Chapter 1: Getting Started with Next.js

Creating a New Project

Use pnpm as Package Manager: Faster than npm or yarn.

```
npm install -g pnpm
```

Create Next.js App: Use CLI to set up a new project with a starter example.

```
npx create-next-app@latest nextjs-dashboard --example "https://github.com/vercel/next-learn/tree/main/dashboard/starter-example" --use-pnpm
```

Exploring the Project

Goal: Focus on learning Next.js features with pre-existing code.
Folder Structure:
/app: Main application logic and routes.
/app/lib: Reusable functions (e.g., data fetching).
/app/ui: Pre-styled UI components.
/public: Static assets (images, etc.).
Config Files: Pre-configured files like `next.config.js`.

Placeholder Data

Use Placeholder Data: For development without a live database.
Located in `/app/lib/placeholder-data.ts`.
Example: `invoices` table structure.

TypeScript in Project

File Suffixes: .ts and .tsx indicate TypeScript.
Types: Defined in `/app/lib/definitions.ts`.

```
export type Invoice = {
  id: string;
  customer_id: string;
  amount: number;
  date: string;
  status: 'pending' | 'paid';
};
```

Type-Safety Tools: Consider Prisma or Drizzle for automatic type generation.

Running the Development Server
Install Packages:

```
pnpm i
```

Start Development Server:

```
pnpm dev
```

Access Project: Open http://localhost:3000 to view the home page.

Chapter 2: CSS Styling

1. **Overview of Styling Options**
    - Adding global CSS files.
    - Using Tailwind CSS and CSS Modules.
    - Toggling class names with clsx utility.
2. **Global Styles**
    - `/app/ui/global.css` contains site-wide styles.
    - Import global styles in `/app/layout.tsx`.
3. **Tailwind CSS**
    - Utility-first CSS framework for quick styling.
    - Add classes like `text-blue-500` directly in JSX for element-specific styling.
    - Included automatically if selected during `create-next-app`.
4. **CSS Modules**
    - Scoped, component-specific CSS to prevent style conflicts.
    - Create unique classes for each component (e.g., `/app/ui/home.module.css`).
    - Use `import styles from ...` for applying CSS module classes.
5. **Conditional Styling with clsx**
    - Apply classes based on conditions using `clsx` library.
    - Example: style an InvoiceStatus component based on `status` (e.g., `bg-gray-100` for "pending" and `bg-green-500` for "paid").
6. **Other Styling Solutions**
    - Sass for `.scss` files.
    - CSS-in-JS libraries like styled-jsx, styled-components, and emotion.

CSS Styling Overview

1. **Global Styles**: Use `/app/ui/global.css` for site-wide styles. Import it in `/app/layout.tsx`.
2. **Tailwind CSS**: Utility classes for quick styling directly in JSX (e.g., `text-blue-500`). Available when selected in `create-next-app`.
3. **CSS Modules**: Component-scoped styles to avoid conflicts. Import module (e.g., `home.module.css`) and apply using `styles.className`.
4. **Conditional Styling with clsx**: Toggle classes based on conditions using `clsx` (e.g., change style based on status).
5. **Other Options**: Sass and CSS-in-JS (e.g., styled-components)

Chapter 3: Optimizing Fonts and Images

In this chapter, you'll learn to enhance your Next.js site with custom fonts and images for improved performance and design.

Key Topics:

1. Adding custom fonts with `next/font`.
2. Using optimized images with `next/image`.

3. Benefits of font and image optimization in Next.js.

Why Optimize Fonts?

Custom fonts improve design but may cause layout shifts as they load, affecting performance. Next.js preloads fonts with `next/font`, avoiding extra network requests and enhancing page load speed.

Quiz: How does Next.js optimize fonts?

- **Correct Answer**: D) It hosts font files with other static assets, so there are no additional network requests.

Adding Fonts

1. **Create a font file**: Define the primary font, e.g., Inter, in `/app/ui/fonts.ts`.
2. **Apply the font**: Use it in `<body>` in `/app/layout.tsx` for global styling.

Practice: Add a secondary font, e.g., Lusitana, for specific text elements.

Why Optimize Images?

Using `next/image` improves:

- Responsiveness across devices.
- Avoidance of layout shifts.
- Lazy loading for images outside the viewport.

Adding a Hero Image

In `/app/page.tsx`:

1. Import `next/image`.
2. Add the desktop hero image with width and height for stable loading.
3. Set visibility for desktop screens (`md:block`) and hide on mobile (`hidden`).

Practice: Add a mobile-specific image that only shows on smaller screens.

Chapter 4: Creating Layouts and Pages

In this chapter, you'll learn how to create new pages and layouts in your Next.js app, allowing you to add more routes beyond just the home page. Here's what you'll cover:

- Setting up routes with file-based routing.
- Creating nested layouts to share UI across pages.
- Understanding colocation, partial rendering, and the root layout.

Nested Routing

Next.js uses folders to create nested routes, with each folder mapping to a URL segment. For example, `/app/dashboard/page.tsx` corresponds to `/dashboard`.

To create a dashboard page:

1. Create a folder `/app/dashboard`.
2. Inside, add `page.tsx`:

```
tsx
Copy code
export default function Page() {
  return <p>Dashboard Page</p>;
}
```

Visit `http://localhost:3000/dashboard` to see the "Dashboard Page" text.

Practice: Adding More Pages

Create more routes within the dashboard:

- **Customers Page**: `/dashboard/customers` with `<p>Customers Page</p>`.
- **Invoices Page**: `/dashboard/invoices` with `<p>Invoices Page</p>`.

Dashboard Layout

Add a shared layout using `layout.tsx` in `/dashboard`:

```
tsx
Copy code
import SideNav from '@/app/ui/dashboard/sidenav';

export default function Layout({ children }) {
  return (
    <div className="flex h-screen flex-col md:flex-row">
      <SideNav />
      <div className="flex-grow p-6">{children}</div>
    </div>
  );
}
```

This layout wraps dashboard pages and adds a shared sidebar. The layout remains static on navigation, and only page content updates—a feature called **partial rendering**.

Root Layout

The root layout (`/app/layout.tsx`) wraps all pages, allowing you to set global settings for `<html>` and `<body>`. Use this layout for universal elements like fonts, metadata, or global styles.

Chapter 5: Navigating Between Pages

This chapter covers adding links for navigation in your Next.js app.

Topics:

- Using the `<Link>` component from Next.js.
- Highlighting active links with `usePathname()`.
- Understanding optimized navigation in Next.js.

Why Use `<Link>`?

Traditional `<a>` tags trigger full page refreshes. In Next.js, the `<Link />` component enables client-side navigation, making transitions smoother without refreshing the whole page.

Using `<Link>`

Replace `<a>` tags with `<Link />` for navigation:

In `/app/ui/dashboard/nav-links.tsx`:

```
tsx
Copy code
import Link from 'next/link';

export default function NavLinks() {
  return (
    <>
      {links.map((link) => (
        <Link
          key={link.name}
          href={link.href}
          className="flex h-[48px] items-center gap-2 rounded-md p-3 text-sm
font-medium hover:bg-sky-100 hover:text-blue-600"
        >
          <link.icon className="w-6" />
          <p className="hidden md:block">{link.name}</p>
        </Link>
      ))}
    </>
  );
}
```

Using `<Link />` enables client-side navigation and smooth transitions without page reloads.

Optimized Navigation

Next.js automatically optimizes navigation with:

- **Code splitting**: Loads only necessary code for each route, isolating errors.
- **Prefetching**: In production, Next.js preloads linked routes in the viewport, making navigation nearly instant.

By using `<Link />`, you create a faster, app-like experience for users.

Chapter 6: Setting Up Your Database

This chapter guides you through setting up a PostgreSQL database with @vercel/postgres for your dashboard app.

Topics:

- Push your project to GitHub.
- Set up Vercel account and link your GitHub repo for deployment.
- Create and link a Postgres database.
- Seed the database with initial data.

Steps:

1. **Push to GitHub**:
   - Push your project to GitHub for easier setup and deployment.
2. **Create Vercel Account**:
   - Sign up at vercel.com/signup, link GitHub, and deploy your project.
3. **Create a Postgres Database**:
   - In Vercel Dashboard, navigate to **Storage > Connect Store > Postgres**.
   - Set database region to Washington D.C. for lower latency.
   - Copy database secrets to `.env.local` and install the Vercel Postgres SDK.
4. **Seed Database**:
   - Inside `/app/seed`, uncomment the `route.ts` file to seed the database using SQL.
   - Run `pnpm run dev`, visit `localhost:3000/seed` to seed the data. Delete this file after completion.
5. **Quiz**:
   - **Seeding** means populating the database with initial data.
6. **Explore Database**:
   - In Vercel, navigate to **Data** to see tables like users, customers, invoices, and revenue.
   - Use the **query** tab for SQL commands, like running a JOIN query to view invoice amounts.

This setup helps you build a fully-functional database-driven application in Next.js with Vercel.

Chapter 7: Fetching Data

This chapter explains how to fetch data for your Next.js application and build a dashboard overview page.

Topics:

- Approaches to fetching data (APIs, ORMs, SQL).
- Using Server Components for secure back-end access.
- Understanding network waterfalls.
- Implementing parallel data fetching.

Choosing How to Fetch Data

API Layer:

- APIs act as intermediaries between your app and database, especially when using third-party services or to avoid exposing database secrets on the client.
- You can create API endpoints in Next.js using Route Handlers.

Database Queries:

- For full-stack apps, you'll interact with databases via SQL or ORM.
- If using React Server Components, you can query the database directly without an API layer, keeping secrets secure.

Using Server Components to Fetch Data

- Server Components allow fetching data with async/await, without needing `useEffect` or data-fetching libraries.
- They run on the server, making data fetching more efficient by keeping logic on the server and only sending results to the client.

Chapter 8: Static and Dynamic Rendering

Explore static and dynamic rendering, understanding when and why to use each approach.

Static Rendering

- **Definition**: Renders data on the server at build time or revalidation, serving cached results.
- **Benefits**:
  - Faster load times due to caching
  - Reduces server workload
  - Improves SEO as content is prerendered
- **Ideal Use**: For static content like blogs, not suitable for frequently updating dashboards.

- **Quiz Question**: Why is static rendering not ideal for dashboards?
    - **Answer**: C – Application won't reflect latest data changes.

Dynamic Rendering

- **Definition**: Generates content at request time for the latest data.
- **Benefits**:
    - Real-time data updates
    - User-specific personalization
    - Access to request-time details (e.g., cookies, URL params)

Chapter 9: Streaming

Learn how streaming can improve user experience by progressively loading data.

What is Streaming?

- **Definition**: Sends the webpage in chunks, allowing visible parts to load before the entire page.
- **Benefits**:
    - **Parallel Loading**: Reduces load time as chunks render progressively.
- **Implementation in Next.js**:
    1. **Page-Level Streaming**: Create a `loading.tsx` file for a fallback UI.

```
javascript
Copy code
export default function Loading() {
  return <div>Loading...</div>;
}
```

2. **Loading Skeletons**: Use skeleton components for enhanced UX.

```
javascript
Copy code
import DashboardSkeleton from '@/app/ui/skeletons';
export default function Loading() {
  return <DashboardSkeleton />;
}
```

Chapter 10: Partial Prerendering (PPR)

Discover Partial Prerendering (PPR) in Next.js 14, merging static and dynamic rendering.

Static vs. Dynamic Routes

Most applications use one or the other, but PPR enables mixed rendering on the same page.

What is Partial Prerendering?

- **Definition**: Serves a static shell with placeholders ("holes") for dynamic content.
- **Quiz Question**: What are the holes in Partial Prerendering?
  - **Answer**: B – Locations where dynamic content will load asynchronously.

How to Implement PPR:

1. **Enable PPR in** `next.config.mjs`:

   ```
   javascript
   Copy code
   const nextConfig = {
     experimental: {
       ppr: 'incremental',
     },
   };
   export default nextConfig;
   ```

2. **Add PPR to Layout**:

   ```
   javascript
   Copy code
   export const experimental_ppr = true;
   ```

Chapter 11: Enhancing the Invoices Page with Search and Pagination

A guide to optimizing the `/invoices` page with search, pagination, and efficient URL handling.

1. **Components Overview**: Includes search, pagination, and invoice table components.
2. **Using URL Search Params**:
   - Benefits: Bookmarkable URLs, better server-side rendering, and improved analytics.
3. **Implementing Search**:
   - Use an `onChange` handler to capture input, sync URL parameters with `useSearchParams`.
4. **Syncing Search Input and URL**:
   - Set `defaultValue` to sync input with URL, eliminating the need for client-side state.
5. **Updating the Table**:
   - Pass current query and page to `<Table>`, which fetches filtered invoices.
6. **Debouncing for Optimization**:
   - Debouncing input avoids excessive server requests, improving efficiency.

This chapter introduces Next.js hooks (`useSearchParams`, `usePathname`, `useRouter`) to enhance the `/invoices` page, making it user-friendly and performant.

Chapter 12: Mutating Data

Enhance the Invoices page by adding functionality for creating, updating, and deleting invoices.

Key Topics Covered

- **React Server Actions**: Execute asynchronous data mutations directly on the server, eliminating API endpoints.
- **Form Handling in Server Components**: Use forms with Server Actions, automatically passing FormData for efficient data handling.
- **Type Validation with FormData**: Ensure type safety when managing data in forms.

Chapter 13: Handling Errors

Learn to handle errors gracefully using Next.js's error-handling tools and JavaScript's `try/catch`:

Key Concepts Covered

1. **Using `try/catch` in Server Actions**: Add `try/catch` to manage errors in Server Actions effectively.
2. **Error Boundary with `error.tsx`**: Define an error UI in `error.tsx` for any unexpected errors. This Client Component:
    - Accepts `error` and `reset` props.
    - Allows users to retry by re-rendering.
3. **Handling 404 Errors with `notFound`**: Use `notFound()` in routes when a resource is not found, triggering a 404 response. Create a custom `not-found.tsx` to show a 404 error message with navigation options back to safe pages.

These tools improve the user experience by managing both general and specific errors in your application Chapter 14, *"Improving Accessibility"*, where we focus on making web applications accessible and usable for everyone, covering topics like form validation, error handling, and ESLint accessibility checks.

Key Topics

- **Accessibility Best Practices**
  Implementing accessible design through semantic HTML, proper labeling, and focus state management ensures usability for those with disabilities.
- **ESLint Accessibility Plugin**
  Next.js integrates `eslint-plugin-jsx-a11y`, which detects accessibility issues (e.g., missing `alt` attributes). Running `next lint` catches these issues early.
- **Improving Form Accessibility**
  Using elements like `<input>`, `<label>`, and `htmlFor`, along with focus highlights, improves navigation for keyboard and screen reader users.

- **Form Validation**
  Validation (client-side with `required` and server-side with Zod) ensures data integrity and provides accessible error handling. Key approaches include:
  - **Client-Side**: Basic input requirements.
  - **Server-Side**: Secure validation using `useActionState` for real-time error handling.
- **Server Actions & Zod Integration**
  Zod allows schema definitions for form fields, ensuring structured input and manageable error messages.

Code Samples

1. **Server Validation with Zod**:

```
// actions.ts
export async function createInvoice(prevState, formData) {
  const validatedFields = FormSchema.safeParse({
    customerId: formData.get('customerId'),
    amount: formData.get('amount'),
    status: formData.get('status'),
  });

  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
      message: 'Missing Fields. Failed to Create Invoice.',
    };
  }

  try {
    await sql`...`; // Insert data into database
  } catch (error) {
    return { message: 'Database Error: Failed to Create Invoice.' };
  }
}
```

2. **UI Error Messages**:

```
javascript
Copy code
<form action={formAction}>
  <label htmlFor="customer">Choose customer</label>
  <select id="customer" name="customerId" aria-describedby="customer-
error">
    <option value="">Select a customer</option>
    {/* Options */}
  </select>
  {state.errors?.customerId && (
    <p id="customer-error"
role="alert">{state.errors.customerId[0]}</p>
  )}
</form>
```

This chapter helps developers proactively enhance accessibility with proper validation, React state handling, and a focus on error visibility, fostering an inclusive user experience.

Chapter 15: Adding Authentication

In this chapter, you'll integrate authentication into your app using **NextAuth.js**. Here's a breakdown:

1. **Authentication vs. Authorization**:
   - **Authentication**: Verifies the user's identity (e.g., username/password).
   - **Authorization**: Decides what parts of the app the user can access after authentication.
2. **Creating the Login Route**:
   Create a `/login` route with a login form.
3. **NextAuth.js Setup**:
   Install NextAuth.js and set up the secret key for encrypting sessions.
4. **Protecting Routes with Middleware**:
   Use middleware to redirect unauthenticated users to the login page when accessing protected routes.
5. **Password Hashing**:
   Hash passwords with bcrypt for security before storing them in the database.
6. **Adding Credentials Provider**:
   Set up a **Credentials provider** for login (username/password).
7. **Handling Sign In**:
   Use `authorize` to validate credentials and compare passwords with bcrypt.
8. **Updating the Login Form**:
   Use React's `useActionState` to handle form states and errors during login.

This chapter teaches you how to secure your app, handle authentication, and protect routes with NextAuth.js!

Chapter 16: Adding Metadata

Metadata enhances SEO and shareability. It's hidden information in the HTML `<head>` element, not visible to users but crucial for search engines and social media platforms.

Why is metadata important? It helps search engines index pages better, improving ranking, and optimizes how pages appear on social media.

Types of Metadata:

1. **Title:** The title displayed in the browser tab (`<title>`).
2. **Description:** A brief overview shown in search results (`<meta name="description">`).
3. **Keywords:** Relevant terms for indexing (`<meta name="keywords">`).

4. **Open Graph:** Optimizes content for social media sharing (`<meta property="og:title">`, `<meta property="og:image">`).
5. **Favicon:** Links to the site's icon (`<link rel="icon">`).

Adding Metadata in Next.js:

- **Config-based:** Use a static metadata object or dynamic function in `layout.js` or `page.js`.
- **File-based:** Use files like `favicon.ico`, `opengraph-image.jpg`, and `robots.txt` for static metadata.

Favicon & Open Graph: Place images like `favicon.ico` and `opengraph-image.jpg` in the `/app` folder for automatic use.

Page Title & Descriptions: You can set default titles in `layout.tsx` and override them in specific pages:

```
// /app/layout.tsx
import { Metadata } from 'next';

export const metadata: Metadata = {
  title: 'Acme Dashboard',
  description: 'Next.js Course Dashboard',
  metadataBase: new URL('https://your-site.com'),
};

// Specific page metadata
// /app/dashboard/invoices/page.tsx
export const metadata: Metadata = {
  title: 'Invoices | Acme Dashboard',
};
```

Template Titles: Use the `title.template` field to avoid repeating the company name across pages.

```
// /app/layout.tsx
export const metadata: Metadata = {
  title: { template: '%s | Acme Dashboard', default: 'Acme Dashboard' },
};
```

Now, titles will be dynamic across pages. Add metadata to various pages like `/login`, `/dashboard`, etc.

Next.js Metadata API allows full control over your app's metadata.

Chapter 17: Next Steps

Congratulations on completing the Next.js dashboard course! You've learned the core features of Next.js and best practices for web development.

Next Steps:

- Explore more Next.js features to build side projects, startups, or large-scale apps.
- Resources for further learning:
  - Next.js Documentation
  - Next.js Templates
  - (Admin Dashboard, Commerce,
  - Blog Kit,
  - AI Chatbot, Image Gallery)
  - Next.js Repository
  - Vercel YouTube

Share Your App: Share your project on X (Twitter) and tag @nextjs for feedback.

Keep building and learning!