Introduction to Web Development with Remix

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-05

Table des matières

0.1 Overview 1
0.2 JavaScript 2
0.2.a Overview 2
0.2.b What is JavaScript? 2
0.2.c Variables 2
0.2.d Data Types 2
0.2.e Conditional Statements 3
0.2.f Loops 3
0.2.g Defining Functions 3
0.2.h Event Handling 3
0.2.i DOM Manipulation 3
0.2.j What is Inheritance? 4
0.2.k Prototypal Inheritance in JavaScript 4
0.2.l Class Syntax 4
0.2.m What is TypeScript? 4
0.2.n TypeScript Features 4
0.2.o Product List 5
0.3 Summary 5
0.4 React 5
0.5 Remix 5
0.5.a Learning Remix By doing 6
0.5.b Step 1: Create a New Remix Project 6
0.5.c Step 2: Project Structure**: 6
0.5.d Detailed Project Structure 6
0.5.e Step 4: Start the Development Server 6
0.6 A Remix Page 7
1 The Root Page and routes 8
1.0.a Usage of $ in Route 9
1.0.b Product Details Page 9
1.0.c How Routes Work in Remix 9
1.0.d Benefits of Using Routes 10
1.0.e Understanding Layout Routes in Remix 10
1.0.f Benefits 10
1.1 Components in Remix 10
1.1.a Form Component 10
1.1.b Link Component 11
1.1.c NavLink Component 11
1.2 Data Management in Remix 12
1.2.a Prisma Syntax 12
1.2.b User Model 13
1.2.c Order Model 13
1.2.d Update the database schema . 13
1.2.e Custom Queries in Prisma 13
1.2.f Querying the Database 14
1.3 Summary 15
1.4 Additional Resources 15
1.5 Note 16
1.6 Hands-on Practice 16

0.1 Overview

- **Topics**:
  - Introduction to JavaScript
  - Introduction to React
  - Introduction to Remix
  - Developing E-commerce Websites with Remix

0.2 JavaScript

- **What is JavaScript?**
  - JavaScript is a programming language used to create dynamic and interactive web content.
  - It is one of the core technologies of the web, alongside HTML and CSS.
- **Why JavaScript?**
  - Adds interactivity to websites (e.g., form validation, dynamic content updates).
  - Widely supported by all modern web browsers.
  - Extensive ecosystem with numerous libraries and frameworks.

0.2.a Overview

- **Topics**:
  - What is JavaScript?
  - Basic Syntax and Data Types
  - Control Structures
  - Functions
  - Event Handling and DOM Manipulation
  - Inheritance
  - TypeScript: Interest and Features

0.2.b What is JavaScript?

- **Definition**:
  - JavaScript is a programming language used to create dynamic and interactive web content.
  - It is one of the core technologies of the web, alongside HTML and CSS.
- **Why JavaScript?**
  - Adds interactivity to websites (e.g., form validation, dynamic content updates).
  - Widely supported by all modern web browsers.
  - Extensive ecosystem with numerous libraries and frameworks.
  - Can also be used for server-side development.

0.2.c Variables

- **Declaring Variables**:
  - In JavaScript, variables can be declared using `let`, `const`, or `var`.
  - **`let`**: Used to declare variables that can be reassigned.
  - **`const`**: Used to declare variables that cannot be reassigned.
  - **`var`**: An older way to declare variables, generally not recommended due to its function-scoped behavior.
  - **Examples**:

```
// Using let for a variable that can be reassigned
let productName = "Laptop";
productName = "Smartphone"; // Reassigning the variable

// Using const for a variable that cannot be reassigned
const price = 999.99;
// price = 899.99; // This will cause an error

// Using var (not recommended)
var stock = 50;
stock = 30; // Reassigning the variable
```

- **Best Practices**:
  - Prefer `let` and `const` over `var` to avoid issues with variable scoping.
  - Use `const` by default, and only use `let` if you need to reassign the variable.
  - Avoid using `var` to prevent unexpected behavior due to its function-scoped nature.

0.2.d Data Types

- **Common Data Types**:
  - String, Number, Boolean, Array, Object

- Example:

```
let productName = "Laptop"; // String
let price = 999.99; // Number
let inStock = true; // Boolean
let categories = ["Electronics", "Computers"]; // Array
let product = { name: "Laptop", price: 999.99 }; // Object
```

0.2.e Conditional Statements

- **if-else Statement**:
  - Example:

```
let stock = 50;

if (stock > 0) {
  console.log("In stock");
} else {
  console.log("Out of stock");
}
```

0.2.f Loops

- **for Loop**:
  - Example:

```
let products = ["Laptop", "Smartphone", "Tablet"];

for (let i = 0; i < products.length; i++) {
  console.log(products[i]);
}
```

0.2.g Defining Functions

- **Function Declaration**:
  - Example:

```
function calculateTotal(price, quantity) {
  return price * quantity;
}

let total = calculateTotal(999.99, 2);
console.log(total); // 1999.98
```

- **Arrow Function**:
  - Arrow functions provide a shorter syntax for writing function expressions in JavaScript.

```
const calculateTotal = (price, quantity) => price * quantity;

let total = calculateTotal(999.99, 2);
console.log(total); // 1999.98
```

0.2.h Event Handling

- **Adding Event Listeners in the Browser**:
  - Event listeners allow you to execute code in response to user interactions.

```
<button id="helloButton">Click Me</button>
<script>
 document.getElementById("helloButton").addEventLis
() => {
    alert("Hello, World!");
  });
</script>
```

  - The `addEventListener` method attaches a click event listener to the button with the ID `helloButton`.
  - When the button is clicked, an alert message « Hello, World! » is displayed.
  - This example demonstrates how to handle events in the browser using JavaScript.

0.2.i DOM Manipulation

- **Changing Content in the Browser**:

- DOM manipulation allows you to change the content of HTML elements dynamically.
- **Example**:

```
<h1 id="productTitle">Product Name</h1>
<script>
 document.getElementById("productTitle") = "Laptop";
</script>
```

0.2.j What is Inheritance?

- **Definition**:
  - Inheritance is a feature of object-oriented programming that allows a class to inherit properties and methods from another class.
  - Promotes code reuse and establishes a natural hierarchy between classes.

0.2.k Prototypal Inheritance in JavaScript

- OPTIONAL FOR NON COMPUTER SCIENCE STUDENTS
- **Prototype Chain**:
  - JavaScript uses prototypal inheritance, where objects inherit properties and methods from a prototype object.

```
function Product(name, price) {
  this.name = name;
  this.price = price;
}

Product.prototype.display = function() {
  console.log(`${this.name} costs $${this.price}`);
};

let laptop = new Product("Laptop", 999.99);
laptop.display(); // Laptop costs $999.99
```

0.2.l Class Syntax

- ES6 introduced class syntax to simplify prototypal inheritance.

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }

  display() {
    console.log(`${this.name} costs $${this.price}`);
  }
}

let laptop = new Product("Laptop", 999.99);
laptop.display(); // Laptop costs $999.99
```

0.2.m What is TypeScript?

- **Definition**:
  - TypeScript is a superset of JavaScript that adds static types.
  - Developed and maintained by Microsoft.
- Why TypeScript?
- **Advantages**:
  - **Type Safety**: Helps catch errors at compile time rather than runtime.
  - **Improved IDE Support**: Enhanced code completion, navigation, and refactoring.
  - **Scalability**: Better suited for large-scale applications.

0.2.n TypeScript Features

- **Static Typing**:
  - Example:

```
let productName: string = "Laptop";
let price: number = 999.99;
let inStock: boolean = true;
```

- **Interfaces**:
  - Example:

```
interface Product {
  name: string;
  price: number;
}

let laptop: Product = {
  name: "Laptop",
  price: 999.99
};
```

0.2.o Product List

- **Displaying a List of Products computed on the client side**:

```
<ul id="productList"></ul>
<script>
    let products = ["Laptop", "Smartphone", "Tablet"];
 let productList = document.getElementById("productList");

  products.forEach(product => {
            let listItem = document.createElement("li");
    listItem.textContent = product;

productList.appendChild(listItem);
  });
</script>
```

0.3 Summary

- **Key Points**:
  - JavaScript is essential for creating dynamic web content.
  - Understanding basic syntax, data types, control structures, and functions is crucial.
  - Event handling and DOM manipulation enable interactivity.
  - Inheritance promotes code reuse and establishes a natural hierarchy between classes.
  - TypeScript adds static types to JavaScript, improving type safety and scalability.
  - Practical examples in e-commerce demonstrate how to build functional websites.

0.4 React

- **What is React?**
  - React is a JavaScript library for building user interfaces.
  - Developed and maintained by Facebook.
  - Allows developers to create reusable UI components.
- **Why React?**
  - Component-based architecture promotes reusability and maintainability.
  - Efficient rendering with the Virtual DOM.
  - Strong community support and extensive ecosystem.
- **Context in This Lecture**:
  - In this lecture, we will primarily focus on Remix, a framework that relies on React.
  - Understanding the basics of React will help you grasp how Remix leverages React to build modern web applications.

0.5 Remix

- **What is Remix?**
  - Remix is a full-stack web framework for building modern web applications.
  - Built on top of React, it focuses on performance and user experience.
  - Provides server-side rendering (SSR) and static site generation (SSG).
- **Why Remix?**
  - Simplifies data loading and routing.
  - Enhances performance with server-side rendering.

- Integrates seamlessly with React.
- https://remix.run/docs/en/main

0.5.a Learning Remix By doing

- With remix Stacks are essentially templates/ starter kits that provide the groundwork for a complete project right out of the box.
- For our project, we will use the Blues Stack, which gives us a fully configured Remix project with Prisma, Tailwind, and an entire module that shows how to use these tools to build a CRUD feature.
- **Step 0: Install Node.js and npm**
  - **Node.js**: This is a tool that allows you to run JavaScript on your computer, not just in a web browser.
  - **npm**: This is a tool that helps you install and manage software packages written in JavaScript.

  To install both, download and install Node.js from the official website. npm comes bundled with Node.js.

0.5.b Step 1: Create a New Remix Project

- Run the following command in your terminal:

```
npx create-remix --template remix-run/blues-stack ama
```

- This command will create a new folder named `ama` and set up the entire project after you answer `yes` to a couple of questions.

0.5.c Step 2: Project Structure**:

- **Prisma Directory**:
  - Contains your database schema and migration files.
- **Public Directory**:
  - Contains any assets the app needs, such as icons, images, etc.
- **App Directory**:
  - Houses all your code, both client and server.

0.5.d Detailed Project Structure

Prisma Directory: - **Purpose**: Manages database schema and migrations. - **Contents**: - `schema.prisma`: Defines your database schema. - Migration files: Track changes to your database schema over time.

Public Directory: - **Purpose**: Stores static assets. - **Contents**: - Icons, images, and other static files that your application needs.

App Directory: - **Purpose**: Contains the main application code. - **Contents**: - **Routes**: Define the different pages and API endpoints of your application. - **Components**: Reusable UI components. - **Styles**: Tailwind CSS configuration and custom styles. - **Utils**: Utility functions and helpers. - **Data**: Database access and data fetching logic.

0.5.e Step 4: Start the Development Server

- **database configuration**:
  - The Blues Stack uses PostgreSQL as the database. You need to set up a PostgreSQL database and configure the connection in the `.env` file. If needed correct the database URL in the `.env` file: default is `DATABASE_URL="postgresql://<USERNAME>:<PASSWORD>@loca<DATABASE_NAME>"` with your database credentials (postgres for both ?).

```
DATABASE_URL="postgresql://<USERNAME>:<PASSWORD>@loca<DATABASE_NAME>"
```

- **Initial Setup**:
  - Before starting the development server, you need to set up the project dependencies and environment variables.
  - Run the following command to install dependencies and set up the environment:

```
npm run setup
```

- **Run the First Build**:
  - After setting up the project, you need to build it for the first time. This will compile the project and prepare it for development:

```
npm run build
```

- **Start the Development Server**:
  - Finally, start the development server to run your project locally:

```
npm run dev
```

  - This will start the server, and you can view your project by opening a web browser and going to `http://localhost:3000`.
- **Development URL**:
  - Open your web browser and navigate to:

```
http://localhost:3000
```

  - You should see a Remix application to manage notes running locally.

By following these steps, you will have your development server up and running, allowing you to view and interact with your Remix project in real-time.

- https://github.com/remix-run/blues-stack#development

This starts your app in development mode, rebuilding assets on file changes.

0.6 A Remix Page

- **Page Description**: In Remix, a page is described and rendered using JavaScript code.
- **File Structure**: Each page corresponds to a file in the `app/routes` directory.
- **URL Path**: The file name and its path determine the URL path for the page.
- **Content Definition**: The content of the page is defined using React components.
- **The root page**: The root page of the application is typically defined in a file named `root.jsx` in the project directory.

```
  // app/routes/dummyProducts.tsx
  import { Link } from 'remix';

        export    default    function
DummyProducts() {
    return (
      <div>
        <h1>Dummy Products</h1>
        <ul>
          <li>Product 1 - $10.00</li>
          <li>Product 2 - $20.00</li>
          <li>Product 3 - $30.00</li>
        </ul>
         <Link to="/">Back to Home</
Link>
      </div>
    );
  }
```

```
// app/routes/dummyProducts.tsx

//   Import   necessary   types   and
functions from Remix
import type { LoaderFunctionArgs }
from "@remix-run/node";
import { json } from "@remix-run/
node";
import   {   useLoaderData   }   from
"@remix-run/react";

// Mock data for three products
const products = [
   { id: 1, name: 'Laptop', price:
999.99 },
  { id: 2, name: 'Smartphone', price:
499.99 },
```

```
   { id: 3, name: 'Tablet', price: 299.99 },
];

// Loader function to provide data to the component
// This function will be replaced with a database query in a real application
export const loader = async ({ request }: LoaderFunctionArgs) => {
   return json(products); // Return the mock products data as JSON
};

// Default export function for the products page
export default function DummyProductsPage() {
  const data = useLoaderData<typeof loader>(); // Use the loader data in the component

  return (
    <div className="min-h-screen bg-gray-100 p-4">
      {/* Container for the content, centered and with padding */}
          <div  className="max-w-4xl mx-auto bg-white shadow-md rounded-lg p-6">
        {/* Title of the page */}
       <h1 className="text-3xl font-bold mb-4">Products</h1>
        {/* List of products */}
        <ul className="space-y-4">
          {data.map(product => (
                <li  key={product.id} className="border-b pb-4">
             {/* Product name */}
            <div className="text-xl font-semibold">{product.name}</div>
             {/* Product price */}
              <div className="text-gray-600">${product.price}</div>
            </li>
          ))}
        </ul>
      </div>
    </div>
  );
}
```

1 The Root Page and routes

- **Root Page Definition**:
  - The root page of the application is typically defined in a file named `root.tsx` in the `app/routes` directory.
  - This file corresponds to the root URL (`/`). When a user navigates to the root URL, Remix will render the content defined in this file.
- **App Component**:
  - This is the root component of your Remix application, following standard HTML structure.
- **Understanding Routes in Remix**:
  - **Definition**: A route is a path in your web application that corresponds to a specific page or component.
  - **Purpose**: Routes help in navigating between different pages or views in your application.
- **File-Based Routing**:
  - The file `root.tsx` in the `app/routes` directory corresponds to the root URL (`/`).
  - When a user navigates to the root URL, Remix will render the content defined in this file.
  - This structure helps in organizing the application and makes it easy to manage different pages and their content.

1.0.a Usage of `$` in Route

- **Dynamic Segment**: The `$` symbol in the file name `products/$productId.jsx` indicates that `productId` is a dynamic segment. This means that any value in the URL that matches this segment will be captured as a parameter.
  - **Example URL**: `/products/1` or `/products/2`
  - **Captured Parameter**: In the URL `/products/1`, the `productId` parameter will capture the value `1`.
- **Accessing Parameters**: The captured parameter can be accessed in the `loader` function using `params`.
  - **Example**: `params.productId` will give the value of `productId` from the URL.
- **Dynamic Routing**: This allows you to create dynamic routes that can handle different values for the same route pattern.
  - **Example**: The same route file `products/$productId.jsx` can handle URLs like `/products/1`, `/products/2`, etc., and render the appropriate product details based on the `productId` parameter.

This feature is useful for creating pages that need to display different content based on the URL, such as product details pages, user profiles, etc.

1.0.b Product Details Page

- **Explanation**: In Remix, the `$` symbol in a route file name indicates a dynamic segment. This allows the route to capture a part of the URL as a parameter.
- **Example**: Create a file named `products.$productId.tsx in theapp/routes` directory.

```
import { useLoaderData } from 'remix';

export let loader = async ({ params }) => {
  let products = [
      { id: 1, name: 'Laptop', price: 999.99, description: 'A high-performance laptop.' },
    { id: 2, name: 'Smartphone', price: 499.99, description: 'A latest model smartphone.' },
  ];
    return products.find(product => product.id === parseInt(params.productId));
};

export default function Product() {
  let product = useLoaderData();
  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      <p>${product.price}</p>
    </div>
  );
}
```

1.0.c How Routes Work in Remix

- **File-Based Routing**: In Remix, routes are defined by the file structure in the `app/routes` directory.
  - Each file in the `app/routes` directory represents a route.
  - The file name and its path determine the URL path for the route.
- **Product Details Page Route**:
  - File: `app/routes/products/$productId.jsx`
  - URL: `/products/:productId`
  - Example:

```
import { useLoaderData } from
'remix';

export let loader = async
({ params }) => {
  let products = [
      { id: 1, name: 'Laptop',
price: 999.99, description: 'A
high-performance laptop.' },
    { id: 2, name: 'Smartphone',
price: 499.99, description: 'A
latest model smartphone.' },
  ];
    return products.find(product
=>         product.id         ===
parseInt(params.productId));
};

export default function Product()
{
  let product = useLoaderData();
  return (
    <div>
      <h1>{product.name}</h1>
     <p>{product.description}</p>
      <p>${product.price}</p>
    </div>
  );
}
```

1.0.d Benefits of Using Routes

- **Organized Structure**: Routes help in organizing your application's structure.
- **Easy Navigation**: Users can easily navigate between different pages.
- **Dynamic Content**: Routes can be dynamic, allowing you to create pages that display different content based on the URL parameters.

1.0.e Understanding Layout Routes in Remix

- **Layout Routes**:
  - In Remix, the _ prefix in `.tsx` files typically indicates that the file is a layout route.
  - Layout routes are used to define common layouts or components that should be shared across multiple nested routes.
  - These files are not directly accessible via a URL but instead provide a structure for other routes.
- **_root.tsx**:
  - This file would be used to define the layout for the root route.
  - Any nested routes would inherit this layout, ensuring a consistent structure across your application.

1.0.f Benefits

- **Code Organization**:
  - Helps in organizing the code by clearly separating layout components from specific route components.
  - Makes it easier to manage and update common layouts across multiple pages.
- **Reusability**:
  - Promotes reusability of layout components, reducing code duplication.
  - Ensures a consistent look and feel across different parts of the application.

1.1 Components in Remix

- Components are reusable pieces of UI that can be used throughout your application.
- They help in breaking down the UI into smaller, manageable parts.
- **Benefits**:
  - **Reusability**: Write once, use anywhere.
  - **Maintainability**: Easier to manage and update.
  - **Separation of Concerns**: Keeps different parts of the UI isolated.

1.1.a Form Component

- **Purpose**: Handles form submissions.

- **Example** (create a file named `contact.jsx` in the `app/routes` directory):

```
import { Form } from 'remix';

export default function Contact() {
  return (
    <div>
      <h1>Contact Us</h1>
     <Form method="post" action="/
submit-contact">
        <label>
          Name:
               <input type="text"
name="name" />
        </label>
        <label>
          Email:
              <input type="email"
name="email" />
        </label>
                              <button
type="submit">Submit</button>
      </Form>
    </div>
  );
}
```

1.1.b Link Component

- **Purpose**: Provides client-side navigation between routes.
- **Example** (create a file named `welcome.jsx` in the `app/routes` directory):

```
import { Link } from 'remix';

export default function Welcome() {
  return (
    <div>
      <h1>Welcome to My E-commerce
Store</h1>
      <nav>
        <ul>
          <li>
          <Link to="/products">View
Products</Link>
          </li>
          <li>
                          <Link  to="/
contact">Contact Us</Link>
          </li>
        </ul>
      </nav>
    </div>
  );
}
```

- **Explanation**:
  - The `Link` component is imported from Remix.
  - The `Welcome` component renders a heading and a navigation menu with links to the `/products` and `/contact` routes.
  - Clicking on these links will navigate to the respective routes without a full page reload, providing a smoother user experience.

1.1.c NavLink Component

- **Purpose**: The `NavLink` component in Remix is similar to the `Link` component but provides additional styling for active links. This helps in indicating the current active route to the user.
- **Usage**:
  - The `NavLink` component is used to create navigational links that automatically apply an active class when the link's route matches the current URL.
- **Example**:
  - Create a file named `header.jsx` in the `app/components` directory:

```
import { NavLink } from 'remix';

export default function Header()
{
  return (
    <header>
     <h1>My E-commerce Store</h1>
      <nav>
        <ul>
          <li>
                 <NavLink to="/"
end activeClassName="active">
              Home
            </NavLink>
          </li>
          <li>
          <NavLink to="/products"
activeClassName="active">
              Products
            </NavLink>
          </li>
          <li>
           <NavLink to="/contact"
activeClassName="active">
              Contact
            </NavLink>
          </li>
        </ul>
      </nav>
    </header>
  );
}
```

- **Explanation**:
  - The `NavLink` component is imported from Remix.
  - The `Header` component renders a heading and a navigation menu with `NavLink` components for the `/`, `/products`, and `/contact` routes.
  - The `activeClassName="active"` prop applies the `active` class to the link when its route matches the current URL.
  - This provides visual feedback to the user, indicating which page they are currently on.

By using the `NavLink` component, you can create a more user-friendly navigation experience in your Remix application.

1.2 Data Management in Remix

- In a Remix application, managing data efficiently is crucial for building dynamic and responsive web applications.
- This can be done manually with JavaScript or by using tools like Prisma for database management.
- **What is Prisma?**
  - Prisma is an open-source ORM (Object-Relational Mapping) tool for Node.js and TypeScript.
  - **ORM**: An ORM is a tool that allows you to interact with your database using an object-oriented paradigm. It maps database tables to classes and rows to objects, simplifying database operations.
  - Prisma simplifies database access and management by providing a type-safe API.
  - **Type-Safe API**: Ensures that the types of data you work with in your code match the types defined in your database schema, reducing runtime errors.
  - For more information, visit the Prisma Documentation.

1.2.a Prisma Syntax

High-Level Summary:

- **Entity/Table Definition**:
  - An entity or table definition starts with `model <EntityName> {}`.
  - Inside the curly braces `{}`, you define all the columns/properties of the entity and relationships with other entities.
  - Example: A table for comments would look like `model Comment {}`.

- **Column Definitions**:
  - Column definitions usually look like `<columnName> <columnType> <default/relationship/other specifiers>`.
  - Example: If our comment entity requires a column to store the content of the comment input by the user, it would look like:
- **Product Model**:
  - Defines a product entity with columns for id, name, price, and description.
  - install the `prisma` vscode extension.
  - add entity `Product` to the `prisma/schema.prisma` file:

```
model Product {
   id              Int       @id
@default(autoincrement())
  name        String
  price       Float
  description String?
      createdAt         DateTime
@default(now())
}
```

1.2.b User Model

- Defines a user entity with columns for id, name, email, and a relationship to the orders placed by the user.

```
model Customer {
    id              Int        @id
@default(autoincrement())
  name      String
  email     String  @unique
  orders    Order[]
}
```

1.2.c Order Model

- Defines an order entity with columns for id, total amount, and a relationship to the user who placed the order.
- Relationships between 2 tables/entities are usually defined via a foreign key column so these are also defined alongside other columns. The definition usually requires 2 lines. A column to contain the foreign key id and the other to specify the name used to access related entity which usually looks like: @relation(fields: [], references: [id], onDelete: Cascade, onUpdate: Cascade)

```
model Order {
    id              Int        @id
@default(autoincrement())
  total     Float
            createdAt      DateTime
@default(now())
  userId    Int
 user    User    @relation(fields:
[userId], references: [id])
}
```

1.2.d Update the database schema

- Prisma provide a database migration tool to ease schema changes. To apply the changes to the database, you need to run the following command:

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

- This command will create a new migration file in the `prisma/migrations` directory and apply the changes to the database. You should see changes in the database schema after running this command.
- Essential for e-commerce applications (e.g., products, orders, customers).

1.2.e Custom Queries in Prisma

- **Introduction**:
  - Prisma allows you to write custom queries to interact with your database.

- These queries can be used to fetch, update, delete, or aggregate data based on your application's requirements.

- **Example**:
  - Suppose you want to fetch all orders placed by a specific user. You can write a custom query using the Prisma client.

```
// app/data/orders.ts
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export async function getOrdersByUserId(userId: number) {
  return await prisma.order.findMany({
    where: {
      userId: userId,
    },
    include: {
      user: true,
    },
  });
}
```

- **Explanation**:
  - The `getOrdersByUserId` function takes a `userId` as a parameter and returns all orders placed by that user.
  - The `findMany` method is used to fetch multiple records from the `order` table.
  - The `where` clause filters the orders based on the `userId`.
  - The `include` clause is used to fetch related user data along with the orders.
- **SQL Query Generation**:
  - Prisma automatically generates the SQL queries needed to interact with your database based on the Prisma Client methods you use.
  - This means you don't have to write raw SQL queries; Prisma handles the query generation for you.
- **JavaScript Instances**:
  - Prisma also generates JavaScript instances of your database records.
  - When you fetch data using Prisma Client, it returns JavaScript objects that you can easily work with in your application.

By writing custom queries in Prisma, you can efficiently interact with your database and fetch the data you need for your Remix application. Prisma handles the SQL query generation and creates JavaScript instances of your data, simplifying database operations.

1.2.f Querying the Database

- Use the Prisma client to query the database in your Remix loaders or actions. For example, create a file named `products.jsx` in the `app/routes` directory:

```
import { useLoaderData } from 'remix';
import prisma from '../prisma';

export let loader = async () => {
  const products = await prisma.product.findMany();
  return products;
};

export default function Products() {
  let products = useLoaderData();
  return (
    <div>
      <h1>Products</h1>
      <ul>
        {products.map(product => (
          <li key={product.id}>
            {product.name} - ${product.price}
```

```
          </li>
        ))}
      </ul>
    </div>
  );
}
```

1.3 Summary

- **Key Points**:
  - **JavaScript**:
    - Essential for creating dynamic web content.
    - Understanding basic syntax, data types, control structures, and functions is crucial.
    - Event handling and DOM manipulation enable interactivity.
  - **React**:
    - A powerful library for building user interfaces.
    - Component-based architecture promotes reusability and maintainability.
    - Efficient rendering with the Virtual DOM.
  - **Remix**:
    - Simplifies full-stack web development with React.
    - Provides server-side rendering (SSR) and static site generation (SSG).
    - Enhances performance and user experience.
  - **E-commerce Examples**:
    - Practical examples demonstrate how to build functional websites.
    - Includes setting up routes, handling dynamic segments, and creating reusable components.
  - **Data Management**:
    - Connecting to relational databases is crucial for managing persistent data.
    - Prisma simplifies database access and management with a type-safe API.
    - Efficiently manage data in your Remix application using Prisma and PostgreSQL.
  - **Tailwind CSS**:
    - A utility-first CSS framework that allows for highly customizable and responsive designs.
    - Provides a set of utility classes that can be combined to create custom designs without writing custom CSS.
    - Offers greater flexibility and customization compared to component-based frameworks like Bootstrap.

By understanding these key points, you will have a solid foundation for building modern web applications using JavaScript, React, Remix, Prisma, and Tailwind CSS. The practical examples and hands-on practice will help reinforce your learning and enable you to apply these concepts in real-world scenarios.

By understanding these key points, you will have a solid foundation for building modern web applications using JavaScript, React, and Remix. The practical examples and hands-on practice will help reinforce your learning and enable you to apply these concepts in real-world scenarios.

1.4 Additional Resources

- **JavaScript**:
  - JavaScript Documentation: Comprehensive guide and reference for JavaScript.
  - JavaScript Info: In-depth tutorials and articles on JavaScript.
  - ECMAScript 2024 (ES15) Features: Latest ECMAScript standard features.
- **TypeScript**:
  - TypeScript Documentation: Official documentation for TypeScript, including setup, language features, and compiler options.
  - TypeScript Handbook: Comprehensive guide to TypeScript.

- **React**:
  - React Documentation: Official documentation for React, including tutorials and API reference.
  - React Tutorial: Step-by-step guide to building a React app.
- **Remix**:
  - Remix Documentation: Official documentation for Remix, covering setup, routing, data loading, and more.
  - Remix Blog: Articles and updates from the Remix team.
- **PostgreSQL**:
  - PostgreSQL Documentation: Official documentation for PostgreSQL, including installation, configuration, and usage.
  - PostgreSQL Tutorial: Tutorials and guides for learning PostgreSQL.
- **Prisma**:
  - Prisma Documentation: Official documentation for Prisma, including setup, schema, and API reference.
  - Prisma Examples: Collection of example projects using Prisma.
- **Tailwind CSS**:
  - Tailwind CSS Documentation: Official documentation for Tailwind CSS, including setup, customization, and utility classes.
  - Tailwind CSS Cheat Sheet: A handy cheat sheet for quickly referencing Tailwind CSS classes.

These resources will help you deepen your understanding of JavaScript, TypeScript, React, Remix, PostgreSQL, Prisma, and Tailwind CSS, and provide valuable reference material as you build your web applications.

1.5 Note

- **Quick Bootstrap**:
  - This lecture serves as a quick bootstrap to get you started with web development using Remix.
  - It provides an overview of essential concepts and practical examples to help you begin building web applications.
  - **Further Study**:
    - This is not a comprehensive course. To gain a deeper understanding, further study and practice are highly recommended.
    - Explore the additional resources provided to expand your knowledge and skills.
  - **Hands-On Practice**:
    - Engage in hands-on practice by setting up projects, experimenting with code, and building your own applications.
    - Use the provided Git repository and examples as a reference to guide your learning journey.
  - **Continuous Learning**:
    - Web development is a constantly evolving field. Stay updated with the latest trends, tools, and best practices.

1.6 Hands-on Practice

- **Practice**:
  - **Git Repository**:
    - A Git repository with Remix examples is provided.
    - Clone the repository to access the examples and use them as a reference for your own projects.
    - Repository URL: GitHub Repository
  - **Set Up a Remix Project**:
    - Follow the steps to create a new Remix project using the Blues Stack.
    - Create basic pages and routes to understand the project structure.
  - **Add Interactivity and Dynamic Content**:
    - Experiment with adding interactivity using event listeners and dynamic content using state management.
  - **Build an E-commerce Website**:

  - Use the examples provided to build your own e-commerce website.
  - Implement features such as product listings, product details, and a shopping cart.
- **Connect to a PostgreSQL Database**:
  - Set up a PostgreSQL database and configure the connection in your Remix application.
  - Perform basic queries to fetch and display data from the database.

By following these practice exercises, you will gain hands-on experience with Remix and build a functional e-commerce website. The provided Git repository will serve as a valuable resource for learning and reference.