React with TypeScript Notes

Day 1: TypeScript Fundamentals

Learning Objectives

- Understand why TypeScript is used.
- Master basic types, interfaces, and functions.
- Write type-safe code.

---

1. What is TypeScript?

- **Superset of JavaScript**: Adds static typing to catch errors during development (e.g., passing a string to a function expecting a number).
- **Benefits**:
  - **Early Error Detection**: Type mismatches flagged in your IDE.
  - **Better Autocompletion**: IDEs understand data shapes (e.g., `user.name` vs. `user.naem`).
  - **Clearer Code**: Types act as documentation.

---

2. Basic Types

Primitive Types

```
let name: string = "Alice";
let age: number = 25;
let isActive: boolean = true;
```

- **Arrays**:

```
let scores: number[] = [90, 85, 95];
let mixed: (string | number)[] = ["Alice", 30]; // Union type
```

Objects and Interfaces

```
interface User {
  id: number;
  name: string;
  email?: string; // Optional property
}

const user: User = { id: 1, name: "Alice" };
```

Functions

```
function greet(name: string): string {
  return `Hello, ${name}!`;
}

// Arrow function with type
const add = (a: number, b: number): number => a + b;
```

---

3. Advanced Types

Generics

Create reusable components that work with multiple types:

```
function identity<T>(arg: T): T {
  return arg;
}

let output = identity<string>("Hello"); // Output type: string
```

Type Aliases vs. Interfaces

- **Interfaces**: Extendable (use for object shapes).
- **Type Aliases**: Union/intersection types (e.g., `type Status = "active" | "inactive"`).

---

4. Classes

```
class Person {
  // Shorthand for declaring and assigning properties
  constructor(public name: string, private age: number) {}

  describe(): string {
    return `${this.name} is ${this.age} years old.`;
  }
}

const person = new Person("Alice", 25);
```

Key Takeaways

- TypeScript adds static types to JavaScript for reliability.
- Use interfaces for objects and generics for flexible functions.

Day 2: Setting Up React in WebStorm & React Basics

Learning Objectives

- Set up a React + TypeScript project.
- Understand components, props, and JSX.

---

1. Project Setup

1. Install **Node.js** (includes npm).
2. Create a React app:
   ```
   npx create-react-app my-app --template typescript
   ```
3. Open the project in **WebStorm**:
   - Use the built-in terminal to run `npm start`.

---

2. React Components

Functional Components

```tsx
// App.tsx
import React from 'react';

interface WelcomeProps {
  name: string;
}

const Welcome: React.FC<WelcomeProps> = ({ name }) => {
  return <h1>Hello, {name}!</h1>;
};

export default Welcome;
```

JSX Rules

- Return a **single root element** (use `<div>` or `<> </>` fragments).
- Use `className` instead of `class`.
- Embed JavaScript expressions with `{}`:
  ```
  <p>Score: {score * 10}</p>
  ```

---

3. Styling

Inline Styles

```tsx
<div style={{ color: 'red', fontSize: '20px' }}>Warning!</div>
```

CSS Modules

1. Create App.module.css:
```
.container { padding: 20px; }
```
2. Import:
```
import styles from './App.module.css';
<div className={styles.container}>...</div>
```

Key Takeaways

- Use create-react-app for quick setup.
- Components are reusable UI pieces with typed props.

Day 3: State Management with useState and Forms

Learning Objectives

- Manage component state with useState.
- Handle form submissions and routing.

---

1. The useState Hook

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

const Counter: React.FC = () => {
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};
```

Why Immutability Matters

- **Never modify state directly**:
```
// ❌ Wrong
customers.push(newCustomer);
// ✅ Correct
setCustomers([...customers, newCustomer]);
```

---

2. Forms in React

Controlled Components

Bind form inputs to state:

```
const [name, setName] = useState<string>("");

return (
  <form onSubmit={(e) => { e.preventDefault(); console.log(name); }}>
    <input
     type="text"
     value={name}
     onChange={(e) => setName(e.target.value)}
    />
    <button type="submit">Submit</button>
  </form>
);
```

Handling Multiple Inputs

```
const [formData, setFormData] = useState({ name: "", email: "" });

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  setFormData({
    ...formData,
    [e.target.name]: e.target.value
  });
};
```

---

3. React Router Setup

1. Install:
   ```
   npm install react-router-dom @types/react-router-dom
   ```
2. Define Routes:
   ```
   import { BrowserRouter, Routes, Route } from 'react-router-dom';

   const App = () => (
     <BrowserRouter>
       <Routes>
         <Route path="/" element={<Home />} />
         <Route path="/add-customer" element={<AddCustomer />} />
       </Routes>
     </BrowserRouter>
   );
   ```

Key Takeaways

- Use `useState` to manage dynamic data.
- Forms are controlled by React state.
- React Router enables navigation between pages.

Day 4: Full Customer CRUD with useState and React Router

Learning Objectives

- Build a full CRUD (Create, Read, Update, Delete) application.
- Use React Router for navigation.
- Style components with Tailwind CSS.

---

1. CRUD Operations with useState

Create (Add Customer)

```
interface Customer {
  id: number;
  name: string;
  email: string;
}

const [customers, setCustomers] = useState<Customer[]>([]);

const addCustomer = (customer: Customer) => {
  setCustomers([...customers, customer]);
};
```

Read (Display Customers)

```
const CustomerList: React.FC = () => {
  return (
    <ul>
      {customers.map((customer) => (
        <li key={customer.id}>{customer.name}</li>
      ))}
    </ul>
  );
};
```

Update (Edit Customer)

```
const updateCustomer = (id: number, updatedCustomer: Customer) => {
  setCustomers(customers.map((customer) =>
    customer.id === id ? updatedCustomer : customer
  ));
};
```

Delete (Remove Customer)

```
const deleteCustomer = (id: number) => {
  setCustomers(customers.filter((customer) => customer.id !== id));
};
```

---

2. React Router for CRUD

Routes for Each Operation

```
<Routes>
  <Route path="/" element={<CustomerList />} />
  <Route path="/add-customer" element={<AddCustomer />} />
  <Route path="/edit-customer/:id" element={<EditCustomer />} />
</Routes>
```

Navigate Programmatically

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

const navigate = useNavigate();
navigate('/'); // Redirect to home
```

---

3. Tailwind CSS

Installation

1. Install Tailwind:
   ```
   npm install tailwindcss postcss autoprefixer
   npx tailwindcss init
   ```
2. Configure tailwind.config.js:
   ```
   module.exports = {
     content: ['./src/**/*.{js,jsx,ts,tsx}'],
     theme: { extend: {} },
     plugins: [],
   };
   ```
3. Add Tailwind to index.css:
   ```
   @tailwind base;
   @tailwind components;
   @tailwind utilities;
   ```

Example Usage

```
<button className="bg-blue-500 text-white p-2 rounded">Submit</button>
```

Key Takeaways

- CRUD operations are managed using useState.
- React Router enables navigation between pages.
- Tailwind CSS simplifies styling with utility classes.

Day 5: useReducer and Context API

Learning Objectives

- Manage complex state with useReducer.
- Share state globally using the Context API.

---

1. useReducer

Why Use useReducer?

- Better for managing state with multiple sub-values or complex logic.

Example: Customer Reducer

```
interface Customer {
  id: number;
  name: string;
}

type Action =
  | { type: 'ADD_CUSTOMER'; payload: Customer }
  | { type: 'DELETE_CUSTOMER'; payload: number };

const customerReducer = (state: Customer[], action: Action): Customer[] => {
  switch (action.type) {
    case 'ADD_CUSTOMER':
      return [...state, action.payload];
    case 'DELETE_CUSTOMER':
      return state.filter((customer) => customer.id !== action.payload);
    default:
      return state;
  }
};

const [state, dispatch] = useReducer(customerReducer, []);
```

---

2. Context API

Create a Context

```
import { createContext, useContext } from 'react';

interface CustomerContextType {
  customers: Customer[];
  dispatch: React.Dispatch<Action>;
}

const CustomerContext = createContext<CustomerContextType>(null!);
```

Provide Context

```
const CustomerProvider: React.FC = ({ children }) => {
  const [state, dispatch] = useReducer(customerReducer, []);

  return (
    <CustomerContext.Provider value={{ customers: state, dispatch }}>
      {children}
    </CustomerContext.Provider>
  );
};
```

Consume Context

```
const { customers, dispatch } = useContext(CustomerContext);
```

Key Takeaways

- `useReducer` is ideal for complex state logic.
- Context API shares state across components without prop drilling.

Day 6: Redux with `createStore`

Learning Objectives

- Understand Redux fundamentals.
- Implement Redux in a React app.

---

1. Redux Basics

Store, Actions, and Reducers

- **Store**: Holds the global state.
- **Actions**: Describe what happened (e.g., `{ type: 'ADD_CUSTOMER', payload: customer }`).
- **Reducers**: Update the state based on actions.

Example Reducer

```
const customerReducer = (state = [], action) => {
  switch (action.type) {
    case 'ADD_CUSTOMER':
      return [...state, action.payload];
    default:
      return state;
  }
};
```

Create Store

```
import { createStore } from 'redux';
const store = createStore(customerReducer);
```

---

2. Connect Redux to React

useSelector and useDispatch

```
import { useSelector, useDispatch } from 'react-redux';

const customers = useSelector((state) => state.customers);
const dispatch = useDispatch();

dispatch({ type: 'ADD_CUSTOMER', payload: newCustomer });
```

Key Takeaways

- Redux centralizes state management.
- Use useSelector and useDispatch to interact with Redux in React.

Day 7: Redux Thunk and Admin Dashboard

Learning Objectives

- Handle async actions with Redux Thunk.
- Build an admin dashboard.

---

1. Redux Thunk

Async Action Example

```
const fetchCustomers = () => async (dispatch) => {
  const response = await axios.get('/api/customers');
  dispatch({ type: 'SET_CUSTOMERS', payload: response.data });
};
```

Apply Middleware

```
import { applyMiddleware, createStore } from 'redux';
import thunk from 'redux-thunk';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

---

2. Admin Dashboard

- Manage customers, items, and orders using Redux.
- Example:

```
<Route path="/admin/customers" element={<CustomerList />} />
<Route path="/admin/orders" element={<OrderList />} />
```

Key Takeaways

- Redux Thunk handles async actions like API calls.
- Admin dashboards centralize management tasks.

Day 8: Node.js and Express.js

Learning Objectives

- Set up a Node.js server.
- Build RESTful APIs with Express.js.

---

1. Introduction to Node.js

- **Node.js**: A runtime for executing JavaScript on the server.
- **Why Node.js?**
  - Non-blocking I/O: Handles many requests efficiently.
  - Unified language (JavaScript) for frontend and backend.

---

2. Setting Up Node.js

1. Install Node.js: Download from nodejs.org.
2. Initialize a project:

```
npm init -y
```

3. Install Express:

```
npm install express
```

---

3. Building a REST API with Express

Basic Server

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

app.get('/', (req, res) => {
```

```
  res.send('Hello, World!');
});

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

CRUD Endpoints

```
let customers = [];

// Create
app.post('/customers', (req, res) => {
  const customer = req.body;
  customers.push(customer);
  res.status(201).json(customer);
});

// Read
app.get('/customers', (req, res) => {
  res.json(customers);
});

// Update
app.put('/customers/:id', (req, res) => {
  const id = req.params.id;
  const updatedCustomer = req.body;
  customers = customers.map((customer) =>
    customer.id === id ? updatedCustomer : customer
  );
  res.json(updatedCustomer);
});

// Delete
app.delete('/customers/:id', (req, res) => {
  const id = req.params.id;
  customers = customers.filter((customer) => customer.id !== id);
  res.status(204).send();
});
```

---

4. Middleware

Body Parser

Parse incoming request bodies:

```
app.use(express.json());
```

Error Handling

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Key Takeaways

- Node.js enables server-side JavaScript.
- Express simplifies building RESTful APIs.
- Middleware processes requests and responses.

Day 9: Prisma ORM and Backend Development

Learning Objectives

- Understand what an ORM is and why Prisma is used.
- Set up Prisma with a MySQL database.
- Build a fully functional backend using Node.js, Express, and Prisma.

---

1. What is Prisma?

- **ORM (Object-Relational Mapping)**: A tool that maps database tables to JavaScript objects.
- **Why Prisma?**
    - Type-safe database queries.
    - Auto-generated migrations.
    - Easy-to-use API for CRUD operations.

---

2. Setting Up Prisma

1. Install Prisma:
```
npm install prisma --save-dev
```
2. Initialize Prisma:
```
npx prisma init
```
    - This creates a `prisma` folder with a `schema.prisma` file.
3. Configure the Database Connection:
    - Open `prisma/schema.prisma` and update the `datasource`:
```
datasource db {
  provider = "mysql"
  url      = "mysql://USER:PASSWORD@HOST:PORT/DATABASE"
}
```
4. Define Models:
    - Example: `Customer` model:
```
model Customer {
  id    Int    @id @default(autoincrement())
  name  String
  email String @unique
}
```

5. Run Migrations:

```
npx prisma migrate dev --name init
```

   - This creates the database tables based on your models.

---

3. Using Prisma in Express

1. Install Prisma Client:

```
npm install @prisma/client
```

2. Create a Prisma Client Instance:

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
```

3. CRUD Operations with Prisma:
   - **Create**:

```
app.post('/customers', async (req, res) => {
  const { name, email } = req.body;
  const customer = await prisma.customer.create({
    data: { name, email },
  });
  res.json(customer);
});
```

   - **Read**:

```
app.get('/customers', async (req, res) => {
  const customers = await prisma.customer.findMany();
  res.json(customers);
});
```

   - **Update**:

```
app.put('/customers/:id', async (req, res) => {
  const { id } = req.params;
  const { name, email } = req.body;
  const customer = await prisma.customer.update({
    where: { id: parseInt(id) },
    data: { name, email },
  });
  res.json(customer);
});
```

   - **Delete**:

```
app.delete('/customers/:id', async (req, res) => {
  const { id } = req.params;
  await prisma.customer.delete({
    where: { id: parseInt(id) },
  });
  res.status(204).send();
});
```

Key Takeaways

- Prisma simplifies database interactions with type-safe queries.
- Use Prisma Client to perform CRUD operations in Express.

Day 10: JWT Authentication

Learning Objectives

- Implement JWT-based authentication in the backend.
- Secure routes with middleware.
- Handle authentication in the frontend.

---

1. What is JWT?

- **JSON Web Token (JWT)**: A compact, URL-safe token for securely transmitting information.
- **Structure**:
  - Header: Algorithm and token type.
  - Payload: Data (e.g., user ID, roles).
  - Signature: Ensures token integrity.

---

2. Backend Implementation

1. Install Dependencies:

```
npm install jsonwebtoken bcryptjs
```

2. Create a Login Endpoint:

```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');

app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await prisma.user.findUnique({ where: { email } });

  if (!user || !bcrypt.compareSync(password, user.password)) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }

  const token = jwt.sign({ userId: user.id }, 'SECRET_KEY', { expiresIn: '1h' });
  res.json({ token });
});
```

3. Secure Routes with Middleware:

```
const authenticate = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).json({ error: 'Unauthorized' });

  jwt.verify(token, 'SECRET_KEY', (err, decoded) => {
    if (err) return res.status(401).json({ error: 'Invalid token' });
    req.userId = decoded.userId;
    next();
  });
};

app.get('/protected', authenticate, (req, res) => {
  res.json({ message: 'You are authenticated!' });
```

```
});
```

---

3. Frontend Implementation

1. Store Token in Local Storage:
```
localStorage.setItem('token', response.data.token);
```
2. Attach Token to Requests:
```
axios.defaults.headers.common['Authorization'] = `Bearer ${localStorage.getItem('token')}`;
```
3. Handle Token Expiry with Refresh Tokens:
    - Implement a `/refresh-token` endpoint to issue new tokens.

Key Takeaways

- JWT is used for secure authentication.
- Use middleware to protect routes in the backend.
- Store and attach tokens in the frontend.

Day 11: Connecting Backend and Frontend

Learning Objectives

- Connect the React frontend to the Node.js backend.
- Use `useEffect` for data fetching.
- Implement navigation with `useNavigate`.

---

1. Fetching Data with `useEffect`

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

const CustomerList: React.FC = () => {
  const [customers, setCustomers] = useState<Customer[]>([]);

  useEffect(() => {
    axios.get('/api/customers')
      .then((response) => setCustomers(response.data))
      .catch((error) => console.error(error));
  }, []);

  return (
    <ul>
      {customers.map((customer) => (
        <li key={customer.id}>{customer.name}</li>
      ))}
    </ul>
  );
};
```

---

2. Navigation with useNavigate

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

const navigate = useNavigate();

const handleAddCustomer = () => {
  navigate('/add-customer');
};
```

---

3. Building the Full Application

- Combine all concepts:
    - **Frontend**: React, TypeScript, Redux, Tailwind CSS.
    - **Backend**: Node.js, Express, Prisma, MySQL.
    - **Authentication**: JWT.

Key Takeaways

- Use useEffect to fetch data from the backend.
- useNavigate enables programmatic navigation.
- Build a full-stack application by integrating frontend and backend.

Reference links

https://react.dev/reference/react

https://react-redux.js.org/api/hooks

https://reactrouter.com/home

https://tailwindcss.com/docs/installation/framework-guides

https://jwt.io/