

NEXT.JS



Next.js

WORKSHOP

Fast-Loading

NEXT

Server Rendering

SERVER-SIDE RENDERING

Fast-Loading

Next

FAST-LOADING
RENDERING

SERVER-SIDE
RENDERING

Hi, my name is Basti 🖐️



- Sebastian Springer
- React & Node.js
- Munich

Agenda

- Setup
- Static Rendering
- Error Handling
- Layout
- Styling
- Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

What are we building today?

Our goal is to build an application for managing books. You will be able to add new books, edit their details, remove them, and rate them.

Setup

<https://github.com/sspringer82/ctwebdev>

Agenda

- 🖱️ Setup
- Static Rendering
- Error Handling
- Layout
- Styling
- Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Create-Next-App

```
$ npx create-next-app@latest
```

Create-Next-App

```
$ npx create-next-app@latest  
What is your project named? ... next-app
```

Choose a suitable name for your project. In our example, it's `next-app`. This will be the name of the folder created in the current directory, which will contain the application.

```
$ npx create-next-app@latest  
✓ What is your project named? ... next-app  
Would you like to use TypeScript? ... No / Yes
```

Naturally, we're using TypeScript.

Next.js will handle the setup and configuration, including the installation of all necessary packages, so we can start working with TypeScript right away.

Create-Next-App

```
$ npx create-next-app@latest  
✓ What is your project named? ... next-app  
✓ Would you like to use TypeScript? ... Yes  
Would you like to use ESLint? ... No / Yes
```

We are using ESLint, a powerful tool to maintain high code quality and consistency. By default, it includes the rule sets `next/core-web-vitals` and `next/typescript`. You can customize ESLint's behavior in the `.eslintrc.json` file located in the root directory of your application.

For more information, visit the [official ESLint website](https://eslint.org/).

Create-Next-App

```
$ npx create-next-app@latest
✓ What is your project named? ... next-app
✓ Would you like to use TypeScript? ... Yes
✓ Would you like to use ESLint? ... No / Yes
Would you like to use Tailwind CSS? ... No / Yes
```

Tailwind CSS is a utility-first CSS framework that provides low-level utility classes to build custom designs directly in your markup. It allows for rapid UI development by composing small, reusable classes to style elements without writing custom CSS. This approach promotes consistency and reduces the need for context switching between HTML and CSS files.

For more information, visit the [official Tailwind CSS documentation](#).

Create-Next-App

```
$ npx create-next-app@latest
✓ What is your project named? ... next-app
✓ Would you like to use TypeScript? ... Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... Yes
Would you like your code inside a `src/` directory? ... No / Yes
```

The `src/` directory in Next.js is for organizing application code separately from configuration files. When used, Next.js looks for main application files (like pages, components, styles, etc.) inside this directory, keeping the project root cleaner. Most Next.js projects do not use this directory.

Create-Next-App

```
$ npx create-next-app@latest
✓ What is your project named? ... next-app
✓ Would you like to use TypeScript? ... Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... Yes
✓ Would you like your code inside a `src/` directory? ... No
Would you like to use App Router? (recommended) ... No / Yes
```

Next.js uses a file system-based approach for routing. The App Router, introduced in Version 13, replaces the older Pages Router. The new App Router provides greater control and flexibility. By default, components in the App Router are Server Components, meaning they are rendered on the server and then sent to the client.

Create-Next-App

```
$ npx create-next-app@latest
✓ What is your project named? ... next-app
✓ Would you like to use TypeScript? ... Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... Yes
✓ Would you like your code inside a `src/` directory? ... No
✓ Would you like to use App Router? (recommended) ... Yes
Would you like to use Turbopack for next dev? ... No / Yes
```

- **Turbopack:** A new, fast bundler introduced by the Vercel team, intended as a successor to Webpack.
- **Modern JavaScript Applications:** Focuses on incremental and parallel compilation for faster builds and better developer experience.
- **Built with Rust:** Optimizes bundling performance by leveraging efficient memory usage and parallelization.
- **Speed Improvements:** Promises substantial speed improvements, particularly for large-scale Next.js projects.
- **Deep Integration with Next.js:** Offers enhanced developer workflows and productivity.
- **Early Stages:** Not yet recommended for production use.

Create-Next-App - Import Alias

```
$ npx create-next-app@latest
✓ What is your project named? ... next-app
✓ Would you like to use TypeScript? ... Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... Yes
✓ Would you like your code inside a `src/` directory? ... No
✓ Would you like to use App Router? (recommended) ... Yes
✓ Would you like to use Turbopack for next dev? ... Yes
Would you like to customize the import alias (@/* by default)? ... No / Yes
```

- **Custom Path Mapping:** Import aliases allow you to create **custom path mappings** using the `paths` property in `tsconfig.json`, making imports more concise and readable.
- **Simplified Imports:** They help avoid long relative paths by allowing imports like `@components/Button` instead of `../.././../components/Button`.
- **Improved Code Maintainability:** Aliases make refactoring easier, as you can change the base path in the config without updating all import statements throughout the project.

(Optional)React Compiler

- **Build-Time Optimization:** React Compiler optimizes React apps at build time by automatically memoizing components and hooks, reducing unnecessary re-renders and improving performance.
- **Automatic Memoization:** It applies fine-grained reactivity similar to manual optimizations like `useMemo` or `useCallback` but without developer intervention.
- **ESLint Plugin Support:** Comes with an ESLint plugin to detect violations of React rules.
- **Handles Expensive Computations:** Memoizes heavy calculations during rendering within components.
- **Beta Stage:** Currently in beta, it supports React 17+ and requires following the Rules of React.
- **Compatibility Check:** The React Compiler checks your code for compatibility with the defined [“Rules of React”](#), ensuring that your components and hooks are optimized for this build-time transformation and that your code adheres to React’s best practices.

React Compiler

Install the Babel Plugin

```
npm install babel-plugin-react-compiler
```

Change the next.config.tsfile:

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    reactCompiler: true,
  },
}

export default nextConfig
```


Setup

Once you have completed all the prompts, the setup assistant will generate the file and folder structure for your application and install all necessary dependencies.

Structure

- Root directory: includes configuration files such as `package.json`, linter and TypeScript configurations, and Next.js specific settings
- `app` directory: this is where you place the Page components of your application in order to configure the App Router of Next
- `.next` directory: contains the build output generated by Next.js, including all compiled files and assets necessary for your application.
- `node_modules`: contains all the npm-installed dependencies and packages, along with their own dependencies, necessary for your application to run correctly.
- `public` directory: contains static assets such as images and fonts, which can be accessed directly via the root URL without any processing or transformations.

Start your engine

```
$ npm run dev
```




your brand new Next.js application is available at <http://localhost:3000>

NEXT.JS

1. Get started by editing `app/page.tsx` **have fun!**.
2. Save and see your changes instantly.

 Deploy now

Read our docs

 Learn  Examples  Go to nextjs.org →

Task 1: setup your application

Static Rendering

Agenda

- ~~Setup~~
- 📌 Static Rendering
- Error Handling
- Layout
- Styling
- Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Server Side Rendering (SSR) by default

- Components in Next.js are Server Components by default
- They are rendered on the server at build time by default
- Component functions can be async and you can use serverside APIs
- No lifecycle, no state
- [Documentation](#)

Benefits of SSR

- **Optimized Data Fetching:** Server Components fetch data directly from the server, improving performance by reducing client requests.
- **Enhanced Security:** Keeps sensitive information like tokens and API keys securely on the server.
- **Caching Benefits:** Server-rendered results can be cached for better performance and lower costs.
- **Improved Performance:** Minimizes client-side JavaScript load, aiding users with slower devices.
- **Faster Initial Load:** Renders HTML on the server, leading to quicker page load and First Contentful Paint (FCP).
- **SEO and Social Sharing:** Renders HTML for better indexing by search engines and social previews.
- **Streaming Support:** Streams page parts as they render, enhancing perceived speed.

Next Pages

- Next.js uses the file based App Router
- You can define new routes as folders and files in the app folder
- Page components must be named page.tsx
- /app/users/page.tsx => <http://localhost:3000/users>
- Page components are of the NextPage type

Example: Static rendered

`/app/users/page.tsx`: here you see an example of a page component

- Async function
- Contains server side data fetching
- Handles error state, if something went wrong while getting the data
- Handles empty state, if there are no datasets to be displayed
- Shows the data
- Built entirely at build time

Task 2: Create your own Server Component

Repo URL: <https://github.com/sspringer82/ctwebdev>

- add a folder books to the app folder
- create a file named `page.tsx`
- Implement an async Component named `BooksListPage` of the type `NextPage`
- Use the `fetch` api to get allBooks from `http://localhost:3001/books`
- Optional: put the function in a file `/app/api/book.api.ts`
- Show the data in a table
- Bonus: Take care of errors in the communication and display a corresponding error message
- Bonus: Show a message if there are no book records

Layout

Agenda

- ~~Setup~~
- ~~Static Rendering~~
- ~~Error Handling~~
- 📌 Layout
- Styling
- Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Layout

- Component that wraps the content of a page with common elements (header, footer, sidebar)
- One layout can be used for multiple pages. Via a common parent folder
- Can be nested
- The page is inserted as `children`
- The layout is not re-rendered at navigation
- There needs to be at least a root layout (in the `app` directory). It contains the `html` and `body` tags.
- Layouts have the name `layout.tsx`
- Templates (template.tsx) are similar to layouts but they are re-rendered at navigation
- [Documentation](#)

Task 3: Create a layout

- Add a new Legal page `/app/legal/page.tsx` with some dummy content
- open the `app/layout.tsx` file
- add a header and footer element
- add a link to the legal page in the footer with the `Link` component and the `href` attribute
- Add a `layout.tsx` to the `/app/books` directory and add a headline
- [Link](#)

Styling

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- 🙌 Styling
- Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

CSS

- Inline CSS: style attribute to style elements directly. Uses JavaScript objects and only affects the element it is applied to.
- Global CSS/CSS Import: place CSS selectors and rules in a regular CSS file and import the file in a layout or page file. Applies to all elements currently in the browser document.
- CSS Modules: Create a file with the Extension `.module.css`. Only class selectors are available. Use a default import and apply the classes as `className` for elements.
- [Documentation](#)

Sass

- **Sass Overview:** Sass (Syntactically Awesome Stylesheets) is a CSS preprocessor that adds features like variables, nesting, and mixins, making CSS more organized and maintainable.
- **Built-in Support:** Next.js has built-in support for Sass, allowing you to directly import .scss or .sass files into your components without extra configuration.
- **File Usage:** Simply create .scss files and import them in your components or app.js for global styles. For (component) scoped styles, use .module.scss.
- **Optimizations:** Next.js optimizes Sass imports and supports CSS Modules to prevent style conflicts and improve maintainability.
- [Documentation](#)

Css-in-js

- CSS-in-JS Overview: CSS-in-JS allows writing CSS directly within JavaScript, enabling dynamic styles based on component state or props.
- Usage in React: Libraries like Styled-Components and Emotion are popular for creating styled components and scoping styles in React.
- Next.js Integration: Next.js supports CSS-in-JS with seamless SSR, ensuring styles are correctly rendered on both server and client.
- [Documentation](#)

Tailwind

- Tailwind CSS is a utility-first framework with pre-defined classes to build custom designs quickly and efficiently.
- Utility Classes: It uses small, reusable classes like flex and mt-4, allowing direct styling in HTML or JSX.
- Next.js Integration: Tailwind is easily set up in Next.js either during project creation or in an already existing application.
- Optimization: It purges unused styles in production, optimizing bundle size and performance in Next.js projects.
- [Documentation](#)

Task 4: Apply some Styling

- Use tailwind in order to style the root layout.
- some ideas:
 1. Style the Body Element: Apply Tailwind classes to the `<body>` element, including the custom font variables, antialiasing, and a background and text color. Use the classes `${geistSans.variable} ${geistMono.variable} antialiased bg-gray-50 text-gray-900`.
 2. Create a Header with a Centered Title: Add a `<header>` element with a fixed position, a white background, and a shadow. Use Tailwind classes like `bg-white shadow-md fixed top-0 left-0 w-full z-10`. Inside the header, create a container that centers its content using `flex` and `justify-center` classes. Inside this container, add an `<h1>` element with a bold, large font size using `text-2xl font-bold`.
 3. Add a Main Section with Proper Padding: Create a `<main>` section below the header that includes the container `mx-auto px-4` classes for centering the content and setting responsive padding. Add `pt-20` to provide enough top padding for the fixed header.
 4. Style the Footer for Consistency: Create a `<footer>` at the bottom of the page, using Tailwind classes `bg-white shadow-md fixed bottom-0 left-0 w-full p-4 flex justify-between items-center text-sm`. The footer should include a `<p>` tag for the company name and current year, and a legal link using `Link` with `hover:text-gray-600` for a subtle hover effect.

Material UI

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- 🙌 Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Material UI

- Overview: Material-UI (MUI) is a React component library based on Google's Material Design, offering pre-built UI components for creating responsive and modern interfaces.
- Component-Based: MUI provides a wide range of customizable components like buttons, forms, tables, and dialogs for consistent and efficient UI development.
- Theming: It offers robust theming and styling options to easily match project-specific branding and design requirements.
- Responsive and Accessible: MUI supports responsive design and accessibility standards, ensuring a great user experience across all devices and users.

Setup Material UI

```
npm add @mui/material @emotion/react @emotion/styled @mui/icons-material --force  
npm install @mui/material-nextjs @emotion/cache --force
```

- Integrate the `AppRouterCacheProvider`. It collects the material css and appends it to the tag.
- Create a Material UI theme and add the roboto font in order to optimize it.
- (optional): add `cssVariables` to the theme
- Add the roboto font to the body element
- Integrate the `ThemeProvider`
- [Documentation](#)

Task 5: Styling with Material UI components

Use some material components in the BooksListPage component. Here are some hints:

1. **Import Material-UI Components:** Import the necessary Material-UI components like `Container`, `Typography`, `Table`, `Alert`, etc., at the top of the file.
2. **Wrap the Main Content:** Replace the main wrapper `<div>` with a Material-UI `Container` for consistent padding and centering.
3. **Implement Error and Empty States:** Use Material-UI's `Alert` to display error messages and `Typography` for showing the empty state message.
4. **Refactor Table Structure:** Replace the standard HTML table elements with Material-UI's `Table`, `TableHead`, `TableRow`, `TableCell`, and wrap it in a `TableContainer` with a `Paper` component.
5. **Add Hover Effect for Rows:** Apply a hover effect to the `TableRow` using the `sx` prop for visual feedback on row interaction.
6. **Test and Validate:** Ensure all components render correctly, and verify that dynamic content, error handling, and interactions work as expected.

Dynamic Rendering

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- 📌 Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Dynamic Routes

- Variables in the route path. Folder name wrapped in []
- Path `/app/books/[id]/page.tsx` => `/books/42`
- Access the variable via `params`
- Component is not rendered at build time

Dynamic Rendering

- Routes are rendered for each user at request time.
- Useful for personalized content or content that depends on information e.g. in cookies or the search params.
- Switch from static to dynamic: dynamic apis and or uncached data
- Dynamic APIs: cookies, headers, connection, searchParams Prop
- Dynamic APIs are asynchronous and you have to await them e.g. ``const { id } = await params``
- Uncached data: `fetch('http://localhost:3001/users', {cache: 'no-store'})`

Link and Prefetching

- Link component to link to another Route with the href attribute
- If a link comes into the viewport the target gets prefetched
- You can use the `useRouter` and the `router.prefetch` to manually prefetch.
- Disable prefetch: Set the prefetch attribute of a Link component to false

Example: dynamic routes

- User Details
- Link from list to details
- build
- prefetching

Task 6: Create a detail page for the books

- create a new page component `app/books/[id]/page.tsx`
- the component should be named `BookDetailPage`
- access the variable in the url path via the `params.id` prop of the component function. Take care of the `params` promise and use the correct type.
- fetch the data for the book and show the content
- Go to the List-component and add a `Link` Component to every list entry to enable navigating to the details viewport (Hint: use the `href`-prop)
- Bonus: use material ui and/or tailwind to make it good looking
- Bonus: add a Link from the details page back to the list
- Bonus: put the data fetching function into a separate file `/api/books.api.ts`
- Bonus: take care of errors and display an according message

Static Dynamic Rendering

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- 👉 Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Static Dynamic Routes

- Dynamic routes (the ones with variables) are rendered on request.
- You can implement and export a `generateStaticParams` function in your page file.
- This function is called at build time and returns an array of possible values for the dynamic route.
- Next.js will generate a static page for each value in the array.

Example 7: Static rendering of the user details

- generateStaticParams implementation
- build
- show details
- Bonus: skip one user and request it

Task 7: Static rendering of dynamic routes

- Implement the `generateStaticParams` function that fetches all possible id values
- create a build of your application and run it
- test the statically generated pages

Loading & Streaming

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- 🙌 Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Loading

- A `loading.tsx` creates a suspense boundary that shows a loading indicator.
- Is used for dynamic rendered content

Example: Loading

- Loading.tsx for /app/users/[id]
- util/wait.ts to delay the response
- load user details page

Streaming

1. **Server-Side Data Loading:** When a page loads server-side data using `getServerSideProps` or a `fetch` call within an RSC (React Server Component), the `loading.tsx` is displayed until the data is fetched.
2. **Client-Side Navigation:** During navigation between pages that haven't been preloaded, Next.js shows the `loading.tsx` until the new page is fully loaded.
3. **Suspense-Based Loading:** Next.js uses React's `Suspense` for asynchronous data loading. If you use `await` in your Server Components and wrap them in a `Suspense` component, `loading.tsx` is shown until the data is available.
4. **Layout Loading Delays:** When using a layout (`layout.tsx`) and the underlying content changes, Next.js shows `loading.tsx` if there's a significant delay in loading the child components.

Example: Streaming

- Create new component `/app/users/streaming/[id]/page.tsx` with two suspense components
- create api functions to get user name and user email each delayed
- create a Name and an Email component that load the data and put them into the suspense components
- load the page

Task 8: Loading and Streaming

- implement a loading.tsx for the books details page
- delay the request for the details by 10 seconds
- Test if the loading indicator is shown correctly
- split the display of the book details in two parts (book data + rating)
- create a function that allows you to delay the request of the book data
- fetch the details of the book separately in two sub components wrapped in suspense boundaries
- Bonus: instead of a simple loading text use the skeleton component of material ui

API Routes

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- 🙌 API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

API Routes

- API Routes allow you to create backend endpoints in Next.js
- Supported HTTP methods: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS
- You can access the incoming request (NextRequest) and work with the outgoing response (NextResponse)
- Similar to a page with the page.tsx, you just need to create a file named route.ts
- Within the route file you implement the endpoint functions named like the HTTP method
- Dynamic Route Segments: via the second parameter, an async params object
- Request Body: consume via the first parameter (type Request)
 - Regular Body: `await request.json()`
 - FormData: `await request.formData()`
- Revalidate Data: `revalidatePath`

Documentation

- [Route Handlers](#)
- [revalidate Path](#)

Example: Delete API

- create a new file `/app/users/api/[id]/route.ts`
- implement a DELETE function
- test it with API client

Task 9: API Route for Rating a book

- Create a new file ``/app/books/api/[id]/route.ts``
- Implement a PUT-handler function
- the handler function receives the id in the url and the updated book in the request body
- update the record of the book with the new information
- invalidate the path for list (`/books`) and details so that the change will take effect

Client Components

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- ~~API Routes~~
- 👉 Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Client Components

- Prerendered on the server and sent to the client
- Interactivity: State, effect (lifecycle) and event listeners
- Browser APIs (e.g. geolocation, localStorage)
- Marked with the "use client" directive.
- [Documentation](#)

How are client components rendered?

server side:

1. Server components are rendered to RSC and includes references to client components
2. RSC payload and client component JS are rendered to HTML

client side:

1. HTML is directly rendered
2. With the RSC payload the component tree is created and the DOM is updated
3. Client components are hydrated and made interactive

Example: Client component - delete user

- create a new file `/app/users/components/delete.tsx`
- create a button and a click handler that deletes the user using the API route
- create a router object with `useRouter` and use the `refresh` method to update the ui

Task 10: Client Component for Book rating

- Create a custom rating component `/app/books/components/rating.tsx`
- include the rating component in the books list
- the component receives the book data in order to display the current rating value in form of star icons (filled and unfilled)
- By clicking on a star you can update the rating
- Use the PUT API route
- Bonus: update the component state without a reload

Error Handling

Agenda

- ~~Setup~~
- ~~Static Rendering~~
- 🙌 Error Handling
- Layout
- Styling
- Material UI
- Dynamic Rendering
- Static Dynamic Rendering
- Streaming
- API Routes
- Client Components
- Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Error Handling

- Special files `error.tsx` and `global-error.tsx` to create an error boundary
- Used to handle unexpected errors
- Catches exceptions in their child components and displays a fallback component

Example: Error Handling

- Create a new file `error.tsx` in `/app/users/[id]/`
- add `throw error` to rethrow the error
- disable the delay + change the url so that the request fails

Task 11: Error handling

- Create an error boundary for the book detail page `/app/books/[id]/error.tsx`
- change the url so that the request fails, rethrow the error within the component so that the error boundary is triggered

Search Params

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- ~~API Routes~~
- ~~Client Components~~
- 📌 Search Params
- Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Search Params

- Static rendering: wrap `useSearchParams` in `suspense` - will be client side rendered
- Dynamic rendering: available during the initial server render of the client component
- Server Component: access data via the `searchParams` Props - leads to dynamic rendering

Example: Extended Search

- Extract the list from the list page to a separate component
- Add search capability to api function
- Create a new search page `/app/users/search/page.tsx`
- include the list
- create a search form `/app/users/components/search.tsx`
- hand over the form values to the url
- fill the form with the search params using `useSearchParams`
- access the search params in the page with the `searchparams` Prop

Task 12: Search feature

Create the possibility to search for books via a single input that searches the fields title and author

- (optional): install `react-hook-form` for the form handling
- Extract the book list into a separate component
- Create a new page `/app/books/search`
- create form with a single input element that enables the search
- the search form should be prefilled by the search value in the search params
- the form submits to the url
- the page takes the search params and filters the result
- enable filtering the server results in the api function

Form Handling + Server Functions

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- ~~API Routes~~
- ~~Client Components~~
- ~~Search Params~~
- 👉 Form Handling + Server Functions
- Server Function in Client Components
- Auth
- Testing

Server Functions aka. Server Actions

- Functions that allow client components to call async functions on the server
- React 19 feature
- Indicated with the "use server" directive
- Can use server side functionality like Server Components. Typically used to modify data
- different possibilities:
 - Server Function in a Server Component
 - Import a Server Function into a Client Component
 - Server Function with Actions
 - Server Functions with Form Actions
 - Server Functions with useActionState

we will have a closer look at the last one (and later on the second one)

useActionState

```
const [state, formAction] = useActionState(fn, initialState, permalink?);
```

- `fn`: async function that is called when the form is submitted, can be a Server Function
- `state`: is used to represent form errors
- `permalink`: is used to handle form submission if the client component is not yet hydrated
- Connect the `formAction` function to the `action` attribute of the form

Example: Create new User

- Create a new file `/app/actions/user.ts` with a Server Function
- Create a new file `/app/users/form/page.tsx` with a form to create a new user
- Connect the Server Action to the form using the `useActionState` hook and the `action` attribute of the form
- Add a Link from the list to the form

Task 13: Create and update books

- Create a new file `/app/actions/book.ts` with a Server Function that takes care of saving the data
- (optional) create a separate API function to save the data
- Create a new page `/app/books/form/page.tsx` with a form to create a new book
- Create a new page `/app/books/form/[id]/page.tsx` with a form to update a book
- (optional) extract the form and reuse the structure
- call the Server Function when submitting the form
- (optional) use `react-hook-form` for the form handling
- (optional) use `startTransition` to call the Server Function, use an Error Boundary
- Add links to the `List` component to create and edit books.

Server Function in Client Components

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- ~~API Routes~~
- ~~Client Components~~
- ~~Search Params~~
- ~~Form Handling + Server Functions~~
- 👉 Server Function in Client Components
- Auth
- Testing

Server Functions in Client Components

- Server Functions can be imported in Client Components
- Client Component is able to execute the Server Function

Example: Delete user

- Implement a deleteUser Server Function
- The `/app/users/components/delete.tsx` component is already a Client Component
- Instead of communicating with the server, use the Server Function
- Errors can be caught and displayed in the component using a try-catch block

Task 14: Delete a book

- Create a Server Function to delete a book by a given id
- (optional) extract the logic to an API function.
- Create a Client Component `/app/books/components/delete.tsx` with a button and call the Server Function on click
- Integrate the Delete component in the books list

Auth

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- ~~API Routes~~
- ~~Client Components~~
- ~~Search Params~~
- ~~Form Handling + Server Functions~~
- ~~Server Function in Client Components~~
- 🙌 Auth
- Testing

Let's Code together: Authentication with Next Auth and the Credentials Provider

- Install `next-auth@beta`
- Create `/auth.config.ts` responsible for protecting pages e.g. `"/books"`
- Create `/auth.ts` which is responsible for credential checking
- Create `/middleware.ts` to enable NextAuth
- Implement a Server Function `/app/actions/auth.ts` that uses the `signIn` method of `/auth.ts`
- Implement a login page `/app/login/page.tsx` and a form `/app/login/components/form.tsx` that integrates the Server Function
- (optional) add a logout button to the root layout that calls `signOut` from `/auth.ts`

Testing

Agenda

- Setup
- ~~Static Rendering~~
- ~~Error Handling~~
- ~~Layout~~
- ~~Styling~~
- ~~Material UI~~
- ~~Dynamic Rendering~~
- ~~Static Dynamic Rendering~~
- ~~Streaming~~
- ~~API Routes~~
- ~~Client Components~~
- ~~Search Params~~
- ~~Form Handling + Server Functions~~
- ~~Server Function in Client Components~~
- ~~Auth~~
- 🙌 Testing

vitest setup

testing a client component

testing a client component with interaction

testing a client component with server interaction

testing a server component

Contact

Sebastian Springer

MaibornWolff GmbH München

sebastian.springer@maibornwolff.de