Next.js App Router - Complete Routing Tutorial

From Basics to Advanced Patterns (Next.js 14/15/16)

Table of Contents

1. Introduction to App Router

What is the App Router?

The **App Router** is Next.js's modern routing system introduced in Next.js 13. It uses the `app/` directory and leverages React Server Components by default.

Key Differences: App Router vs Pages Router

Feature	**App Router (`app/`)**	**Pages Router (`pages/`)**
Default Components	Server Components	Client Components
Data Fetching	`async/await` in components	`getServerSideProps`, `getStaticProps`
Layouts	Nested, preserved on navigation	Re-render on every navigation
Loading States	Built-in `loading.tsx`	Manual implementation
Error Handling	Built-in `error.tsx`	Custom `_error.js`
Streaming	Native support	Limited

Project Structure Overview

```
my-app/
├── app/                    # App Router directory
│   ├── layout.tsx          # Root layout (required)
│   ├── page.tsx            # Home page (/)
│   ├── globals.css         # Global styles
│   └── [routes]/           # Route folders
├── public/                 # Static assets
├── src/                    # Optional: source directory
└── next.config.js          # Next.js configuration
```

---

2. Basic File-Based Routing

Core Concept

In Next.js App Router, **folders define routes** and **special files define UI**.

Special File Conventions

File	Purpose
`page.tsx`	Makes a route publicly accessible
`layout.tsx`	Shared UI wrapper for a segment
`loading.tsx`	Loading UI (Suspense fallback)
`error.tsx`	Error UI (Error boundary)
`not-found.tsx`	404 UI
`template.tsx`	Re-rendered layout (no state preservation)
`default.tsx`	Fallback for parallel routes
`route.ts`	API endpoint

Basic Route Examples

Example 1: Simple Pages

```
app/
├── page.tsx            →  /
├── about/
│   └── page.tsx        →  /about
├── contact/
│   └── page.tsx        →  /contact
└── blog/
    └── page.tsx        →  /blog
```

Code: `app/page.tsx`

```
// This is a Server Component by default
export default function HomePage() {
  return (
    <main>
      <h1>Welcome to My App</h1>
      <p>This is the home page.</p>
    </main>
  );
}
```

Code: `app/about/page.tsx`

```
export default function AboutPage() {
  return (
    <main>
      <h1>About Us</h1>
      <p>Learn more about our company.</p>
    </main>
  );
}
```

Example 2: Nested Routes

```
app/
└── blog/
    ├── page.tsx          →  /blog
    └── posts/
        └── page.tsx      →  /blog/posts
```

Route Segments

Each folder in the `app` directory represents a **route segment** that maps to a **URL segment**.

```
app/dashboard/settings/page.tsx
     ↓         ↓
   /dashboard/settings
```

---

3. Dynamic Routes

What are Dynamic Routes?

Dynamic routes allow you to create pages that depend on dynamic data (like IDs, slugs, etc.).

Syntax Patterns

Pattern	Example	Matches
`[param]`	`[id]`	`/products/1`, `/products/abc`
`[...param]`	`[...slug]`	`/blog/a`, `/blog/a/b/c`
`[[...param]]`	`[[...slug]]`	`/`, `/blog/a`, `/blog/a/b`

Example 1: Single Dynamic Segment

Folder Structure:

```
app/
└── products/
    ├── page.tsx          →  /products
    └── [id]/
        └── page.tsx      →  /products/1, /products/2, etc.
```

Code: `app/products/[id]/page.tsx`

```
// In Next.js 15+, params is a Promise
interface ProductPageProps {
  params: Promise<{ id: string }>;
}

export default async function ProductPage({ params }: ProductPageProps) {
  const { id } = await params;

  // Fetch product data
  const product = await getProduct(id);

  return (
    <main>
      <h1>Product: {product.name}</h1>
      <p>ID: {id}</p>
    </main>
  );
}

// Optional: Generate static params at build time
export function generateStaticParams() {
  return [
    { id: '1' },
    { id: '2' },
    { id: '3' },
```

```
  ];
}
```

Example 2: Catch-All Segments

Folder Structure:

```
app/
└── docs/
    └── [...slug]/
        └── page.tsx
```

Code: `app/docs/[...slug]/page.tsx`

```
interface DocsPageProps {
  params: Promise<{ slug: string[] }>;
}

export default async function DocsPage({ params }: DocsPageProps) {
  const { slug } = await params;

  // slug is an array: /docs/a/b/c → ['a', 'b', 'c']
  return (
    <main>
      <h1>Documentation</h1>
      <p>Path: {slug.join(' / ')}</p>
    </main>
  );
}
```

Matching Behavior:

URL	`slug` value
`/docs/react`	`['react']`
`/docs/react/hooks`	`['react', 'hooks']`
`/docs/react/hooks/useState`	`['react', 'hooks', 'useState']`

Example 3: Optional Catch-All Segments

```
app/
└── shop/
    └── [[...categories]]/
        └── page.tsx
```

Code: `app/shop/[[...categories]]/page.tsx`

```
interface ShopPageProps {
  params: Promise<{ categories?: string[] }>;
}

export default async function ShopPage({ params }: ShopPageProps) {
  const { categories } = await params;

  if (!categories) {
    return <h1>All Products</h1>;
  }

  return (
    <main>
      <h1>Category: {categories.join(' > ')}</h1>
    </main>
  );
}
```

Matching Behavior:

URL	`categories` value
`/shop`	`undefined`
`/shop/electronics`	`['electronics']`
`/shop/electronics/phones`	`['electronics', 'phones']`

4. Route Groups

What are Route Groups?

Route groups let you **organize routes without affecting the URL structure**. They're created using parentheses: `(folderName)`.

Use Cases

1. **Organize by feature/team**
2. **Create multiple root layouts**
3. **Opt specific segments into a layout**

Example 1: Organizing by Feature

```
app/
├── (marketing)/
```

```
│   ├── about/
│   │   └── page.tsx       →  /about
│   └── blog/
│       └── page.tsx       →  /blog
├── (shop)/
│   ├── products/
│   │   └── page.tsx       →  /products
│   └── cart/
│       └── page.tsx       →  /cart
└── page.tsx               →  /
```

Note: `(marketing)` and `(shop)` are NOT part of the URL.

Example 2: Multiple Root Layouts

```
app/
├── (auth)/
│   ├── layout.tsx         # Auth-specific layout (no navbar)
│   ├── login/
│   │   └── page.tsx       →  /login
│   └── register/
│       └── page.tsx       →  /register
└── (main)/
    ├── layout.tsx         # Main layout (with navbar)
    ├── page.tsx           →  /
    └── dashboard/
        └── page.tsx       →  /dashboard
```

Code: `app/(auth)/layout.tsx`

```
export default function AuthLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <div className="auth-container">
      <div className="auth-card">
        {children}
      </div>
    </div>
  );
}
```

Code: `app/(main)/layout.tsx`

```
import Navbar from '@/components/Navbar';

export default function MainLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <>
      <Navbar />
      <main>{children}</main>
    </>
  );
}
```

---

5. Layouts and Templates

Layouts

Layouts are **UI shared between multiple pages**. They preserve state and don't re-render on navigation.

Root Layout (Required)

Every app must have a root layout at `app/layout.tsx`:

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

export const metadata: Metadata = {
  title: 'My App',
  description: 'Welcome to my application',
};

export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

Nested Layouts

Layouts can be nested. Child layouts wrap inside parent layouts:

```
app/
├── layout.tsx              # Root layout
└── dashboard/
    ├── layout.tsx          # Dashboard layout (nested)
    └── page.tsx
```

Code: `app/dashboard/layout.tsx`

```
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <div className="dashboard">
      <aside className="sidebar">
        <nav>
          <a href="/dashboard">Overview</a>
          <a href="/dashboard/settings">Settings</a>
        </nav>
      </aside>
      <main className="content">{children}</main>
    </div>
  );
}
```

Rendering Result:

```
<RootLayout>
  <DashboardLayout>
    <DashboardPage />
  </DashboardLayout>
</RootLayout>
```

Templates

Templates are similar to layouts but **re-render on every navigation**. Use when you need:

- Fresh state on each navigation
- useEffect to run on each navigation
- Animation on enter/exit

Code: `app/template.tsx`

```
'use client';

import { motion } from 'framer-motion';

export default function Template({ children }: { children: React.ReactNode }) {
  return (
    <motion.div
      initial={{ opacity: 0, y: 20 }}
      animate={{ opacity: 1, y: 0 }}
      transition={{ duration: 0.3 }}
    >
      {children}
    </motion.div>
  );
}
```

Layout vs Template Comparison

Feature	Layout	Template
Re-renders on navigation	❌ No	✅ Yes
Preserves state	✅ Yes	❌ No
Effects re-run	❌ No	✅ Yes
Use case	Persistent UI	Animations, analytics

---

6. Loading UI and Streaming

Instant Loading States

The `loading.tsx` file creates an instant loading UI using React Suspense.

Folder Structure:

```
app/
└── products/
    ├── loading.tsx        # Shown while page.tsx loads
    └── page.tsx           # Async Server Component
```

Code: `app/products/loading.tsx`

```
export default function ProductsLoading() {
  return (
    <div className="loading-container">
      <div className="spinner" />
```

```
      <p>Loading products...</p>
    </div>
  );
}
```

Code: `app/products/page.tsx`

```
// This async component triggers the loading state
export default async function ProductsPage() {
  // Simulate slow data fetch
  const products = await fetch('https://api.example.com/products', {
    cache: 'no-store'
  }).then(res => res.json());

  return (
    <ul>
      {products.map((product: any) => (
        <li key={product.id}>{product.name}</li>
      ))}
    </ul>
  );
}
```

How It Works (Under the Hood)

Next.js automatically wraps `page.tsx` in a Suspense boundary:

```
// What Next.js does internally:
<Suspense fallback={<Loading />}>
  <Page />
</Suspense>
```

Streaming with Suspense

You can also use Suspense directly for more granular loading states:

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

export default function DashboardPage() {
  return (
    <div>
      <h1>Dashboard</h1>

      {/* This loads instantly */}
      <WelcomeMessage />
```

```
      {/* This streams in when ready */}
      <Suspense fallback={<StatsSkeleton />}>
        <Stats />
      </Suspense>

      {/* This also streams independently */}
      <Suspense fallback={<ChartSkeleton />}>
        <RevenueChart />
      </Suspense>
    </div>
  );
}

async function Stats() {
  const stats = await fetchStats(); // Slow API
  return <StatsCards data={stats} />;
}

async function RevenueChart() {
  const data = await fetchRevenueData(); // Another slow API
  return <Chart data={data} />;
}
```

Skeleton Loading Pattern

```
// app/products/loading.tsx
export default function ProductsLoading() {
  return (
    <div className="products-grid">
      {Array.from({ length: 6 }).map((_, i) => (
        <div key={i} className="skeleton-card">
          <div className="skeleton skeleton-image" />
          <div className="skeleton skeleton-title" />
          <div className="skeleton skeleton-price" />
        </div>
      ))}
    </div>
  );
}
```

CSS for Skeletons:

```
.skeleton {
  background: linear-gradient(
    90deg,
    #e0e0e0 25%,
    #f0f0f0 50%,
    #e0e0e0 75%
```

```
  );
  background-size: 200% 100%;
  animation: shimmer 1.5s infinite;
  border-radius: 4px;
}

@keyframes shimmer {
  0% { background-position: 200% 0; }
  100% { background-position: -200% 0; }
}

.skeleton-image { height: 200px; }
.skeleton-title { height: 24px; width: 80%; margin-top: 12px; }
.skeleton-price { height: 20px; width: 40%; margin-top: 8px; }
```

---

7. Error Handling

Error Boundaries with `error.tsx`

The `error.tsx` file creates a React Error Boundary for a route segment.

Important: `error.tsx` must be a Client Component.

Folder Structure:

```
app/
└── products/
    ├── error.tsx          # Catches errors in this segment
    ├── loading.tsx
    └── page.tsx
```

Code: `app/products/error.tsx`

```
'use client'; // Required!

import { useEffect } from 'react';

interface ErrorProps {
  error: Error & { digest?: string };
  reset: () => void;
}

export default function ProductsError({ error, reset }: ErrorProps) {
  useEffect(() => {
    // Log error to an error reporting service
    console.error('Products error:', error);
  }, [error]);
```

```
  return (
    <div className="error-container">
      <h2>Something went wrong!</h2>
      <p>{error.message}</p>

      {/* Reset button re-renders the segment */}
      <button onClick={reset}>
        Try again
      </button>
    </div>
  );
}
```

Error Boundary Hierarchy

Errors bubble up to the nearest error boundary:

```
app/
├── error.tsx              # Catches errors from entire app
├── layout.tsx
└── dashboard/
    ├── error.tsx          # Catches errors from dashboard/*
    ├── page.tsx
    └── settings/
        ├── error.tsx      # Catches errors from settings only
        └── page.tsx
```

Handling Errors in Layouts

`error.tsx` **cannot catch errors in** `layout.tsx` of the same segment (because error boundary wraps children, not siblings).

Solution: Use `global-error.tsx` at root level:

```
// app/global-error.tsx
'use client';

export default function GlobalError({
  error,
  reset,
}: {
  error: Error & { digest?: string };
  reset: () => void;
}) {
  return (
    <html>
      <body>
```

```
        <h2>Something went wrong!</h2>
        <button onClick={reset}>Try again</button>
      </body>
    </html>
  );
}
```

Error Recovery Pattern

```
'use client';

import { useState } from 'react';

export default function ProductsError({
  error,
  reset
}: {
  error: Error;
  reset: () => void
}) {
  const [isRetrying, setIsRetrying] = useState(false);

  const handleRetry = async () => {
    setIsRetrying(true);

    // Optional: Clear any cached data
    // await clearCache();

    // Reset the error boundary
    reset();
  };

  return (
    <div className="error-container">
      <div className="error-icon">⚠</div>
      <h2>Failed to load products</h2>
      <p className="error-message">
        {error.message || 'An unexpected error occurred'}
      </p>

      <div className="error-actions">
        <button
          onClick={handleRetry}
          disabled={isRetrying}
          className="btn btn-primary"
        >
          {isRetrying ? 'Retrying...' : 'Try Again'}
        </button>

        <a href="/" className="btn btn-secondary">
```

```
          Go Home
        </a>
      </div>
    </div>
  );
}
```

---

8. Not Found Pages

The `not-found.tsx` File

Handles 404 errors for a route segment.

Code: `app/not-found.tsx`

```
import Link from 'next/link';

export default function NotFound() {
  return (
    <div className="not-found">
      <h1>404</h1>
      <h2>Page Not Found</h2>
      <p>The page you're looking for doesn't exist.</p>
      <Link href="/">Go Home</Link>
    </div>
  );
}
```

Triggering Not Found Programmatically

Use the `notFound()` function to trigger a 404:

```
import { notFound } from 'next/navigation';

interface ProductPageProps {
  params: Promise<{ id: string }>;
}

export default async function ProductPage({ params }: ProductPageProps) {
  const { id } = await params;
  const product = await getProduct(id);

  // Trigger 404 if product doesn't exist
  if (!product) {
    notFound();
  }
```

```
  return <ProductDetail product={product} />;
}
```

Segment-Specific Not Found Pages

```
app/
├── not-found.tsx          # Global 404
└── products/
    ├── not-found.tsx      # Products-specific 404
    └── [id]/
        └── page.tsx       # Can call notFound()
```

Code: `app/products/not-found.tsx`

```
import Link from 'next/link';

export default function ProductNotFound() {
  return (
    <div className="not-found">
      <h2>Product Not Found</h2>
      <p>We couldn't find the product you're looking for.</p>
      <Link href="/products">Browse All Products</Link>
    </div>
  );
}
```

---

9. Parallel Routes

What are Parallel Routes?

Parallel routes allow you to **render multiple pages simultaneously** in the same layout. They're defined using **slots** with the `@` prefix.

Use Cases

1. **Dashboards** with independent sections
2. **Modals** that overlay content
3. **Split views** (e.g., list + detail)
4. **Conditional rendering** based on auth state

Syntax

```
app/
├── layout.tsx             # Receives slots as props
```

```
├── page.tsx               # Default children
├── @analytics/            # Slot: analytics
│   └── page.tsx
└── @team/                 # Slot: team
    └── page.tsx
```

Example 1: Dashboard with Multiple Sections

Folder Structure:

```
app/
└── dashboard/
    ├── layout.tsx
    ├── page.tsx           # Main content
    ├── @stats/
    │   └── page.tsx       # Stats panel
    └── @activity/
        └── page.tsx       # Activity feed
```

Code: `app/dashboard/layout.tsx`

```tsx
interface DashboardLayoutProps {
  children: React.ReactNode;  // page.tsx
  stats: React.ReactNode;     // @stats/page.tsx
  activity: React.ReactNode;  // @activity/page.tsx
}

export default function DashboardLayout({
  children,
  stats,
  activity,
}: DashboardLayoutProps) {
  return (
    <div className="dashboard-grid">
      {/* Main content area */}
      <main className="main-content">
        {children}
      </main>

      {/* Sidebar with parallel routes */}
      <aside className="sidebar">
        <section className="stats-section">
          {stats}
        </section>
        <section className="activity-section">
          {activity}
        </section>
      </aside>
```

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

Code: `app/dashboard/@stats/page.tsx`

```
export default async function StatsPanel() {
  const stats = await fetchStats();

  return (
    <div className="stats-panel">
      <h3>Statistics</h3>
      <div className="stat-card">
        <span>Total Users</span>
        <strong>{stats.users}</strong>
      </div>
      <div className="stat-card">
        <span>Revenue</span>
        <strong>${stats.revenue}</strong>
      </div>
    </div>
  );
}
```

The `default.tsx` File (Critical!)

When using parallel routes, you **must** provide a `default.tsx` for slots that might not have a matching route during navigation.

Why? During soft navigation, Next.js keeps the current slot content. But on hard navigation (refresh), it needs a fallback.

Code: `app/dashboard/@stats/default.tsx`

```
export default function StatsDefault() {
  // Return null to render nothing, or a placeholder
  return null;
}
```

Example 2: Conditional Slots

Parallel routes can render different content based on conditions:

```
app/
└── dashboard/
    ├── layout.tsx
```

```
    ├── @auth/
    │   ├── default.tsx
    │   └── page.tsx
    └── @guest/
        ├── default.tsx
        └── page.tsx
```

Code: `app/dashboard/layout.tsx`

```
import { auth } from '@/lib/auth';

interface DashboardLayoutProps {
  children: React.ReactNode;
  auth: React.ReactNode;
  guest: React.ReactNode;
}

export default async function DashboardLayout({
  children,
  auth: authSlot,
  guest: guestSlot,
}: DashboardLayoutProps) {
  const session = await auth();

  return (
    <div>
      {session ? authSlot : guestSlot}
      {children}
    </div>
  );
}
```

---

10. Intercepting Routes

What are Intercepting Routes?

Intercepting routes let you **load a route within the current layout** while displaying a different URL. This is perfect for **modals**.

The Convention

Pattern	Matches	Use Case
`(.)`	Same level	Modal for sibling route
`(..)`	One level up	Modal for parent's sibling
`(..)(..)`	Two levels up	-

Pattern	Matches	Use Case
`(...)`	From root	Modal for any route

How Interception Works

1. **Soft Navigation** (clicking a Link): Interceptor catches → shows modal
2. **Hard Navigation** (refresh, direct URL): Original route renders → full page

Visual Explanation

```
SOFT NAVIGATION (Link click):
/products → click product → /products/1
           ↓
    Interceptor catches!
           ↓
    Modal shows over /products
    URL: /products/1

HARD NAVIGATION (refresh/direct):
Direct to /products/1
           ↓
    No interception
           ↓
    Full page renders
    URL: /products/1
```

Example: Photo Gallery Modal

Folder Structure:

```
app/
├── layout.tsx
├── @modal/                      # Parallel route slot
│   ├── default.tsx              # Returns null
│   └── (.)photos/[id]/          # Intercepts /photos/[id]
│       └── page.tsx             # Modal content
└── photos/
    ├── page.tsx                 # Photo grid
    └── [id]/
        └── page.tsx             # Full photo page
```

Code: `app/layout.tsx`

```
interface RootLayoutProps {
  children: React.ReactNode;
```

```
  modal: React.ReactNode;
}

export default function RootLayout({ children, modal }: RootLayoutProps) {
  return (
    <html lang="en">
      <body>
        {children}
        {modal}  {/* Modal renders on top */}
      </body>
    </html>
  );
}
```

Code: `app/@modal/default.tsx`

```
export default function ModalDefault() {
  return null; // No modal by default
}
```

Code: `app/@modal/(.)photos/[id]/page.tsx`

```
import Modal from '@/components/Modal';
import { getPhoto } from '@/lib/photos';

interface ModalPhotoProps {
  params: Promise<{ id: string }>;
}

export default async function ModalPhoto({ params }: ModalPhotoProps) {
  const { id } = await params;
  const photo = await getPhoto(id);

  return (
    <Modal>
      <img src={photo.url} alt={photo.title} />
      <h2>{photo.title}</h2>
    </Modal>
  );
}
```

Code: `app/photos/[id]/page.tsx` (Full page version)

```
import { getPhoto } from '@/lib/photos';

interface PhotoPageProps {
```

```
  params: Promise<{ id: string }>;
}

export default async function PhotoPage({ params }: PhotoPageProps) {
  const { id } = await params;
  const photo = await getPhoto(id);

  return (
    <main className="photo-page">
      <img src={photo.url} alt={photo.title} />
      <h1>{photo.title}</h1>
      <p>{photo.description}</p>
    </main>
  );
}
```

Interceptor Pattern Reference

Same level `(.)`:

```
app/
├── @modal/(.)products/[id]/page.tsx  ← Interceptor
└── products/
    ├── page.tsx                       ← Source page
    └── [id]/page.tsx                  ← Target route
```

One level up `(..)`:

```
app/
└── products/
    ├── @modal/(..)products/[id]/page.tsx  ← Interceptor
    ├── page.tsx                            ← Source
    └── [id]/page.tsx                       ← Target
```

---

11. Case Study: Product Quick View Modal

Let's build a complete **product quick view modal** using parallel and intercepting routes.

Requirements

1. `/products` - Grid of product cards
2. `/products/[id]` - Full product page (direct access)
3. Click product card → Modal overlay (intercepted)
4. Refresh while modal open → Full page
5. Loading, error, and 404 states

Final Folder Structure

```
src/
├── app/
│   ├── globals.css
│   ├── layout.tsx              # Root layout with @modal slot
│   ├── page.tsx                # Home page
│   ├── default.tsx             # Root default
│   ├── not-found.tsx           # Global 404
│   ├── @modal/
│   │   ├── default.tsx         # Modal default (null)
│   │   └── (.)products/
│   │       └── [id]/
│   │           └── page.tsx    # Modal content
│   └── products/
│       ├── layout.tsx
│       ├── page.tsx            # Product grid
│       ├── loading.tsx         # Skeleton loading
│       ├── error.tsx           # Error boundary
│       └── [id]/
│           └── page.tsx        # Full product page
├── components/
│   └── Modal.tsx               # Reusable modal
└── lib/
    └── products.ts             # Mock data
```

Step 1: Create Mock Data

File: `src/lib/products.ts`

```
export interface Product {
  id: string;
  name: string;
  description: string;
  price: number;
  category: string;
  imageUrl: string;
  inStock: boolean;
  rating: number;
}

export const products: Product[] = [
  {
    id: "1",
    name: "Wireless Headphones Pro",
    description: "Premium noise-canceling wireless headphones with 40-hour
battery life.",
    price: 299.99,
    category: "Electronics",
```

```
    imageUrl: "https://picsum.photos/seed/headphones/400/300",
    inStock: true,
    rating: 4.8,
  },
  {
    id: "2",
    name: "Mechanical Keyboard RGB",
    description: "Cherry MX switches, per-key RGB lighting, aircraft-grade
aluminum frame.",
    price: 149.99,
    category: "Electronics",
    imageUrl: "https://picsum.photos/seed/keyboard/400/300",
    inStock: true,
    rating: 4.6,
  },
  // ... more products
];

// Simulate API calls with delay
export async function getProducts(): Promise<Product[]> {
  await new Promise((resolve) => setTimeout(resolve, 500));
  return products;
}

export async function getProductById(id: string): Promise<Product | null> {
  await new Promise((resolve) => setTimeout(resolve, 300));
  return products.find((p) => p.id === id) ?? null;
}
```

Step 2: Root Layout with Modal Slot

File: `src/app/layout.tsx`

```
import type { Metadata } from "next";
import Link from "next/link";
import "./globals.css";

export const metadata: Metadata = {
  title: "Product Store",
  description: "Demo of Next.js routing patterns",
};

// Layout receives @modal slot as a prop
interface RootLayoutProps {
  children: React.ReactNode;
  modal: React.ReactNode;  // ← This comes from @modal folder
}

export default function RootLayout({ children, modal }: RootLayoutProps) {
  return (
```

```
    <html lang="en">
      <body>
        {/* Navigation */}
        <nav className="nav">
          <div className="container nav-content">
            <Link href="/" className="nav-brand">
              🛒 Store
            </Link>
            <Link href="/products" className="nav-link">
              Products
            </Link>
          </div>
        </nav>

        {/* Main content */}
        {children}

        {/* Modal slot - renders parallel to children */}
        {modal}
      </body>
    </html>
  );
}
```

Step 3: Modal Default (Critical!)

File: `src/app/@modal/default.tsx`

```
// This file is REQUIRED for parallel routes!
// Without it, you'll get 404 errors on hard navigation.

export default function ModalDefault() {
  // Return null = no modal shown
  return null;
}
```

Step 4: Products Grid Page

File: `src/app/products/page.tsx`

```
import Link from "next/link";
import { getProducts } from "@/lib/products";

export default async function ProductsPage() {
  const products = await getProducts();

  return (
    <main>
```

```
      <header className="page-header">
        <div className="container">
          <h1>Products</h1>
          <p>Click any product for Quick View</p>
        </div>
      </header>

      <div className="container">
        <div className="products-grid">
          {products.map((product) => (
            <Link
              key={product.id}
              href={`/products/${product.id}`}  // ← This will be intercepted!
              className="product-card"
            >
              <img src={product.imageUrl} alt={product.name} />
              <div className="product-info">
                <span className="product-category">{product.category}</span>
                <h2>{product.name}</h2>
                <div className="product-price">${product.price}</div>
              </div>
            </Link>
          ))}
        </div>
      </div>
    </main>
  );
}
```

Step 5: Products Loading State

File: `src/app/products/loading.tsx`

```
export default function ProductsLoading() {
  return (
    <main>
      <header className="page-header">
        <div className="container">
          <h1>Products</h1>
          <p>Loading...</p>
        </div>
      </header>

      <div className="container">
        <div className="products-grid">
          {Array.from({ length: 6 }).map((_, i) => (
            <div key={i} className="product-card skeleton-card">
              <div className="skeleton" style={{ height: 200 }} />
              <div className="product-info">
                <div className="skeleton" style={{ height: 12, width: '30%' }}
```

```
/>
                <div className="skeleton" style={{ height: 20, width: '80%' }}
/>
                <div className="skeleton" style={{ height: 24, width: '40%' }}
/>
              </div>
            </div>
          ))}
        </div>
      </div>
    </main>
  );
}
```

Step 6: Products Error Boundary

File: `src/app/products/error.tsx`

```
'use client';

import { useEffect } from 'react';

interface ProductsErrorProps {
  error: Error & { digest?: string };
  reset: () => void;
}

export default function ProductsError({ error, reset }: ProductsErrorProps) {
  useEffect(() => {
    console.error('Products error:', error);
  }, [error]);

  return (
    <div className="error-container">
      <div className="error-icon">⚠</div>
      <h2>Failed to load products</h2>
      <p>{error.message}</p>
      <button onClick={reset} className="btn btn-primary">
        Try Again
      </button>
    </div>
  );
}
```

Step 7: Full Product Page

File: `src/app/products/[id]/page.tsx`

```
import Link from "next/link";
import { notFound } from "next/navigation";
import { getProductById, products } from "@/lib/products";

interface ProductPageProps {
  params: Promise<{ id: string }>;
}

export default async function ProductPage({ params }: ProductPageProps) {
  const { id } = await params;
  const product = await getProductById(id);

  if (!product) {
    notFound();  // Triggers not-found.tsx
  }

  return (
    <main className="container">
      <Link href="/products" className="back-link">
        ← Back to Products
      </Link>

      <div className="product-detail">
        <img src={product.imageUrl} alt={product.name} />

        <div className="product-detail-info">
          <span className="product-category">{product.category}</span>
          <h1>{product.name}</h1>
          <div className="product-price">${product.price}</div>
          <p>{product.description}</p>

          <button className="btn btn-primary">
            Add to Cart
          </button>

          {/* Info box explaining routing */}
          <div className="info-box">
            💡 You're seeing the <strong>full page</strong> because you
            navigated directly or refreshed. Go to{' '}
            <Link href="/products">/products</Link> and click a card
            to see the modal.
          </div>
        </div>
      </div>
    </main>
  );
}

// Pre-render all product pages at build time
export function generateStaticParams() {
  return products.map((product) => ({
    id: product.id,
```

```
  }));
}
```

Step 8: Modal Component (Client Component)

File: `src/components/Modal.tsx`

```
'use client';

import { useRouter } from 'next/navigation';
import { useCallback, useEffect } from 'react';

interface ModalProps {
  children: React.ReactNode;
  title?: string;
}

export default function Modal({ children, title }: ModalProps) {
  const router = useRouter();

  // Close modal by going back in history
  const handleClose = useCallback(() => {
    router.back();
  }, [router]);

  // Close on Escape key
  useEffect(() => {
    const handleKeyDown = (e: KeyboardEvent) => {
      if (e.key === 'Escape') {
        handleClose();
      }
    };

    document.addEventListener('keydown', handleKeyDown);
    return () => document.removeEventListener('keydown', handleKeyDown);
  }, [handleClose]);

  // Close when clicking overlay
  const handleOverlayClick = (e: React.MouseEvent) => {
    if (e.target === e.currentTarget) {
      handleClose();
    }
  };

  return (
    <div className="modal-overlay" onClick={handleOverlayClick}>
      <div className="modal-content" role="dialog" aria-modal="true">
        <div className="modal-header">
          <h2>{title || 'Quick View'}</h2>
          <button onClick={handleClose} className="modal-close">
```

```
            ×
          </button>
        </div>
        <div className="modal-body">
          {children}
        </div>
      </div>
    </div>
  );
}
```

Step 9: Intercepting Route (The Magic!)

File: `src/app/@modal/(.)products/[id]/page.tsx`

```
// INTERCEPTING ROUTE
//
// (.) = intercepts routes at the SAME level
// This file intercepts /products/[id] when navigating from /products
//
// How it works:
// 1. User is on /products
// 2. User clicks Link to /products/1
// 3. Next.js sees this interceptor exists
// 4. Instead of products/[id]/page.tsx, THIS renders
// 5. URL changes to /products/1
// 6. But /products page stays visible behind modal!

import Link from "next/link";
import { notFound } from "next/navigation";
import Modal from "@/components/Modal";
import { getProductById } from "@/lib/products";

interface ModalProductPageProps {
  params: Promise<{ id: string }>;
}

export default async function ModalProductPage({ params }:
ModalProductPageProps) {
  const { id } = await params;
  const product = await getProductById(id);

  if (!product) {
    notFound();
  }

  return (
    <Modal title="Quick View">
      <img src={product.imageUrl} alt={product.name} className="modal-image" />
```

```
      <span className="product-category">{product.category}</span>
      <h3>{product.name}</h3>
      <div className="product-price">${product.price}</div>
      <p>{product.description}</p>

      <div className="modal-actions">
        <button className="btn btn-primary">
          Add to Cart
        </button>
        <Link href={`/products/${product.id}`} className="btn btn-outline">
          View Full Page →
        </Link>
      </div>

      <div className="info-box">
        💡 This is an <strong>intercepted route</strong>.
        URL is <code>/products/{id}</code> but you see a modal.
        Refresh to see the full page!
      </div>
    </Modal>
  );
}
```

Step 10: Test the Implementation

1. **Start the dev server:** `npm run dev`
2. **Go to** `http://localhost:3000/products`
3. **Click any product card** → Modal opens, URL changes to `/products/1`
4. **Press Escape** or click outside → Modal closes, back to `/products`
5. **Refresh while modal is open** → Full page renders
6. **Navigate directly to** `/products/1` → Full page renders

How It All Works Together

```
User clicks product card on /products
         ↓
Link navigates to /products/1 (soft navigation)
         ↓
Next.js checks for interceptors
         ↓
Found: @modal/(.)products/[id]/page.tsx
         ↓
Renders interceptor in @modal slot
         ↓
Original /products page stays in {children}
         ↓
RESULT: Modal overlay + products grid visible
         URL: /products/1
```

```
---

User refreshes page (or navigates directly to /products/1)
         ↓
Hard navigation - no interception
         ↓
products/[id]/page.tsx renders
         ↓
RESULT: Full product page
         URL: /products/1
```

---

12. Navigation and Linking

The `<Link>` Component

Next.js `<Link>` enables **client-side navigation** with prefetching.

```
import Link from 'next/link';

export default function Navigation() {
  return (
    <nav>
      {/* Basic link */}
      <Link href="/about">About</Link>

      {/* Dynamic route */}
      <Link href={`/products/${product.id}`}>
        {product.name}
      </Link>

      {/* With query params */}
      <Link href="/search?q=shoes">
        Search Shoes
      </Link>

      {/* Replace history (no back) */}
      <Link href="/login" replace>
        Login
      </Link>

      {/* Disable prefetch */}
      <Link href="/heavy-page" prefetch={false}>
        Heavy Page
      </Link>

      {/* Scroll to top disabled */}
      <Link href="/same-page#section" scroll={false}>
        Jump to Section
      </Link>
```

```
    </nav>
  );
}
```

The `useRouter` Hook

For programmatic navigation in Client Components:

```
'use client';

import { useRouter } from 'next/navigation';

export default function LoginForm() {
  const router = useRouter();

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    const success = await login(/* ... */);

    if (success) {
      // Navigate to dashboard
      router.push('/dashboard');

      // Or replace current history entry
      router.replace('/dashboard');

      // Or go back
      router.back();

      // Or refresh current route
      router.refresh();
    }
  };

  return <form onSubmit={handleSubmit}>...</form>;
}
```

The `redirect` Function

For server-side redirects:

```
import { redirect } from 'next/navigation';

export default async function ProtectedPage() {
  const session = await getSession();

  if (!session) {
```

```
    redirect('/login');  // Server-side redirect
  }

  return <Dashboard />;
}
```

Active Links

Detect active route with `usePathname`:

```
'use client';

import Link from 'next/link';
import { usePathname } from 'next/navigation';

export default function NavLink({
  href,
  children
}: {
  href: string;
  children: React.ReactNode
}) {
  const pathname = usePathname();
  const isActive = pathname === href;

  return (
    <Link
      href={href}
      className={isActive ? 'nav-link active' : 'nav-link'}
    >
      {children}
    </Link>
  );
}
```

---

13. Route Handlers (API Routes)

Creating API Endpoints

Route handlers are created using `route.ts` files.

File: `app/api/products/route.ts`

```
import { NextResponse } from 'next/server';
import { products } from '@/lib/products';

// GET /api/products
```

```
export async function GET() {
  return NextResponse.json(products);
}

// POST /api/products
export async function POST(request: Request) {
  const body = await request.json();

  const newProduct = {
    id: String(products.length + 1),
    ...body,
  };

  products.push(newProduct);

  return NextResponse.json(newProduct, { status: 201 });
}
```

Dynamic Route Handlers

File: `app/api/products/[id]/route.ts`

```
import { NextResponse } from 'next/server';
import { products } from '@/lib/products';

interface RouteParams {
  params: Promise<{ id: string }>;
}

// GET /api/products/[id]
export async function GET(request: Request, { params }: RouteParams) {
  const { id } = await params;
  const product = products.find(p => p.id === id);

  if (!product) {
    return NextResponse.json(
      { error: 'Product not found' },
      { status: 404 }
    );
  }

  return NextResponse.json(product);
}

// DELETE /api/products/[id]
export async function DELETE(request: Request, { params }: RouteParams) {
  const { id } = await params;
  const index = products.findIndex(p => p.id === id);

  if (index === -1) {
```

```
    return NextResponse.json(
      { error: 'Product not found' },
      { status: 404 }
    );
  }

  products.splice(index, 1);

  return NextResponse.json({ success: true });
}
```

Supported HTTP Methods

```
export async function GET(request: Request) {}
export async function POST(request: Request) {}
export async function PUT(request: Request) {}
export async function PATCH(request: Request) {}
export async function DELETE(request: Request) {}
export async function HEAD(request: Request) {}
export async function OPTIONS(request: Request) {}
```

---

14. Middleware

What is Middleware?

Middleware runs **before** a request is completed. Use it for:

- Authentication
- Redirects
- Rewriting URLs
- Adding headers

Creating Middleware

File: `middleware.ts` **(root level)**

```
import { NextResponse } from 'next/server';
import type { NextRequest } from 'next/server';

export function middleware(request: NextRequest) {
  // Get the pathname
  const { pathname } = request.nextUrl;

  // Check for auth token
  const token = request.cookies.get('token');
```

```
  // Protect /dashboard routes
  if (pathname.startsWith('/dashboard') && !token) {
    return NextResponse.redirect(new URL('/login', request.url));
  }

  // Add custom header
  const response = NextResponse.next();
  response.headers.set('x-custom-header', 'my-value');

  return response;
}

// Configure which paths middleware runs on
export const config = {
  matcher: [
    // Match all paths except static files
    '/((?!_next/static|_next/image|favicon.ico).*)',
  ],
};
```

Middleware Patterns

Pattern 1: Authentication

```
export function middleware(request: NextRequest) {
  const token = request.cookies.get('session');

  const protectedPaths = ['/dashboard', '/settings', '/profile'];
  const isProtected = protectedPaths.some(path =>
    request.nextUrl.pathname.startsWith(path)
  );

  if (isProtected && !token) {
    const loginUrl = new URL('/login', request.url);
    loginUrl.searchParams.set('from', request.nextUrl.pathname);
    return NextResponse.redirect(loginUrl);
  }
}
```

Pattern 2: Geolocation Redirect

```
export function middleware(request: NextRequest) {
  const country = request.geo?.country || 'US';

  if (country === 'UK' && !request.nextUrl.pathname.startsWith('/uk')) {
    return NextResponse.redirect(new URL('/uk', request.url));
  }
}
```

15. Best Practices

1. Organize by Feature

```
app/
├── (marketing)/
│   ├── about/
│   ├── blog/
│   └── pricing/
├── (app)/
│   ├── dashboard/
│   ├── settings/
│   └── profile/
└── (auth)/
    ├── login/
    └── register/
```

2. Colocate Related Files

```
app/
└── products/
    ├── page.tsx
    ├── loading.tsx
    ├── error.tsx
    ├── actions.ts      # Server actions
    ├── utils.ts        # Helper functions
    └── components/     # Route-specific components
        ├── ProductCard.tsx
        └── ProductFilter.tsx
```

3. Use Loading and Error States

Always provide feedback:

- `loading.tsx` for data fetching
- `error.tsx` for graceful error handling
- `not-found.tsx` for 404 cases

4. Leverage Server Components

Keep components as Server Components unless they need:

- Event handlers (`onClick`, `onChange`)
- Hooks (`useState`, `useEffect`)
- Browser APIs

5. Prefetch Important Routes

```
// Prefetch on hover
<Link href="/products" prefetch={true}>Products</Link>

// Programmatic prefetch
const router = useRouter();
router.prefetch('/dashboard');
```

6. Handle Loading States Granularly

```
<Suspense fallback={<HeaderSkeleton />}>
  <Header />
</Suspense>

<Suspense fallback={<SidebarSkeleton />}>
  <Sidebar />
</Suspense>

<Suspense fallback={<ContentSkeleton />}>
  <MainContent />
</Suspense>
```

7. Use Route Groups for Layouts

Avoid layout pollution by grouping routes:

```
app/
├── (with-sidebar)/
│   ├── layout.tsx      # Has sidebar
│   ├── dashboard/
│   └── settings/
└── (without-sidebar)/
    ├── layout.tsx      # No sidebar
    ├── login/
    └── register/
```

Quick Reference

File Conventions

File	Purpose
`page.tsx`	Unique UI for a route

File	Purpose
`layout.tsx`	Shared UI wrapper
`loading.tsx`	Loading UI
`error.tsx`	Error UI
`not-found.tsx`	404 UI
`template.tsx`	Re-rendered layout
`default.tsx`	Parallel route fallback
`route.ts`	API endpoint

Dynamic Route Syntax

Pattern	Example	Match
`[param]`	`[id]`	`/1`
`[...param]`	`[...slug]`	`/a/b/c`
`[[...param]]`	`[[...slug]]`	`/`, `/a/b`

Intercepting Route Syntax

Pattern	Match Level
`(.)`	Same level
`(..)`	One level up
`(..)(..)`	Two levels up
`(...)`	Root level

Navigation Methods

```
// Client Component
import { useRouter } from 'next/navigation';
const router = useRouter();
router.push('/path');
router.replace('/path');
router.back();
router.refresh();

// Server Component
import { redirect } from 'next/navigation';
redirect('/path');

// Component
```

```
import Link from 'next/link';
<Link href="/path">Link</Link>
```

Conclusion

Next.js App Router provides a powerful, file-system based routing solution with:

- **Intuitive file conventions** for pages, layouts, and special states
- **Flexible dynamic routing** for data-driven pages
- **Advanced patterns** like parallel and intercepting routes
- **Built-in optimizations** for loading and error states

The modal pattern demonstrated in this tutorial showcases how these features work together to create sophisticated, production-ready user experiences.

Happy routing! 🚀