpdateParams` object containing the new values.
- **Updating Data:** `prisma.user.update()` is a method provided by the Prisma Client to update a record in the `user` table. It requires two main arguments:
 - `where` : An object specifying the criteria to find the record to update. In this case, it's the `username` of the user.
 - `data` : An object containing the fields to update and their new values.
- **Awaiting the Promise:** The `await` keyword pauses execution until the promise returned by `prisma.user.update()` is resolved, ensuring the update operation completes before proceeding.
- **Logging the Result:** The result of the update operation (the updated user record) is stored in the `res` variable and then logged to the console.
- **Executing the Function:** The function is called with a sample `username` and new values for `firstName` and `lastName`. In a real application, these values would likely come from user input.

3] Get a User's Detail

Fetching a user's details from the database based on their username can be done efficiently using Prisma Client.

The solution defines an asynchronous function `getUser` that takes a `username` as an argument. This function uses the Prisma Client to find the first user that matches the given `username`.

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getUser(username: string) {
  const user = await prisma.user.findFirst({
    where: {
      username: username
    }
  });
  console.log(user);
}

getUser("admin1");
```

Here's a breakdown of the key components:

- **Prisma Client Initialization:** `const prisma = new PrismaClient();` creates a new instance of the Prisma Client, which is used to interact with your database.
- **The `getUser` Function:** This asynchronous function is designed to fetch a user's details from the database. It takes a `username` to identify the user.
- **Fetching Data:** `prisma.user.findFirst()` is a method provided by the Prisma Client to retrieve the first record that matches the given criteria from the `user` table. It requires an object with a `where` clause:
 - `where` : An object specifying the criteria to find the record. In this case, it's looking for a user with a matching `username` .
- **Awaiting the Promise:** The `await` keyword is used to wait for the promise returned by `prisma.user.findFirst()` to resolve. This is necessary because database operations are asynchronous.
- **Logging the Result:** The result of the fetch operation (the user record) is stored in the `user` variable and then logged to the console.
- **Executing the Function:** The function is called with a sample `username` of "admin1". In a real application, this value would likely come from user input, such as a login form or a search query.



Week 12.2

Advance Typescript APIs

In today's lecture, Harkirat delves into advanced TypeScript utility types such as `Pick`, `Partial`, `Readonly`, `Record`, `Exclude` and the `Map` type, providing insights into their practical applications. Additionally, the lecture covered type inference in Zod, a TypeScript-first schema declaration and validation library, highlighting how these advanced features can enhance type safety and developer productivity in TypeScript projects.

Advance Typescript APIs

Prerequisites

Recap Setup Procedure

1] Pick

Understanding Pick

Example Usage of Pick

Benefits of Using Pick

2] Partial

Understanding Partial

Example Usage of Partial

Benefits of Using Partial

3] Readonly

Understanding Readonly

Example Usage of Readonly

Benefits of Using Readonly

Important Note

4] Record & Map

Record

Example Using Record

Map

Example Using Map

Record vs. Map

5] Exclude

Understanding Exclude

Example Using Exclude

Benefits of Using Exclude

6] Type Inferences In Zod

How Type Inference Works in Zod

Example: Type Inference with Zod in an Express App

Assigning a Type to updateBody

Benefits of Type Inference in Zod

Before diving into an advanced TypeScript API, it's important to have a solid understanding of the basics of TypeScript, especially when it comes to using it in a Node.js environment. Here's an elaboration on the prerequisites and a recap of the setup procedure for a TypeScript project.

Prerequisites

To be prepared for the advanced TypeScript API module, you should:

- 1. Understand Basic TypeScript Classes:** Familiarity with how classes are defined and used in TypeScript, including constructors, properties, methods, and inheritance.
- 2. Understand Interfaces and Types:** Know how to define and use interfaces and types to enforce the structure of objects and function parameters.
- 3. Experience with TypeScript in Node.js:** Have experience setting up a simple Node.js application with TypeScript and understand how to run and compile TypeScript code.

The following code snippet is a test to check your understanding:

```
interface User {  
  name: string;  
  age: number;  
}
```

```

function sumOfAge(user1: User, user2: User) {
  return user1.age + user2.age;
};

// Example usage
const result = sumOfAge({
  name: "harkirat",
  age: 20
}, {
  name: "raman",
  age: 21
});
console.log(result); // Output: 41

```

In this example, you should understand the following concepts:

- **Interface `User`** : Defines the structure for a user object with `name` and `age` properties.
- **Function `sumOfAge`** : Takes two `User` objects as parameters and returns the sum of their ages.
- **Example Usage**: Demonstrates how to call `sumOfAge` with two user objects and logs the result.

(Note: The original output comment `// Output: 9` seems to be a typo. The correct output should be `41` based on the provided ages.)

Recap Setup Procedure

To start a TypeScript project locally, follow these steps:

1. Initialize TypeScript:

Run `npx tsc --init` in your project directory to create a `tsconfig.json` file, which is the configuration file for TypeScript.

2. Configure `tsconfig.json`:

Edit the `tsconfig.json` file to specify the root directory and the output directory for the compiled JavaScript files.

```
{
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist",
    // ... other options
  }
}
```

- `"rootDir": "./src"` : Tells TypeScript to look for `.ts` files in the `src` directory.
- `"outDir": "./dist"` : Compiled `.js` files will be output to the `dist` directory.

1] Pick

The `Pick` utility type in TypeScript is a powerful feature that allows you to construct new types by selecting a subset of properties from an existing type. This can be particularly useful when you need to work with only certain fields of a complex type, enhancing type safety and code readability without redundancy.

Understanding `Pick`

The `Pick` utility type is part of TypeScript's mapped types, which enable you to create new types based on the keys of an existing type. The syntax for `Pick` is as follows:

```
Pick<Type, Keys>
```

- `Type` : The original type you want to pick properties from.
- `Keys` : The keys (property names) you want to pick from the `Type` , separated by `|` (the union operator).

Example Usage of `Pick`

Consider an interface `User` that represents a user in your application:

```
interface User {
  id: number;
  name: string;
  email: string;
  createdAt: Date;
}
```

Suppose you're creating a function to display a user profile, but you only need the `name` and `email` properties for this purpose. You can use `Pick` to create a new type, `UserProfile` , that includes only these properties:

```
// Creating a new type with only `name` and `email` properties from `User`
type UserProfile = Pick<User, 'name' | 'email'>

// Function that accepts a UserProfile type
const displayUserProfile = (user: UserProfile) => {
  console.log(`Name: ${user.name}, Email: ${user.email}`);
};
```

In this example, `UserProfile` is a new type that has only the `name` and `email` properties from the original `User` interface. The `displayUserProfile` function then uses this `UserProfile` type for its

parameter, ensuring that it can only receive objects that have `name` and `email` properties.

Benefits of Using `Pick`

- Enhanced Type Safety:** By creating more specific types for different use cases, you reduce the risk of runtime errors and make your intentions clearer to other developers.
- Code Readability:** Using `Pick` to create descriptive types can make your code more readable and self-documenting.
- Reduced Redundancy:** Instead of defining new interfaces manually for subsets of properties, `Pick` allows you to reuse existing types, keeping your code DRY (Don't Repeat Yourself).

The `Pick` utility type in TypeScript allows you to create types that are subsets of existing types. It allows you to be explicit about what properties a function or component expects, leading to more maintainable and error-resistant code.

2] Partial

The `Partial` utility type in TypeScript is used to create a new type by making all properties of an existing type optional. This is particularly useful when you want to update a subset of an object's properties without needing to provide the entire object.

Understanding `Partial`

The `Partial` utility type takes a single type argument and produces a type with all the properties of the provided type set to optional. Here's the syntax for using `Partial`:

`Partial<Type>`

- `Type` : The original type you want to convert to a type with optional properties.

Example Usage of `Partial`

Let's say you have a `User` interface representing a user in your application:

```
interface User {  
    id: string;  
    name: string;  
    age: string;  
    email: string;  
    password: string;  
};
```

If you're creating a function to update a user, you might only want to update their `name`, `age`, or `email`, and not all properties at once. You can use `Pick` to select these properties and then apply `Partial` to make them optional:

```
// Selecting 'name', 'age', and 'email' properties from User
type UpdateProps = Pick<User, 'name' | 'age' | 'email'>

// Making the selected properties optional
type UpdatePropsOptional = Partial<UpdateProps>

// Function that accepts an object with optional 'name', 'age', and 'email' properties
function updateUser(updatedProps: UpdatePropsOptional) {
    // hit the database to update the user
}

// Example usage of updateUser
updateUser({ name: "Alice" }); // Only updating the name
updateUser({ age: "30", email: "alice@example.com" }); // Updating age and email
updateUser({}); // No updates, but still a valid call
```

In this example, `UpdatePropsOptional` is a new type where the `name`, `age`, and `email` properties are all optional, thanks to `Partial`. The `updateUser` function can then accept an object with any combination of these properties, including an empty object.

Benefits of Using `Partial`

- 1. Flexibility in Updates:** `Partial` is ideal for update operations where you may only want to modify a few properties of an object.
- 2. Type Safety:** Even though the properties are optional, you still get the benefits of type checking for the properties that are provided.
- 3. Code Simplicity:** Using `Partial` can simplify function signatures by not requiring clients to pass an entire object when only a part of it is needed.

The `Partial` utility type in TypeScript is useful where you need to work with objects that might only have a subset of their properties defined. It allows you to create types that are more flexible for update operations while still maintaining type safety.

3] Readonly

The `Readonly` utility type in TypeScript is used to make all properties of a given type read-only. This means that once an object of this type is created, its properties cannot be reassigned. It's particularly useful for defining configuration objects, constants, or any other data structure that should not be modified after initialization.

Understanding `Readonly`

The `Readonly` utility type takes a type `T` and returns a type with all properties of `T` set as read-only. Here's the basic syntax:

```
 Readonly<Type>
```

- `Type` : The original type you want to convert to a read-only version.

Example Usage of `Readonly`

Consider an interface `Config` that represents configuration settings for an application:

```
interface Config {  
  endpoint: string;  
  apiKey: string;  
}
```

To ensure that a `Config` object cannot be modified after it's created, you can use the `Readonly` utility type:

```
const config: Readonly<Config> = {  
  endpoint: '<https://api.example.com>',  
  apiKey: 'abcdef123456',  
};  
  
// Attempting to modify the object will result in a TypeScript error  
// config.apiKey = 'newkey'; // Error: Cannot assign to 'apiKey' because it is a read-only prop
```

In this example, `config` is an object that cannot be modified after its initialization. Trying to reassign `config.apiKey` will result in a compile-time error, ensuring the immutability of the `config` object.

Benefits of Using `Readonly`

1. **Immutability:** Ensures that objects are immutable after they are created, preventing accidental modifications.
2. **Compile-Time Checking:** The immutability is enforced at compile time, catching potential errors early in the development process.

3. Clarity and Intent: Using `Readonly` clearly communicates the intent that an object should not be modified, making the code easier to understand.

Important Note

It's crucial to remember that the `Readonly` utility type enforces immutability at the TypeScript level, which means it's a compile-time feature. JavaScript, which is the output of TypeScript compilation, does not have built-in immutability, so the `Readonly` constraint does not exist at runtime. This distinction is important for understanding the limitations of `Readonly` and recognizing that it's a tool for improving code quality and safety during development.

The `Readonly` utility type is a valuable feature in TypeScript for creating immutable objects. By preventing reassignment of properties, it helps maintain the integrity of objects that represent fixed configurations or constants.

4] Record & Map

The `Record` utility type and the `Map` object in TypeScript offer two powerful ways to work with collections of key-value pairs. Each has its own use cases and benefits, depending on the requirements of your application.

Record

The `Record<K, T>` utility type is used to construct a type with a set of properties `K` of a given type `T`. It provides a cleaner and more concise syntax for typing objects when you know the shape of the values but not the keys in advance.

Example Using `Record`

```
interface User {
  id: string;
  name: string;
}

// Using Record to type an object with string keys and User values
type Users = Record<string, User>

const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};
```

```
console.log(users['abc123']); // Output: { id: 'abc123', name: 'John Doe' }
```

In this example, `Users` is a type that represents an object with any string as a key and `User` objects as values. The `Record` utility type simplifies the declaration of such structures, making your code more readable and maintainable.

Map

The `Map` object in TypeScript (inherited from JavaScript) represents a collection of key-value pairs where both the keys and values can be of any type. Maps remember the original insertion order of the keys, which is a significant difference from plain JavaScript objects.

Example Using `Map`

```
interface User {
  id: string;
  name: string;
}

// Initialize an empty Map with string keys and User values
const usersMap = new Map<string, User>();

// Add users to the map using .set
usersMap.set('abc123', { id: 'abc123', name: 'John Doe' });
usersMap.set('xyz789', { id: 'xyz789', name: 'Jane Doe' });

// Accessing a value using .get
console.log(usersMap.get('abc123')); // Output: { id: 'abc123', name: 'John Doe' }
```

In this example, `usersMap` is a `Map` object that stores `User` objects with string keys. The `Map` provides methods like `.set` to add key-value pairs and `.get` to retrieve values by key. Maps are particularly useful when you need to maintain the order of elements, perform frequent additions and deletions, or use non-string keys.

Record vs. Map

- **Use `Record` when:** You are working with objects that have a fixed shape for values and string keys. It's ideal for typing object literals with known value types.
- **Use `Map` when:** You need more flexibility with keys (not just strings or numbers), or you need to maintain the insertion order of your keys. Maps also provide better performance for large sets of data, especially when frequently adding and removing key-value pairs.

Both **Record** and **Map** enhance TypeScript's ability to work with collections of data in a type-safe manner, each offering unique benefits suited to different scenarios in application development.

5] Exclude

The **Exclude** utility type in TypeScript is used to construct a type by excluding from a union type certain members that should not be allowed. It's particularly useful when you want to create a type that is a subset of another type, with some elements removed.

Understanding **Exclude**

The **Exclude** utility type takes two arguments:

- **T** : The original union type from which you want to exclude some members.
- **U** : The union type containing the members you want to exclude from **T**.

The result is a type that includes all members of **T** that are not assignable to **U**.

Example Using **Exclude**

Let's say you have a union type **Event** that represents different types of events in your application:

```
type Event = 'click' | 'scroll' | 'mousemove';
```

If you have a function that should handle all events except for **scroll** events, you can use **Exclude** to create a new type that omits **scroll**:

```
// Using Exclude to create a new type without 'scroll'  
type ExcludeEvent = Exclude<Event, 'scroll'>; // 'click' | 'mousemove'  
  
// Function that accepts only 'click' and 'mousemove' events  
const handleEvent = (event: ExcludeEvent) => {  
    console.log(`Handling event: ${event}`);  
};  
  
handleEvent('click'); // OK  
handleEvent('scroll'); // Error: Argument of type '"scroll"' is not assignable to parameter of
```

In this example, **ExcludeEvent** is a new type that includes only '**click**' and '**mousemove**', as '**scroll**' has been excluded. The **handleEvent** function then uses this **ExcludeEvent** type for its parameter, ensuring that it cannot receive a '**scroll**' event.

Benefits of Using `Exclude`

1. **Type Safety:** `Exclude` helps you enforce stricter type constraints in your functions and variables, preventing unwanted types from being used.
2. **Code Readability:** Using `Exclude` can make your type intentions clearer to other developers, as it explicitly shows which types are not allowed.
3. **Utility:** It's a built-in utility type that saves you from having to manually construct new types, making your code more concise and maintainable.

The `Exclude` utility type in TypeScript allows to create types that exclude certain members from a union. It allows you to refine type definitions for specific use cases, enhancing type safety and clarity in your code.

6] Type Inferences In Zod

Type inference in Zod is a powerful feature that allows TypeScript to automatically determine the type of data validated by a Zod schema. This capability is particularly useful in applications where runtime validation coincides with compile-time type safety, ensuring that your code not only runs correctly but is also correctly typed according to your Zod schemas.

How Type Inference Works in Zod

Zod schemas define the shape and constraints of your data at runtime. When you use Zod with TypeScript, you can leverage Zod's type inference to automatically generate TypeScript types based on your Zod schemas. This means you don't have to manually define TypeScript interfaces or types that replicate your Zod schema definitions, reducing redundancy and potential for error.

Example: Type Inference with Zod in an Express App

Consider an Express application where you want to validate and update a user's profile information. You define a Zod schema for the profile update request body:

```
import { z } from 'zod';
import express from "express";

const app = express();
app.use(express.json()); // Middleware to parse JSON bodies

// Define the schema for profile update
```

```

const userProfileSchema = z.object({
  name: z.string().min(1, { message: "Name cannot be empty" }),
  email: z.string().email({ message: "Invalid email format" }),
  age: z.number().min(18, { message: "You must be at least 18 years old" }).optional(),
});

app.put("/user", (req, res) => {
  const result = userProfileSchema.safeParse(req.body);

  if (!result.success) {
    res.status(400).json({ error: result.error });
    return;
  }

  // Type of updateBody is inferred from userProfileSchema
  const updateBody = result.data;

  // update database here
  res.json({
    message: "User updated",
    updateBody
  });
});

app.listen(3000, () => console.log("Server running on port 3000"));

```

In this example, `userProfileSchema.safeParse(req.body)` validates the request body against the `userProfileSchema`. The `safeParse` method returns an object that includes a `success` boolean and, on success, a `data` property containing the validated data.

Assigning a Type to `updateBody`

Thanks to Zod's type inference, the type of `updateBody` is automatically inferred to be:

```
{
  name: string;
  email: string;
  age?: number;
}
```

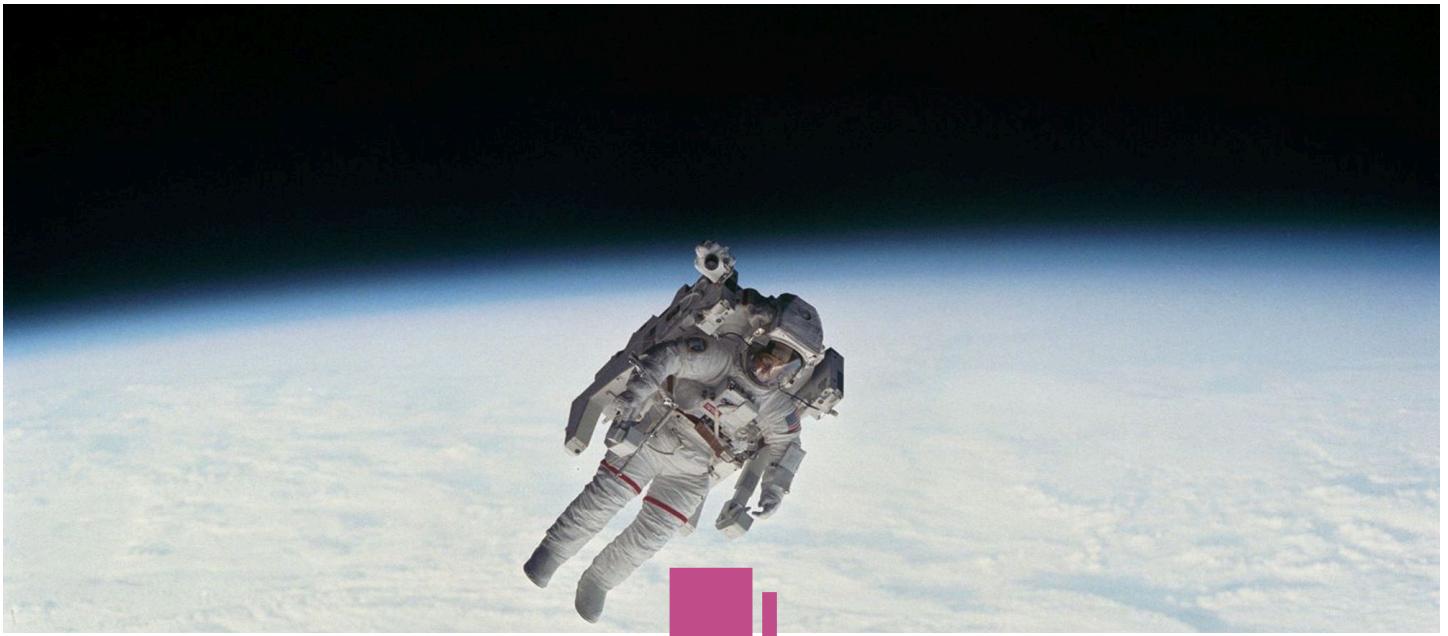
This inferred type is derived directly from the `userProfileSchema` definition. If you try to access a property on `updateBody` that isn't defined in the schema, TypeScript will raise a compile-time error, providing an additional layer of type safety.

Benefits of Type Inference in Zod

- 1. Reduced Boilerplate:** You don't need to manually define TypeScript types that mirror your Zod schemas.

2. **Type Safety:** Ensures that your data conforms to the specified schema both at runtime (through validation) and at compile-time (through type checking).
3. **Developer Productivity:** Type inference, combined with Zod's expressive API for defining schemas, makes it easier to write, read, and maintain your validation logic and related type definitions.

Type inference in Zod bridges the gap between runtime validation and compile-time type safety in TypeScript applications. By automatically generating TypeScript types from Zod schemas, Zod helps ensure that your data validation logic is both correct and type-safe, enhancing the reliability and maintainability of your code.



Week 12.3

Actionable Docker

If you're aiming to develop a thorough understanding of Docker and delve into its extensive features, I strongly suggest watching the detailed Docker series available on YouTube, which is split into [part 1](#) and [part 2](#). These tutorials are designed to provide you with an in-depth exploration of Docker, covering a wide range of topics from the basics to more advanced features.

Today's session, however, is focused on providing a short, actionable introduction to Docker. This will help you get started with running packages locally and give you a taste of what Docker can do.

[Actionable Docker](#)

[Installation](#)

[Verification](#)

[Why Docker?](#)

[Docker Hub](#)

[Common Commands](#)

[Common Packages](#)

Installation

Docker can be installed using the Docker GUI, which simplifies the setup process. Detailed instructions for various operating systems can be found on the official Docker documentation website at <https://docs.docker.com/engine/install/>.

Verification

After installation, you should verify that Docker is installed correctly by running the `docker run hello-world` command. This command will pull the "hello-world" image from Docker Hub and run it in a new container, which should print a message to the terminal.

```
→ ~ docker run hello-world
^[Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
478afc919002: Pull complete
Digest: sha256:d000bc569937abbe195e20322a0bde6b2922d805332fd6d8a68b19f524b7d21d
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(arm64v8)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

```
https://hub.docker.com/
```

For more examples and ideas, visit:

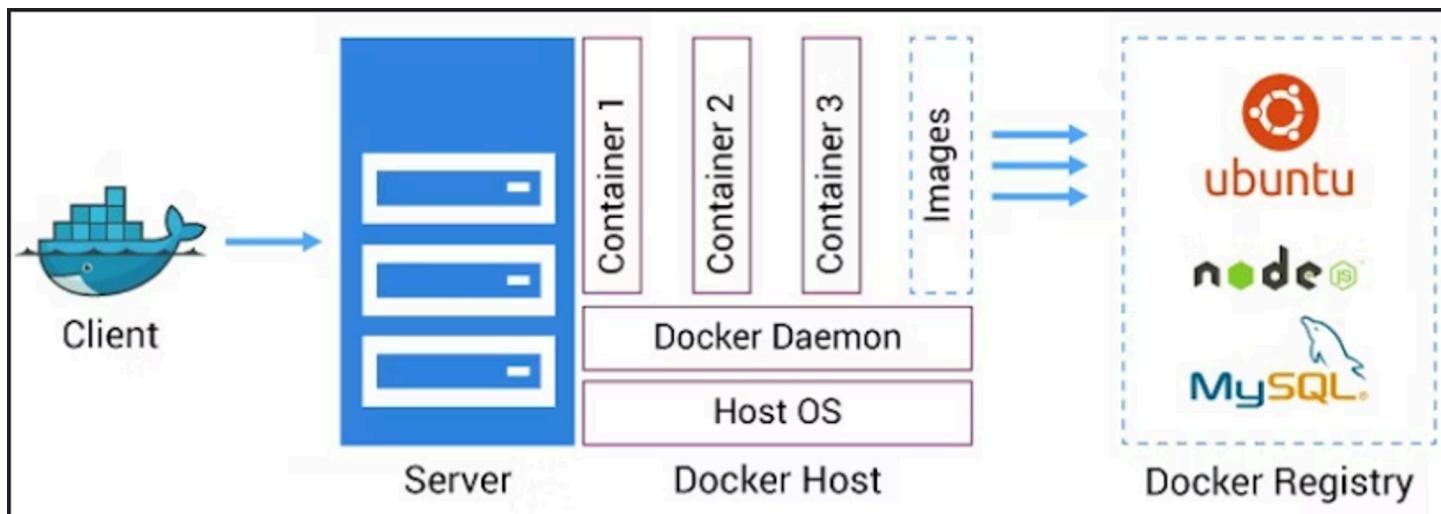
```
https://docs.docker.com/get-started/
```

Why Docker?

Docker is a powerful platform that serves several purposes in the development, deployment, and running of applications. Below are the reasons why it is used:

- **Containerization of Applications:** Docker allows you to package your application and its dependencies into a container, which is a lightweight, standalone, and executable software package. This containerization ensures that the application runs consistently across different computing environments, from development to staging to production.

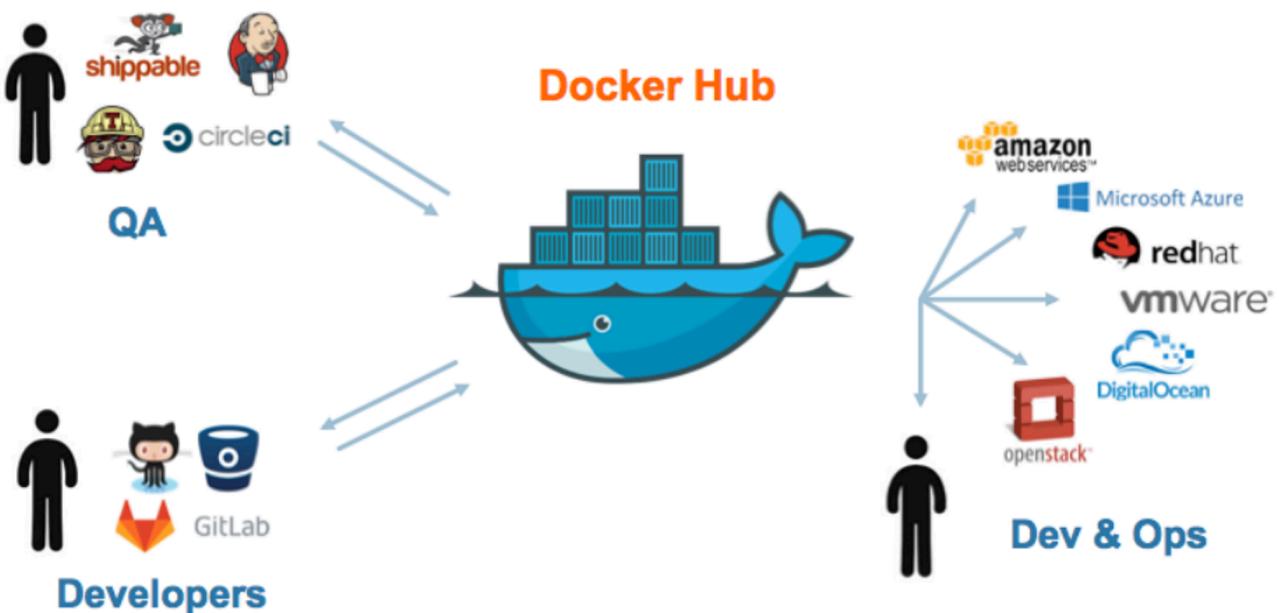
- **Running Other People's Code and Packages:** With Docker, you can easily run software and applications built by others without worrying about setting up the required environment or dependencies. This is because all the necessary components are included within the container.
- **Running Common Software Packages:** Docker provides the ability to run common software packages, such as databases (MongoDB, PostgreSQL, etc.), within containers. This means you can quickly deploy and manage these services without the need to install and configure them directly on your host machine.



The use of Docker streamlines the development process, simplifies deployment, and enhances the scalability and portability of applications. It isolates applications in containers, making it easier to manage dependencies and avoid conflicts between different software running on the same system.

Docker Hub

Docker registries are similar to version control repositories for code, such as GitHub or GitLab, but instead of code, they store Docker images. Docker images are the blueprints for creating Docker containers, which include the application and all of its dependencies.



Docker Hub is the default registry for Docker and is analogous to GitHub in the context of Docker images. It's a cloud-based repository where users can sign up for an account, push their custom images, pull images published by others, and work with automated build workflows. Here's the relevance of Docker Hub in comparison to GitHub:

- **Version Control and Collaboration:** Just as GitHub allows developers to store, version, and collaborate on code, Docker Hub provides similar functionalities for Docker images. Users can keep track of different versions of their images, collaborate with team members, and integrate with continuous integration/continuous deployment (CI/CD) pipelines.
- **Public and Private Repositories:** Both platforms offer the ability to have public repositories, where anyone can access and use the resources, and private repositories, which are restricted to authorized users.
- **Community and Official Images:** Docker Hub hosts a vast collection of community-generated images, similar to how GitHub hosts open-source projects. It also provides official images maintained by software vendors or the Docker team, ensuring a trusted source of commonly used software packages.
- **Automated Builds:** Docker Hub can automatically build images from source code in a repository when changes are made, similar to how CI/CD systems work with GitHub to automate the testing and deployment of code.

In summary, Docker Hub is a central repository for Docker images, where users can store, manage, and distribute their containerized applications. It plays a crucial role in the Docker ecosystem, facilitating the sharing and deployment of software in a manner that's consistent with how code is managed on platforms like GitHub.

Common Commands

The commands listed are part of the basic Docker CLI (Command Line Interface) operations that allow you to manage Docker containers. Here's an explanation of each command, along with an analogy to help understand their functions:

- **`docker run`** : This command is used to create and start a Docker container from a specified image. It's like saying "start this application" in the Docker world. For example, `docker run mongo` starts a MongoDB container using the official MongoDB image from Docker Hub.
- **`docker ps`** : This command lists all currently running containers, much like the `ps` command in Unix-based systems that shows running processes. It's like looking at a list of active applications on your computer.
- **`docker kill <container_id>`** : This command stops a running container immediately. It's similar to force-quitting an application on your computer.

Now, let's delve into the specific scenarios mentioned:

- **Running a simple image:** When you run `docker run mongo`, you're starting a MongoDB container. However, without port mapping, you won't be able to access the MongoDB instance from your host machine.
- **Adding a port mapping:** By using `docker run -p 27017:27017 mongo`, you map the default MongoDB port (27017) from the container to the host. This is like setting up a direct phone line to a specific office in a large building; the port number is the extension number.
- **Starting in detached mode:** The `d` flag starts the container in the background (detached mode), allowing you to continue using the terminal. It's like putting a program to run in the background on your computer so you can do other tasks.
- **Inspecting a container with `docker ps`:** This shows you all the containers that are currently running, providing details such as container ID, image used, command executed, creation time, status, and ports.
- **Stopping a container with `docker kill`:** When you want to stop a container, you use `docker kill` followed by the container ID. This is like using the "End Task" feature in a task manager to stop a program.

In summary, the flow of commands for running a MongoDB container with port mapping in detached mode and then inspecting and stopping it would be:

1. `docker run -d -p 27017:27017 mongo` (Run MongoDB in detached mode with port mapping)
2. `docker ps` (Inspect running containers)
3. `docker kill <container_id>` (Stop the specified container)

These commands provide a basic workflow for managing Docker containers, from starting them to making them accessible and finally stopping them when they're no longer needed.

Common Packages

To better understand the use of Docker for running database services, let's consider the MongoDB and PostgreSQL packages. Docker allows you to run these databases in containers, which are isolated environments that contain everything the software needs to run.

- **MongoDB:**

```
docker run -d -p 27017:27017 mongo
```

This command runs a MongoDB container in detached mode (`-d`), which means it runs in the background. The `-p 27017:27017` option maps the default MongoDB port inside the container (27017) to the same port on the host machine, allowing you to connect to MongoDB from your local machine as if it were running natively.

- **PostgreSQL:**

```
docker run -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres
```

This command runs a PostgreSQL container with a specified environment variable (`-e`) setting the default user's password to "mysecretpassword". It also runs in detached mode and maps the default PostgreSQL port (5432) from the container to the host. The connection string provided is used to connect to this PostgreSQL instance from your local machine.

The connection string `postgresql://postgres:mysecretpassword@localhost:5432/postgres` is used to connect to the PostgreSQL server from your local machine. It includes the username, password, host, port, and database name.

Below is a simple Node.js script to test the connection to the PostgreSQL database running in the Docker container:

```
// Import the pg library
const { Client } = require('pg');

// Define your connection string (replace placeholders with your actual data)
const connectionString = 'postgresql://postgres:mysecretpassword@localhost:5432/postgres';

// Create a new client instance with the connection string
const client = new Client({
  connectionString: connectionString
});

// Connect to the database
client.connect(err => {
  if (err) {
    console.error('connection error', err.stack);
  } else {
    console.log('connected to the database');
  }
});

// Run a simple query (Example: Fetching the current date and time from PostgreSQL)
client.query('SELECT NOW()', (err, res) => {
  if (err) {
    console.error(err);
  } else {
    console.log(res.rows[0]);
  }
}

// Close the connection
client.end();
});
```

The Node.js code snippet provided is an example of how to use the `pg` library to connect to the PostgreSQL server running in the Docker container. It creates a new client with the connection string, connects to the database, runs a query to fetch the current date and time, and then closes the connection.

This script does the following:

1. Imports the `pg` library, which is a PostgreSQL client for Node.js.
2. Defines the connection string using the credentials and port specified in the `docker run` command for PostgreSQL.

3. Creates a new client instance with the connection string.
4. Connects to the PostgreSQL database.
5. Runs a query to fetch the current date and time from the database.
6. Logs the result of the query or an error if the connection or query fails.
7. Closes the database connection.

The images provided in the search results show the output of running the `docker run hello-world` command, which is a test to ensure that Docker is installed and running correctly on your system. This command is not directly related to the MongoDB and PostgreSQL commands or the Node.js code snippet.

Using Docker to run these databases in containers simplifies the setup and ensures that the software runs the same way on any machine, regardless of the host environment. This is because the container includes the database software and all of its dependencies, configured exactly as needed to run.



Week 12.4

SQL Relationships & Joins

In today's lecture, Harkirat revisits the [Fundamentals of SQL](#), with a particular focus on the crucial concepts of [relationships and transactions](#) in SQL databases. The session provided a deeper understanding of how to structure and query relational data using [joins](#), and explored the various [types of joins](#) available in SQL.

SQL Relationships & Joins

Recap

[Types of Databases](#)

[Why Not NoSQL](#)

[Why SQL?](#)

[Creating a PostgreSQL Database](#)

[Using psql](#)

[Creating a Table and Defining Its Schema](#)

[Interacting with the Database \(CRUD Operations\)](#)

[Using the pg Library](#)

Relationships & Transactions

[Relationships in MongoDB \(NoSQL\)](#)

[Relationships in SQL](#)

[SQL Queries for Relationships](#)

[Transactions in SQL](#)

Node.js Code for Transactions

Joins

The Problem with Separate Queries

The Power of Joins

Implementing Joins in a Node.js Application

Benefits of Using a Join

Types of Joins

1. INNER JOIN

2. LEFT JOIN (or LEFT OUTER JOIN)

3. RIGHT JOIN (or RIGHT OUTER JOIN)

4. FULL JOIN (or FULL OUTER JOIN)

Recap

Before diving into advanced topics such as "Relations, Transactions, and Join Queries in SQL Databases," it's crucial to have a solid foundation in the basics of SQL and database management. Here's a detailed recap of the previously covered topics:

Types of Databases

- **Relational (SQL) Databases:** Organize data into tables consisting of rows and columns. Each table represents a relation, and the rows hold individual records. Popular relational database management systems (RDBMS) include MySQL, PostgreSQL, Microsoft SQL Server, and Oracle Database[1][2].
- **Non-relational (NoSQL) Databases:** Use different data models for storing, managing, and accessing data. Common models include document-oriented, key-value, graph, and wide-column stores. NoSQL databases are designed for flexibility, scalability, and high performance with unstructured or semi-structured data[2].

Why Not NoSQL

NoSQL databases offer schema-less data storage, which is beneficial for rapid development and handling large volumes of unstructured data. However, the lack of a strict schema can lead to data inconsistency and challenges in enforcing data integrity as applications grow[1].

Why SQL?

SQL databases are preferred for applications requiring strict data integrity, complex transactions, and relationships between data entities. They support ACID (Atomicity, Consistency, Isolation, Durability)

properties, ensuring reliable transaction processing. SQL databases are ideal for applications like e-commerce platforms and financial systems where data consistency is critical[1][2].

Creating a PostgreSQL Database

- **Using `psql`**: A terminal-based front-end to PostgreSQL that allows for interactive command execution and database management.
- **Using Docker**: Running PostgreSQL in a Docker container for development purposes.

Using `psql`

`psql` is a powerful tool for interacting with PostgreSQL databases, allowing users to execute SQL queries, manage database objects, and view data directly from the terminal[1].

Creating a Table and Defining Its Schema

SQL databases require defining a schema before inserting data. This involves creating tables and specifying columns with data types and constraints. For example:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Interacting with the Database (CRUD Operations)

- **INSERT**: Adds new records to a table.
- **UPDATE**: Modifies existing records based on specific criteria.
- **DELETE**: Removes records from a table.
- **SELECT**: Retrieves data from one or more tables.

Using the `pg` Library

The `pg` library is a Node.js package for interfacing with PostgreSQL databases. It allows for executing SQL queries from within a Node.js application, leveraging JavaScript for database operations.

```
const { Client } = require('pg');
const client = new Client({ connectionString: "your_connection_string" });
await client.connect();
// Perform database operations
await client.end();
```

Now after this recap, we are all set to explore more advanced database concepts, including relationships between tables, transactions that ensure data integrity across multiple operations, and join queries that retrieve related data from multiple tables.



Relationships & Transactions

In database systems, relationships are fundamental for linking data stored across different tables. This section will elaborate on how relationships are handled in both NoSQL and SQL databases, and how transactions are used in SQL databases to ensure data integrity.

Relationships in MongoDB (NoSQL)

MongoDB, a NoSQL database, allows for flexible data modeling. You can store related data together in a single document, which can contain nested objects and arrays. This is useful for encapsulating all data about an entity in one place without the need for separate tables.

Example: Storing User Details with Address in MongoDB

```
{  
  "username": "john_doe",  
  "email": "john@example.com",  
  "password": "securepassword",  
  "address": {  
    "city": "New York",  
    "country": "USA",  
    "street": "123 Liberty St",  
    "pincode": "10005"  
  }  
}
```

This structure allows you to store a user's details along with their address in a single, nested object.

Relationships in SQL

SQL databases, on the other hand, require a more structured approach. Since SQL databases don't store objects directly, you need to create separate tables for different entities and establish relationships between them using keys.

Example: Defining Relationships in SQL

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE addresses (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    city VARCHAR(100) NOT NULL,
    country VARCHAR(100) NOT NULL,
    street VARCHAR(255) NOT NULL,
    pincode VARCHAR(20),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

In this SQL schema, the `addresses` table has a `user_id` column that serves as a foreign key, creating a relationship with the `users` table. The `FOREIGN KEY` constraint enforces the relationship and ensures referential integrity.

SQL Queries for Relationships

To insert an address for a user, you would use an `INSERT` statement:

```
INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (1, 'New York', 'USA', '123 Broadway St', '10001');
```

To retrieve the address of a user given their ID, you would use a `SELECT` statement:

```
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = 1;
```

Transactions in SQL

Transactions are critical in SQL databases to ensure that a series of operations either all succeed or all fail. This is important when you have multiple related operations that must be treated as a single unit.

Example: Using Transactions in SQL

```
BEGIN; -- Start transaction

INSERT INTO users (username, email, password)
VALUES ('john_doe', 'john_doe1@example.com', 'securepassword123');

INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (currval('users_id_seq'), 'New York', 'USA', '123 Broadway St', '10001');

COMMIT;
```

In this example, both `INSERT` statements are wrapped in a transaction. If any statement fails, the entire transaction is rolled back, leaving the database in a consistent state.

Node.js Code for Transactions

In a Node.js application, you can use the `pg` library to manage transactions programmatically.

Example: Inserting User and Address with Transaction in Node.js

```
import { Client } from 'pg';

async function insertUserAndAddress(
    username: string,
    email: string,
    password: string,
    city: string,
    country: string,
    street: string,
    pincode: string
) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
        user: 'postgres',
        password: 'mysecretpassword',
    });

    try {
        await client.connect();

        // Start transaction
        await client.query('BEGIN');

        // Insert user
```

```

const insertUserText = `

    INSERT INTO users (username, email, password)
    VALUES ($1, $2, $3)
    RETURNING id;

`;

const userRes = await client.query(insertUserText, [username, email, password]);
const userId = userRes.rows[0].id;

// Insert address using the returned user ID
const insertAddressText = `

    INSERT INTO addresses (user_id, city, country, street, pincode)
    VALUES ($1, $2, $3, $4, $5);

`;

await client.query(insertAddressText, [userId, city, country, street, pincode]);

// Commit transaction
await client.query('COMMIT');

console.log('User and address inserted successfully');
} catch (err) {
    await client.query('ROLLBACK'); // Roll back the transaction on error
    console.error('Error during transaction, rolled back.', err);
    throw err;
} finally {
    await client.end(); // Close the client connection
}
}

// Example usage
insertUserAndAddress(
    'johndoe',
    'john.doe@example.com',
    'securepassword123',
    'New York',
    'USA',
    '123 Broadway St',
    '10001'
);

```

In the provided Node.js function `insertUserAndAddress`, a transaction is started with `BEGIN`, and `COMMIT` is used to apply the changes. If an error occurs, `ROLLBACK` is called to undo the transaction. This ensures that either both the user information and address are inserted into the database, or neither is, maintaining data integrity.

While NoSQL databases like MongoDB allow for flexible data storage, SQL databases require a structured approach with defined relationships and transactions to ensure data integrity. Using transactions in SQL databases,

especially when performing multiple related operations, is crucial to prevent partial updates and maintain consistency. The `pg` library in Node.js provides the necessary tools to handle transactions effectively in your applications.

Joins

Joins in SQL are a powerful feature that allows you to combine rows from two or more tables based on a related column between them. This is essential when you want to fetch data that is distributed across multiple tables.

The Problem with Separate Queries

Fetching user details and their address could be done with two separate queries, but this approach has several drawbacks:

Approach 1: Separate Queries (Not Recommended)

```
-- Query 1: Fetch user's details
SELECT id, username, email
FROM users
WHERE id = YOUR_USER_ID;

-- Query 2: Fetch user's address
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = YOUR_USER_ID;
```

This approach results in two round trips to the database, which can increase latency. It also complicates the application logic, as you need to manage two separate result sets and ensure they are related correctly in your application code.

The Power of Joins

Using joins, you can retrieve related data from multiple tables in a single query. This is more efficient and simplifies your application logic.

Approach 2: Using Joins (Recommended)

```
SELECT users.id, users.username, users.email, addresses.city, addresses.country, addresses.street
FROM users
JOIN addresses ON users.id = addresses.user_id
WHERE users.id = YOUR_USER_ID;
```

In this query, the `JOIN` clause is used to combine rows from `users` and `addresses` where the `id` of the user matches the `user_id` in the addresses table.

Implementing Joins in a Node.js Application

When implementing joins in a Node.js application using the `pg` library, you can use the same SQL join syntax within your application code.

Approach 2: Using Joins in Node.js

```
import { Client } from 'pg';

// Async function to fetch user data and their address together
async function getUserDetailsWithAddress(userID: string) {
    const client = new Client({
        // ...connection config
    });

    try {
        await client.connect();
        const query = `
            SELECT u.id, u.username, u.email, a.city, a.country, a.street, a.pincode
            FROM users u
            JOIN addresses a ON u.id = a.user_id
            WHERE u.id = $1
        `;
        const result = await client.query(query, [userID]);

        if (result.rows.length > 0) {
            console.log('User and address found:', result.rows[0]);
            return result.rows[0];
        } else {
            console.log('No user or address found with the given ID.');
            return null;
        }
    } catch (err) {
        console.error('Error during fetching user and address:', err);
        throw err;
    } finally {
        await client.end();
    }
}

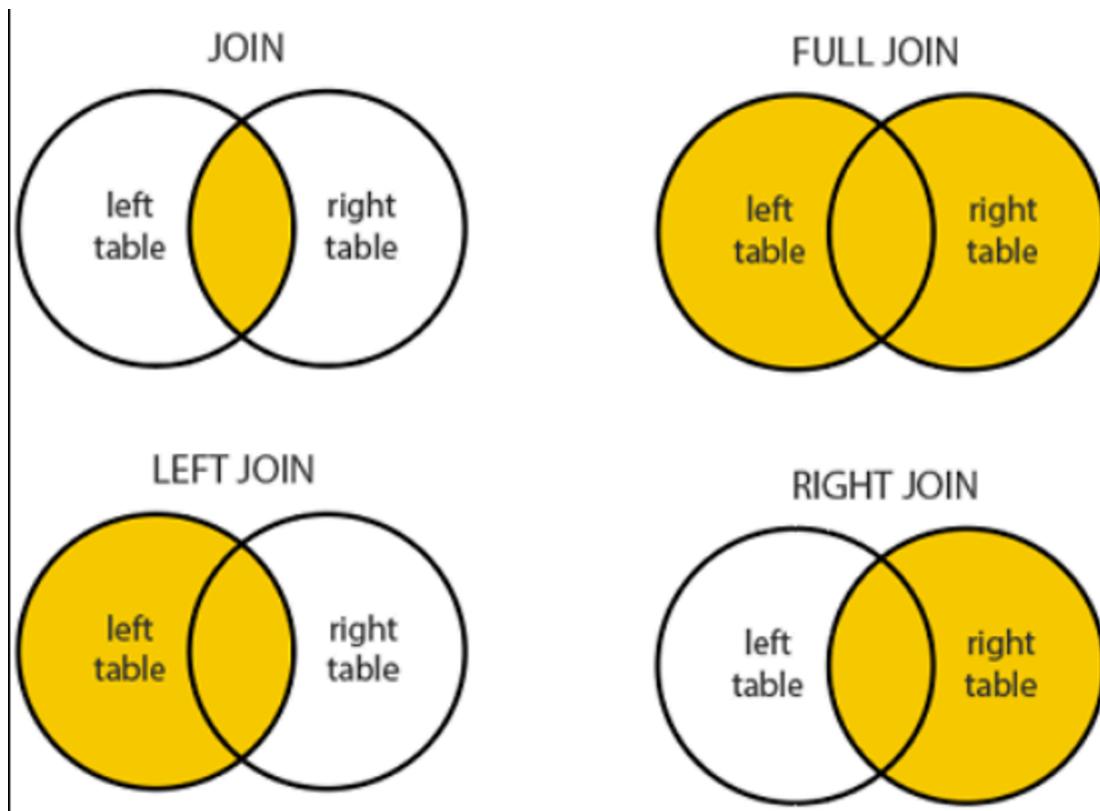
getUserDetailsWithAddress("1");
```

In this function, `getUserDetailsWithAddress`, a single query with a join is used to fetch both the user details and their address. This reduces latency and simplifies the application logic, as you handle only one result set.

Benefits of Using a Join

- **Reduced Latency:** Fewer database calls mean less communication overhead and faster responses.
- **Simplified Application Logic:** Handling a single result set is easier than coordinating multiple queries and their results.
- **Transactional Integrity:** A join query ensures that the data retrieved is consistent and reflects the state of the database at the time of the query.

Using joins is a best practice for fetching related data in SQL databases. It leverages the relational nature of SQL databases and provides a performant, reliable, and clean way to retrieve and work with data in your applications.



Types of Joins

SQL joins are used to combine rows from two or more tables based on a related column between them. There are several types of joins, each with its own use case depending on the nature of the data and the desired results. Here's a detailed explanation of the different types of joins:

1. INNER JOIN

Definition: The `INNER JOIN` keyword selects records that have matching values in both tables.

Use Case: You want to retrieve only the rows with matching keys in both tables. For example, if you want to find all users who have provided their address details, you would use an **INNER JOIN**.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode  
FROM users  
INNER JOIN addresses ON users.id = addresses.user_id;
```

In this query, only users with corresponding entries in the **addresses** table will be returned.

2. LEFT JOIN (or LEFT OUTER JOIN)

Definition: The **LEFT JOIN** keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is **NULL** from the right side if there is no match.

Use Case: To list all users and their address information, regardless of whether they have provided an address. Users without an address will appear with **NULL** values for address fields.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode  
FROM users  
LEFT JOIN addresses ON users.id = addresses.user_id;
```

This query includes all users, with address information where available.

3. RIGHT JOIN (or RIGHT OUTER JOIN)

Definition: The **RIGHT JOIN** keyword returns all records from the right table, and the matched records from the left table. The result is **NULL** from the left side if there is no match.

Use Case: While less common due to typical foreign key constraints, a **RIGHT JOIN** would be used if you want to start with the "addresses" table and optionally include user information.

SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode  
FROM users  
RIGHT JOIN addresses ON users.id = addresses.user_id;
```

Given the usual foreign key constraints, every address should have a corresponding user, making **RIGHT JOIN** less likely to be used in this context.

4. FULL JOIN (or FULL OUTER JOIN)

Definition: The `FULL JOIN` keyword returns all records when there is a match in either left (table1) or right (table2) table records.

Use Case: A `FULL JOIN` would be used to combine all records from both "users" and "addresses" tables, showing all users and all addresses, with `NULL` values in columns from the table that lacks a matching row.

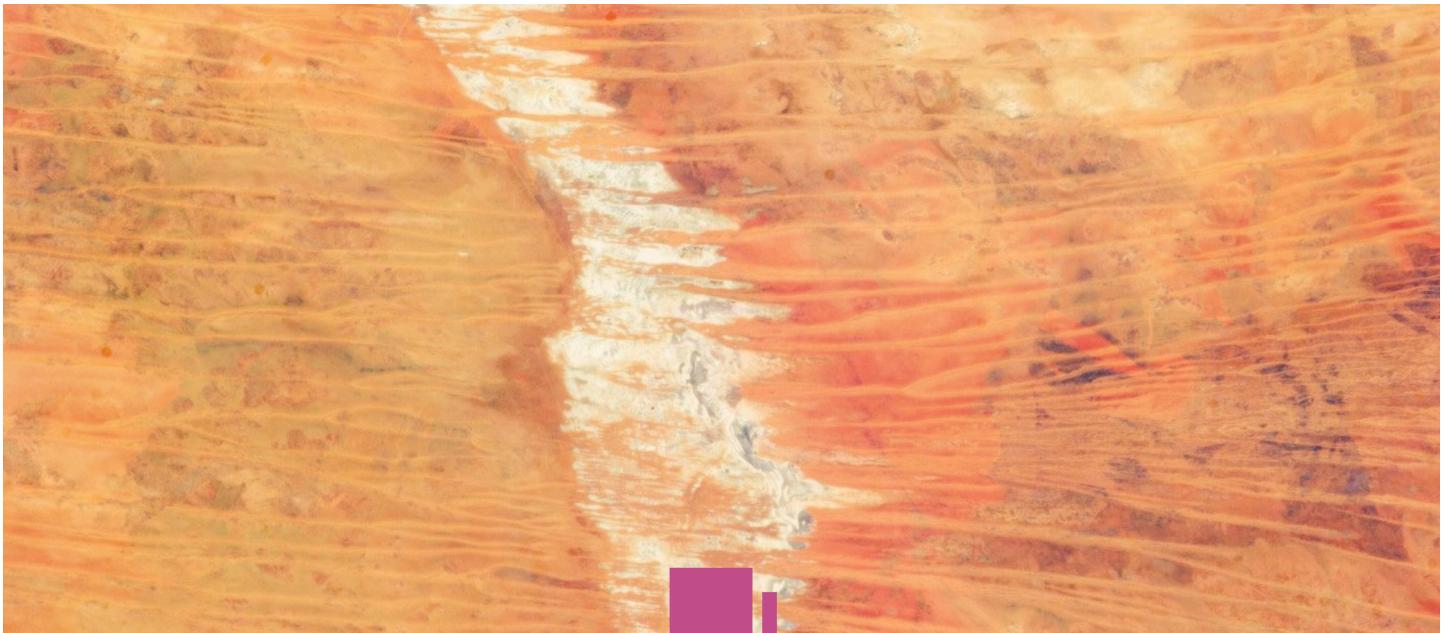
SQL Example:

```
SELECT users.username, addresses.city, addresses.country, addresses.street, addresses.pincode  
FROM users  
FULL JOIN addresses ON users.id = addresses.user_id;
```

This query would reveal all users and addresses, including any orphaned records that don't have a match in the other table.

Each type of join serves a specific purpose and can be chosen based on the data relationships and the information you need to retrieve:

Use `INNER JOIN` when you need to match rows from both tables. Use `LEFT JOIN` to include all rows from the left table, with matching rows from the right table if available. Use `RIGHT JOIN` to include all rows from the right table, with matching rows from the left table if available. Use `FULL JOIN` to include rows when there is a match in either table.



Week 12.5

Prisma Relationships

In today's lecture, Harkirat provided a comprehensive [Recap of Prisma](#), focusing on the framework's approach to defining and [managing relationships within a database schema](#). He also explains the significance of the [Prisma Client](#) in facilitating database operations and the role of [Prisma's migration system](#) in tracking and applying schema changes.

[Prisma Relationships](#)

[Relationships](#)

[Types of Relationships in Prisma](#)

[One to Many Relationship in the TODO App](#)

[Updating the Prisma Schema](#)

[Why do you need Prisma Client](#)

[Updating the Database and the Prisma Client](#)

[Exploring the Prisma Migrations Folder](#)

[Todo Functions](#)

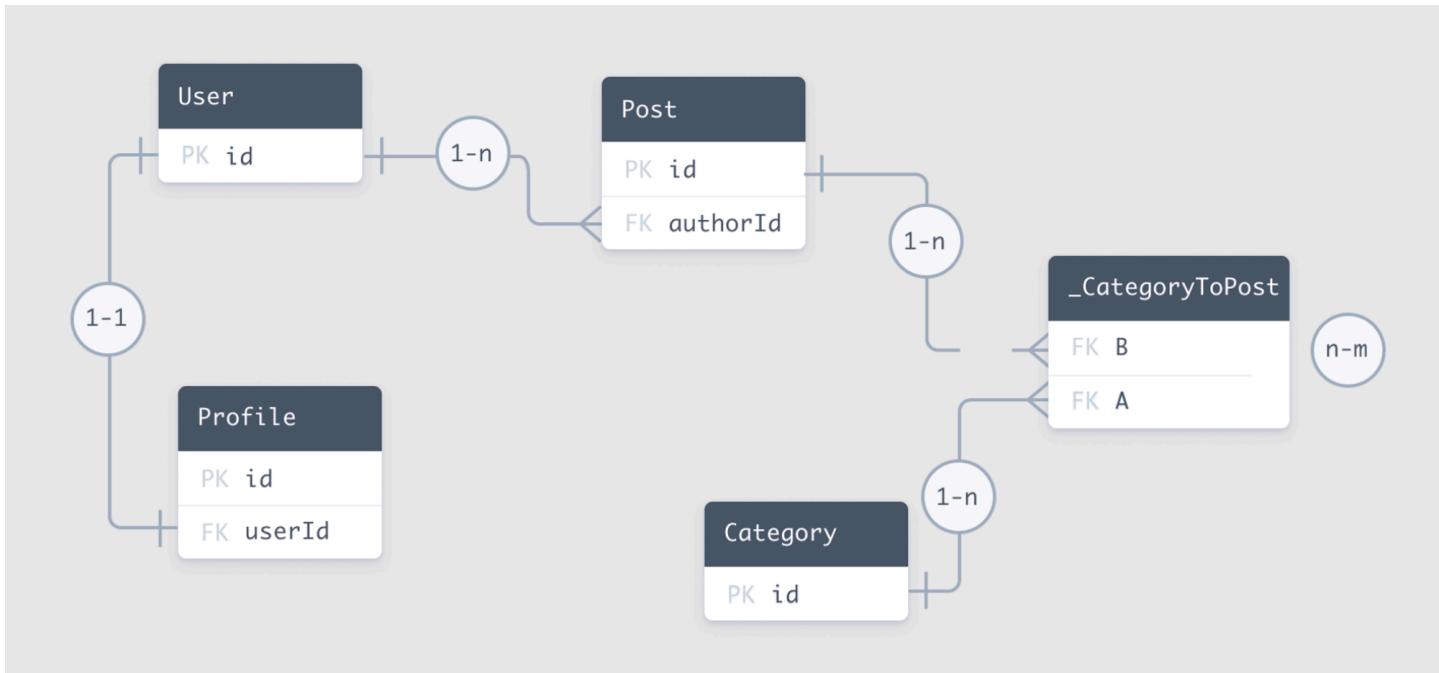
[1. createTodo Function](#)

[2. getTodos Function](#)

[3. getTodosAndUserDetails Function](#)

Relationships

In Prisma, relationships between tables are defined using a straightforward and expressive syntax within the Prisma schema file. These relationships are crucial for representing how data in one table is associated with data in another, and Prisma supports several types of relationships to model the various ways in which data can be interconnected.



Types of Relationships in Prisma

Prisma allows you to define the following types of relationships:

- One to One**: A relationship where a single record in one table is linked to a single record in another table.
- One to Many**: A relationship where a single record in one table is linked to multiple records in another table.
- Many to One**: The inverse of one to many, where multiple records in one table are linked to a single record in another table.
- Many to Many**: A relationship where multiple records in one table are linked to multiple records in another table.

One to Many Relationship in the TODO App

For the TODO app, there is a one-to-many relationship between the `User` and `Todo` models. This means that one user can have many todos, but each todo is associated with only one user.

Updating the Prisma Schema

To define a one-to-many relationship in Prisma, you update the `schema.prisma` file to include a reference from the `Todo` model to the `User` model. Here's how the updated schema looks based on the provided image:

```
// This is your Prisma schema file,
// learn more about it in the docs: <https://pris.ly/d/prisma-schema>

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = "postgresql://postgres:mysecretpassword@localhost:5432/postgres"
}

model User {
  id      Int      @id @default(autoincrement())
  username String  @unique
  password String
  firstName String
  lastName String
  todos   Todo[]
}

model Todo {
  id      Int      @id @default(autoincrement())
  title  String
  description String
  done    Boolean @default(false)
  userId  Int
  user    User     @relation(fields: [userId], references: [id])
}
```

In this schema:

- The `User` model has a `todos` field, which is an array of `Todo` objects. This represents the "many" side of the one-to-many relationship.
- The `Todo` model has a `userId` field, which stores the reference to the associated `User`. It also has a `user` field that establishes the relationship using the `@relation` attribute. The `fields: [userId]` part specifies which field on the `Todo` model is used to store the connection, and `references: [id]` part specifies which field on the `User` model is being referred to.

Why do you need Prisma Client

The Prisma Client is an auto-generated and type-safe database client that allows developers to interact with their database in a comfortable and secure way. It is part of the Prisma ecosystem, which aims to make database access easy and robust.

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

async function createTodoForUser(userId: number, title: string, description: string) {
  const todo = await prisma.todo.create({
    data: {
      title,
      description,
      user: {
        connect: { id: userId },
      },
    },
  });
  return todo;
}

async function getUserWithTodos(userId: number) {
  const userWithTodos = await prisma.user.findUnique({
    where: { id: userId },
    include: { todos: true },
  });
  return userWithTodos;
}

// Example usage
createTodoForUser(1, 'Prisma Client', 'Learn how to use Prisma Client').then(todo => {
  console.log('Created new todo:', todo);
});

getUserWithTodos(1).then(user => {
  console.log('User with todos:', user);
});
```



In this example, `createTodoForUser` creates a new `Todo` record associated with a `User` by their `id`. The `getUserWithTodos` function retrieves a user and their related `Todo` items using Prisma Client's `findUnique` method with the `include` option to fetch related records.

Updating the Database and the Prisma Client

After updating the schema, you need to apply the changes to your database and regenerate the

Prisma Client to reflect the new relationship:

```
npx prisma migrate dev --name relationship  
npx prisma generate
```

The `prisma migrate dev` command creates a new migration file in the `prisma/migrations` folder, which includes the SQL statements necessary to update the database schema with the new relationship. The `prisma generate` command updates the Prisma Client to include the new relationship logic.

Exploring the Prisma Migrations Folder

When you explore the `prisma/migrations` folder after running the migration, you will see a new directory for the migration you just created. Inside this directory, there will be files that describe the changes made to the database schema, including the addition of foreign keys and any other constraints related to the new relationship.

Todo Functions

In the context of a Prisma-based application, you can create functions to interact with the database and perform CRUD operations on the `Todo` and `User` models. Below are detailed explanations and code snippets for creating todos, retrieving todos for a user, and fetching todos along with user details.

1. createTodo Function

The `createTodo` function allows you to insert a new todo into the database for a specific user.

Solution:

```
import { PrismaClient } from "@prisma/client";  
  
const prisma = new PrismaClient();  
  
async function createTodo(userId: number, title: string, description: string) {  
  const todo = await prisma.todo.create({  
    data: {  
      title,  
      description,  
      userId  
    },  
  });  
  console.log(todo);  
}
```

```
createTodo(1, "go to gym", "go to gym and do 10 pushups");
```

In this function, `prisma.todo.create` is used to create a new `Todo` record associated with a `User` by their `userId`. The `data` object contains the fields required for the `Todo` model.

2. getTodos Function

The `getTodos` function retrieves all todos associated with a specific user.

Solution:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodos(userId: number) {
  const todos = await prisma.todo.findMany({
    where: {
      userId: userId,
    },
  });
  console.log(todos);
}

getTodos(1);
```

Here, `prisma.todo.findMany` is used with a `where` clause to filter todos by the `userId`, returning all todos for that user.

3. getTodosAndUserDetails Function

The `getTodosAndUserDetails` function fetches todos along with the details of the user who created them. This is similar to performing a join in SQL.

Bad Solution (Separate Queries):

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodosAndUserDetails(userId: number) {
  const user = await prisma.user.findUnique({
    where: {
      id: userId
    }
  });
}
```

```
const todos = await prisma.todo.findMany({
  where: {
    userId: userId,
  }
});
console.log(user);
console.log(todos);
}

getTodosAndUserDetails(1);
```

This approach uses two separate queries to fetch the user and their todos, which is less efficient.

Good Solution (Using Select):

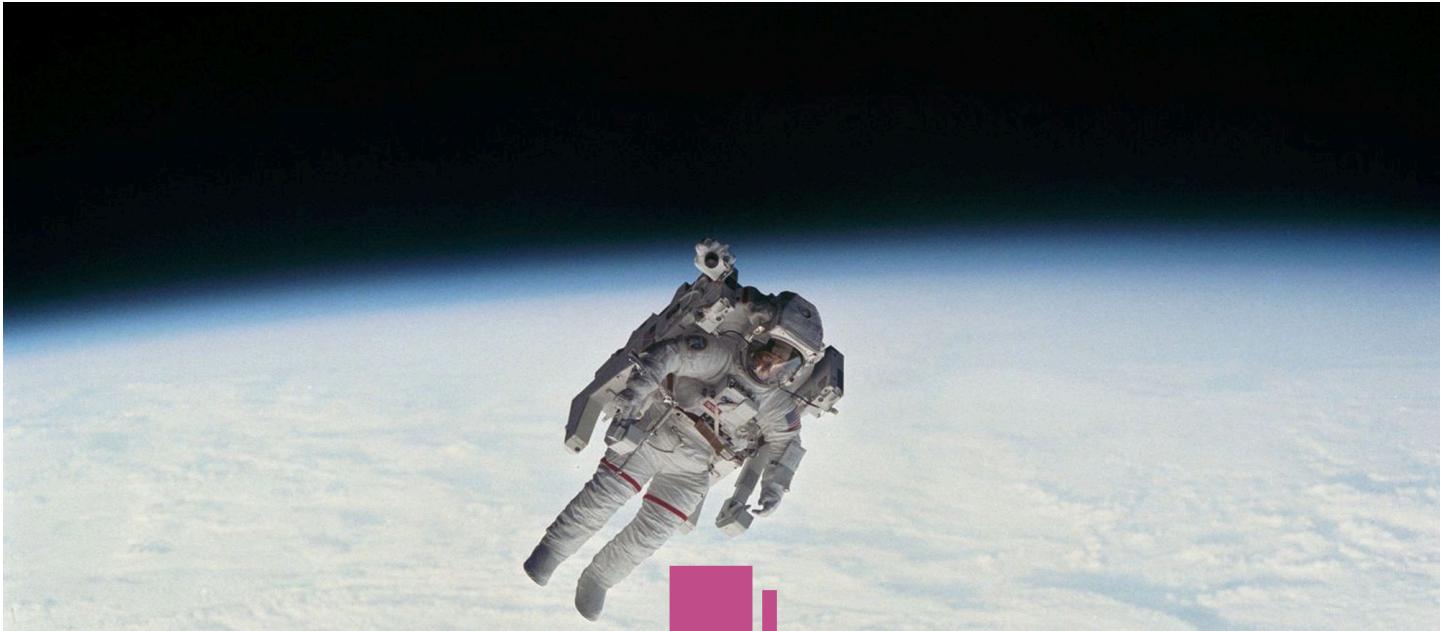
```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodosAndUserDetails(userId: number) {
  const todos = await prisma.todo.findMany({
    where: {
      userId: userId,
    },
    select: {
      title: true,
      description: true,
      done: true,
      user: {
        select: {
          username: true,
          firstName: true,
          lastName: true
        }
      }
    }
  });
  console.log(todos);
}

getTodosAndUserDetails(1);
```

In this improved solution, a single query with a `select` statement is used to fetch todos and include the user details for each todo. The `select` statement specifies that we want to include the `user` object with only the `username`, `firstName`, and `lastName` fields for each todo.



Week 12.6

Connection Pooling In Serverless ENV

In today's lecture, Harkirat explores [connection pooling in serverless environments](#), emphasizing its importance for managing the high volume of database connections that platforms like [Cloudflare Workers](#) generate. The discussion also covered the challenges of using Prisma ORM in such settings and introduced [Prisma Accelerate](#) as a managed service for efficient connection pooling, ensuring scalable and stable database access for serverless applications.

Connection Pooling In Serverless ENV

Connection Pooling

What is Connection Pooling?

Connection Pooling In Prisma for Serverless ENV

1] Install Prisma in Your Project:

2] Initialize Prisma:

3] Create a Basic Schema:

4] Create Migrations:

5] Sign Up to Prisma Accelerate:

6] Generate an API Key:

7] Add Accelerate as a Dependency:

8] Generate the Prisma Client Without Binary Engines:

9] Set Up Your Code:

The wrangler.toml File

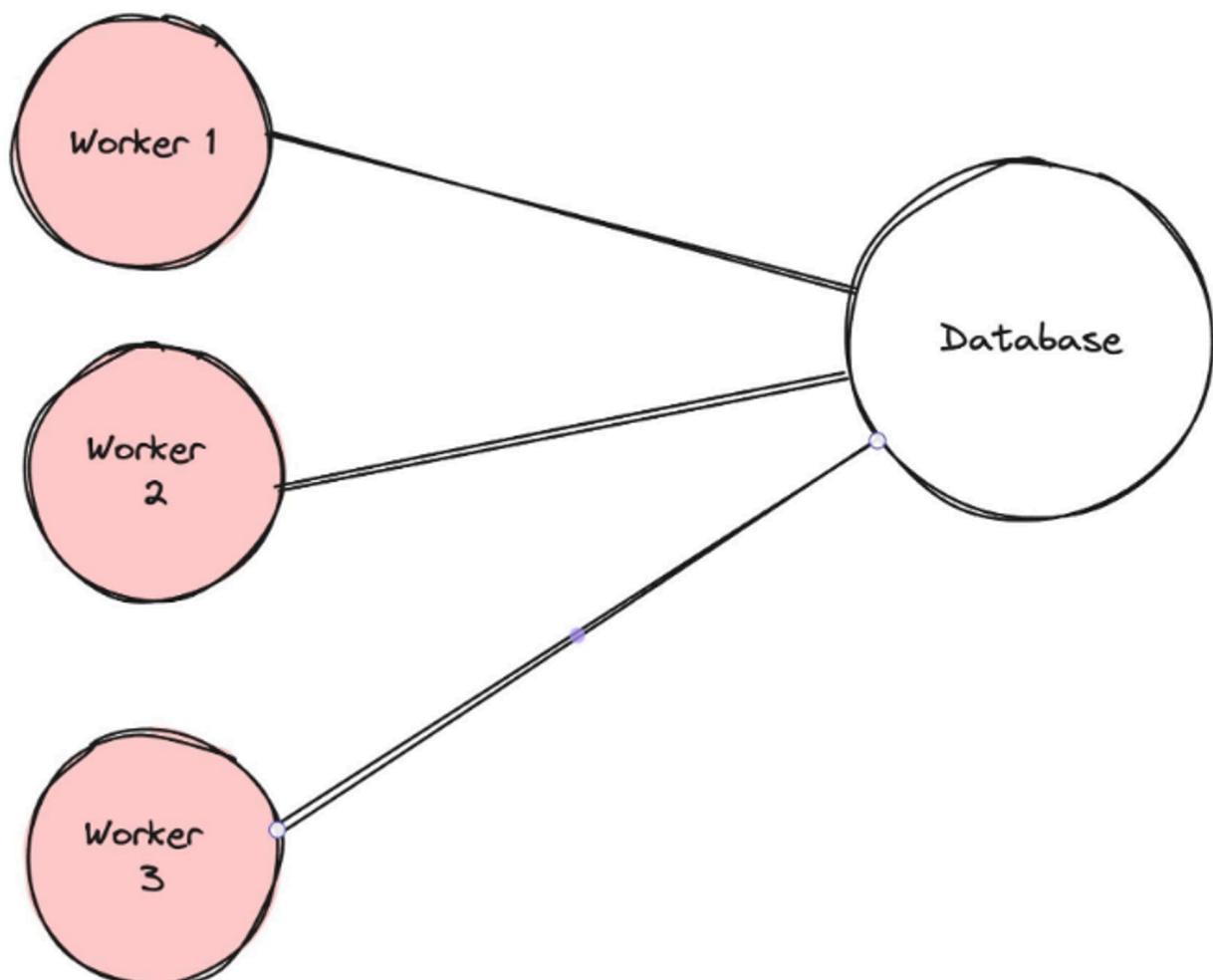
Connection Pooling

Connection pooling is a technique used to manage and reuse database connections efficiently. It is particularly relevant in serverless environments and applications with high levels of concurrency. Here's a detailed explanation of connection pooling and its importance:

What is Connection Pooling?

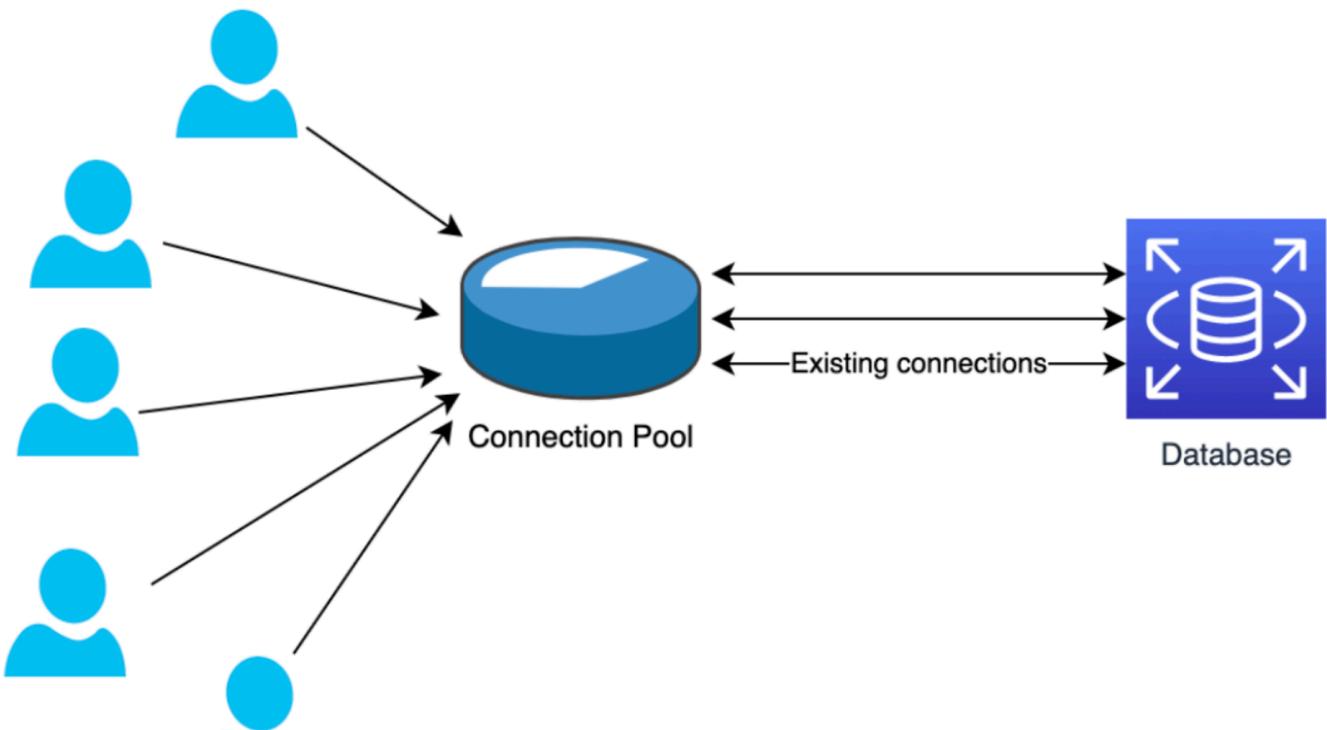
Connection pooling refers to the practice of maintaining a cache of database connection objects that can be reused by multiple clients. Instead of opening and closing a new connection for each user request, a connection pool allows a set of connections to be shared among requesting threads or processes. When a new request comes in, it can borrow a connection from the pool, use it for the database operation, and then return it to the pool for future use.

Serverless environments, such as Cloudflare Workers, present unique challenges for database connectivity. Since serverless functions can scale up rapidly and run in multiple regions, they can potentially open a large number of connections to the database. This can quickly exhaust the database's connection limit and degrade performance.



The above image illustrates a scenario where multiple workers (Worker 1, Worker 2, Worker 3) are each establishing their own connections to a single database. In a serverless environment, this can lead to a high number of simultaneous database connections, which can strain the database server and potentially exceed its connection limit. Using a connection pool can resolve this issue by acting as an intermediary that manages database connections on behalf of the workers. Here's how it works:

1. **Centralized Connection Management:** The connection pool maintains a set of open connections to the database. Instead of each worker opening its own connection, they request a connection from the pool.
2. **Efficient Resource Utilization:** When a worker needs to interact with the database, it checks out a connection from the pool, uses it for the database operation, and then returns it to the pool. This allows connections to be reused, reducing the overhead of establishing new connections for each operation.
3. **Controlled Concurrency:** The pool limits the number of active connections. If all connections in the pool are in use and another worker requests a connection, it will have to wait until a connection is returned to the pool. This prevents the database from being overwhelmed with too many concurrent connections.
4. **Improved Performance:** Connection pooling improves the performance of the system by reducing the time spent on connection setup and teardown. It also helps in maintaining a stable number of connections, which can be tuned for optimal performance based on the database's capacity.
5. **Scalability:** As the number of workers increases, the connection pool can help scale the system more effectively. It ensures that the growth in the number of workers does not lead to a proportional increase in the number of database connections, which could otherwise lead to scalability issues.



In summary, a connection pool collects all the database requests from the workers and manages the connections collectively. This approach streamlines the process of connecting to the database, enhances performance, and ensures that the database server is not overloaded with connection requests. It is a critical component in serverless architectures where the number of instances can fluctuate significantly, and efficient resource management is key to maintaining system stability and performance.

Connection Pooling In Prisma for Serverless ENV

Connection pooling in Prisma for serverless environments is a crucial feature that addresses the challenges of managing database connections in a scalable and efficient manner. Prisma Accelerate is a service that enhances Prisma's capabilities in serverless deployments, such as those on Cloudflare Workers. Here's a detailed explanation of the steps to set up Prisma with connection pooling using Prisma Accelerate:

1] Install Prisma in Your Project:

Add Prisma to your project as a development dependency using npm.

```
npm install --save-dev prisma
```

2] Initialize Prisma:

Set up the initial Prisma configuration files in your project.

```
npx prisma init
```

3] Create a Basic Schema:

Define your data model in the Prisma schema file. This includes specifying the database provider (e.g., PostgreSQL) and the URL, which will later be replaced with the Prisma Accelerate connection string.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id      Int    @id @default(autoincrement())
  name    String
  email   String
  password String
}
```

4] Create Migrations:

Generate the necessary database migrations based on your schema.

```
npx prisma migrate dev --name init
```

5] Sign Up to Prisma Accelerate:

Register for Prisma Accelerate, which provides a managed connection pooling solution for serverless applications.

6] Generate an API Key:

Obtain an API key from Prisma Accelerate and replace the placeholder in your `.env` file with the actual key.

```
DATABASE_URL="prisma://accelerate.prisma-data.net/?api_key=your_key"
```

7] Add Accelerate as a Dependency:

Install the Prisma Accelerate extension to enable connection pooling.

```
npm install @prisma/extension-accelerate
```

8] Generate the Prisma Client Without Binary Engines:

Create the Prisma client tailored for serverless environments.

```
npx prisma generate --no-engine
```

9] Set Up Your Code:

Integrate Prisma Client with Prisma Accelerate in your serverless application code. Here's an example using the Hono framework:

```
import { Hono, Next } from 'hono'
import { PrismaClient } from '@prisma/client/edge'
import { withAccelerate } from '@prisma/extension-accelerate'
import { env } from 'hono/adapter'

const app = new Hono()

app.post('/', async (c) => {
  // Todo: add Zod validation here
  const body = await c.req.json()
  const { DATABASE_URL } = env<{ DATABASE_URL: string }>(c)

  const prisma = new PrismaClient({
```

```
    datasourceUrl: DATABASE_URL,  
  }).$extends(withAccelerate())  
  
  console.log(body)  
  
  await prisma.user.create({  
    data: {  
      name: body.name,  
      email: body.email,  
      password: body.password  
    }  
  })  
  
  return c.json({msg: "User created"})  
})  
  
export default app
```

In this setup, Prisma Accelerate acts as the connection pool manager. It collects all the database requests from the serverless functions (workers) and manages the connections to the database. This ensures that the number of connections does not exceed the database's capacity and that the connections are efficiently reused, improving performance and scalability in serverless environments.

The `wrangler.toml` File

The `wrangler.toml` file is a configuration file used by Wrangler, which is a command-line interface (CLI) tool for Cloudflare Workers. In the context of using Prisma with connection pooling in a serverless environment like Cloudflare Workers, the `wrangler.toml` file would specify the settings and parameters required to deploy and configure the serverless application.

Here's how the `wrangler.toml` file relates to the setup:

- **Project Configuration:** The `wrangler.toml` file contains various configuration options for your Cloudflare Workers project, such as the account ID, project name, and environment variables.
- **Environment Variables:** You can define environment variables in the `wrangler.toml` file, which might include the `DATABASE_URL` for Prisma. This URL would be the connection string provided by Prisma Accelerate, which includes the API key and other necessary parameters for connection pooling.

- **Workers Script Settings:** The file also includes settings related to the Workers script itself, such as the entry point for the application code and any build commands that need to be run before deployment.
- **Dependencies:** If your Worker uses npm packages, such as `@prisma/client` and `@prisma/extension-accelerate`, you may need to specify how these dependencies are bundled with your Worker script.
- **Routes and Triggers:** The `wrangler.toml` file can also define routes and triggers that determine when your Worker is executed in response to HTTP requests.

The `wrangler.toml` file is essential for deploying a serverless application that uses Prisma with connection pooling on Cloudflare Workers. It helps manage the deployment process and ensures that the necessary environment variables and settings are in place for the application to connect to the database using Prisma Accelerate's managed connection pooling service.