

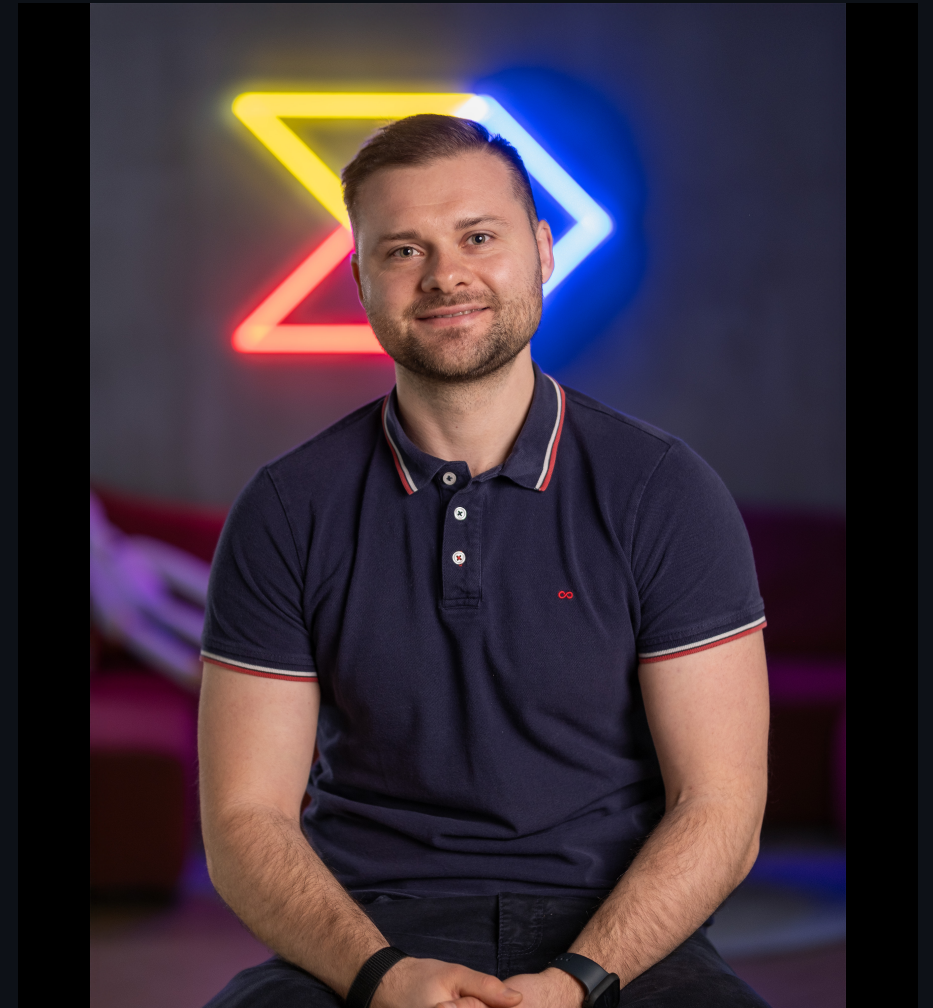
# Next.js Training

## Modern React Web Development

Martin Krištof

# About Me

- **Productboard** (since March 2025)
  - Product Staff Engineer
  - Tech Lead Nucleus Guild, member of FE guild
- **React Experience**
  - React Lover (10+ years)
  - Consultant
  - Courses & Workshops (React, Next.js, QA)
  - Video courses for Skillmea



# Agenda

## 1. Pages Router

### 1.1. File-Based Routing

### 1.2. Rendering

### 1.3. Data fetching

## 2. App Router

### 2.1. File-Based Routing

### 2.2. Rendering

### 2.3 Data fetching

## 3. Middleware

4. Environments

5. Configuration & Instrumentation

6. Extra: Styling, Forms, Error Boundaries, MDX, Images, Testing

# Pages Router

## Pages Router (Legacy)

- Demonstrated in the separate project: **next-guide-pages** ( `apps/next-guide-pages` )
- File-based routing in the `pages/` directory
- Each file in `pages/` is a route (e.g., `index.tsx`, `users/[id].tsx` )
- Dynamic routes: `[id].tsx`, catch-all: `[...slug].tsx`
- API routes: `pages/api/`

- Special files for advanced customization:
  - **\_app.tsx**: Custom root component for all pages ([see file](#)). Use for global styles, context providers, etc.
  - **\_document.tsx**: Customizes the HTML document structure ([see file](#)). Use for meta tags, lang, etc.
  - **\_error.tsx**: Custom error page for runtime errors ([see file](#)).
  - **404.tsx**: Custom 404 Not Found page ([see file](#)).
  - **500.tsx**: Custom 500 Internal Server Error page ([see file](#)).

## Demos:

- Homepage (/)
- User detail (dynamic route) (/users/1)
- API users route (/api/users)
- Catch-all route (/ssg)



## API Routes

- Serverless functions as API endpoints in `pages/api/`
- Each file in `api/` is an endpoint (GET, POST, etc.)
- Use the built-in types: `NextApiRequest` and `NextApiResponse` from `next`
- Response helpers: `res.status`, `res.json`, `res.send`, `res.redirect`, `res.revalidate`
- Supports dynamic routes ( `pages/api/post/[pid].ts` ) and catch-all routes ( `pages/api/post/[...slug].ts` )
- TypeScript support for type-safe APIs
- [Official documentation](#)

## Example: Basic API Route

```
import type { NextApiRequest, NextApiResponse } from 'next';

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  res.status(200).json({ message: 'Hello from Next.js!' });
}
```

## Example: Dynamic API Route

```
// pages/api/post/[pid].ts
import type { NextApiRequest, NextApiResponse } from 'next';

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  const { pid } = req.query;
  res.end(`Post: ${pid}`);
}
```

## Example: Catch-all API Route

```
// pages/api/post/[...slug].ts
import type { NextApiRequest, NextApiResponse } from 'next';

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  const { slug } = req.query;
  res.end(`Post: ${Array.isArray(slug) ? slug.join(', ') : slug}`);
}
```

# Linking and Navigating

- Next.js provides a built-in `<Link>` component for client-side navigation between routes.
- Using `<Link>` enables fast, seamless transitions without full page reloads, preserving state and improving UX.
- `<Link>` automatically prefetches linked pages in the background for faster navigation (when visible in the viewport).
- Prefer `<Link>` over a plain `<a>` tag for internal navigation. Use `<a>` only for external links.
- You can disable prefetching with the `prefetch={false}` prop.
- `<Link>` works with dynamic routes, catch-all routes, and route groups.

## Example:

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

export default function Navigation() {
  return (
    <nav>
      <Link href="/about">About</Link>
      <Link href="/blog" prefetch={false}>
        Blog (no prefetch)
      </Link>
      <a href="https://nextjs.org" target="_blank" rel="noopener noreferrer">
        Next.js Docs
      </a>
    </nav>
  );
}
```

- For advanced use cases, you can use the `useRouter`, `usePathname`, and `useSearchParams` hooks from `next/navigation`.
- [Official documentation: Linking and Navigating](#)

# Rendering & Data Fetching



- The Pages Router supports multiple rendering and data fetching strategies:
  - **SSR (Server-Side Rendering)**: Use `getServerSideProps` to fetch data on every request.
    - [SSR example \(/ssr\)](#)
  - **SSG (Static Site Generation)**: Use `getStaticProps` (and optionally `getStaticPaths`) to pre-render pages at build time.
    - [SSG example \(/ssg\)](#)

## getStaticPaths fallback options:

- `fallback: false` – Only the paths returned by `getStaticPaths` are generated at build time. Any other route will show a 404 page.
- `fallback: true` – New paths not returned by `getStaticPaths` will be rendered on-demand on the first request, then cached for future requests. The page will show a loading state until the content is generated.
- `fallback: 'blocking'` – New paths are rendered on-demand like `true`, but the user will not see a loading state; the server waits until the page is generated and then serves the full page.

Use `false` for small/finite sets of pages, `true` or `'blocking'` for large or dynamic sets where not all paths are known at build time.

- [getStaticPaths details](#)

- **Client-side Fetching:** Use React hooks like `useEffect` to fetch data on the client after the page loads.
  - [CSR example \(/csr\)](#)
- You can combine these strategies as needed for your use case.
- See also: [Next.js Data Fetching Docs](#)

# App Router

# App Router (Modern)

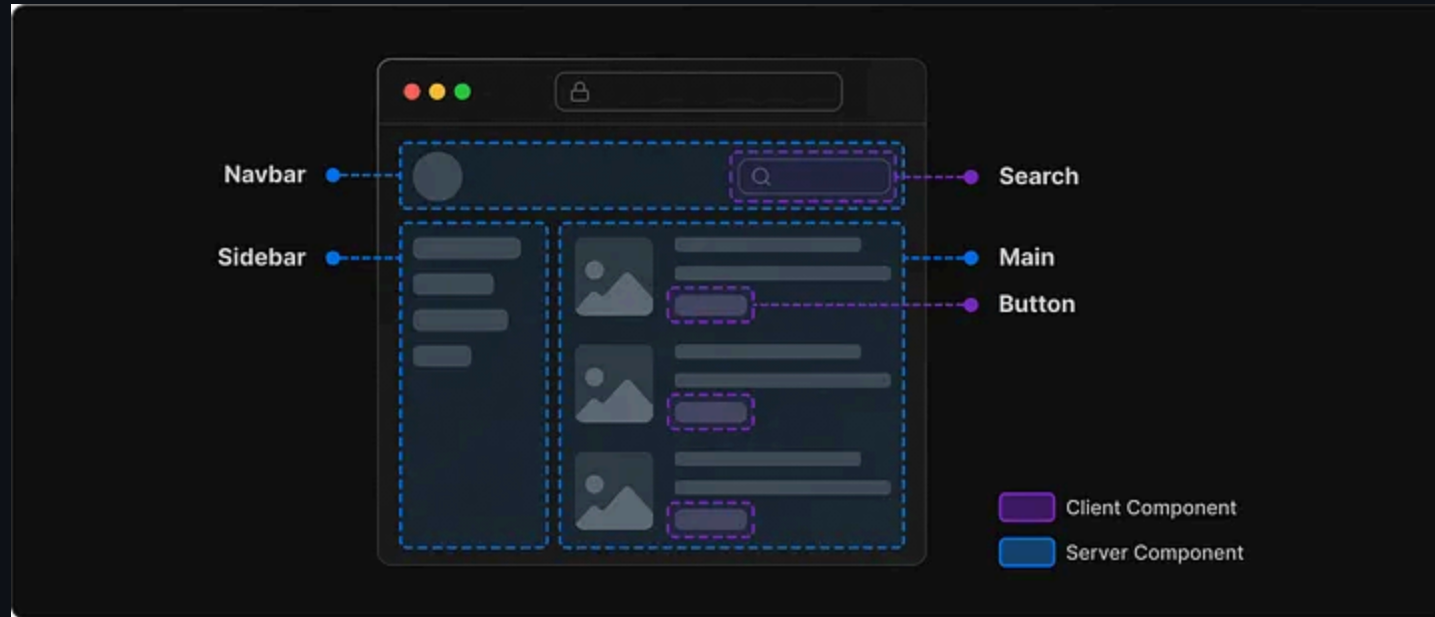
- File-based routing in `src/app/`
- Each folder with `page.tsx` = a route
- Supports layouts, nested routes, dynamic routes, catch-all routes
- **Layout:**
  - [Root layout.tsx](#) – defines the main structure, shared UI, and providers for the whole app.
- **Loading UI:**
  - You can add `loading.tsx` to any route folder for custom loading skeletons.
  - [API loading](#)
  - [Dashboard loading](#)
  - [Blog post loading](#)
- [Official Project Structure documentation](#)

# Rendering & Data Fetching

# Server vs Client Components

Server Components are rendered on the server and sent as HTML to the client, while Client Components are rendered in the browser. In Next.js App Router, **Server Components are the default**—you only need to use Client Components when you need interactivity, browser APIs, or React hooks like `useState`, `useEffect`, etc.

- Server Components improve performance by reducing the amount of JavaScript sent to the client.
- Client Components are needed for interactivity (event handlers, state, browser APIs).
- You can mix Server and Client Components on the same page.
- Mark a component as client by adding `"use client"` at the top of the file.





## Demos:

- [Server component example \(/server-component\)](/server-component)
- [Blog post: Server vs Client Components\(/blog/server-components-vs-client-components\)](/blog/server-components-vs-client-components)

What do you need to do?	Server Component	Client Component
Fetch data. <a href="#">Learn more.</a>	✓	⚠
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server / Reduce client-side JavaScript	✓	✗
Add interactivity and event listeners ( <code>onClick()</code> , <code>onChange()</code> , etc)	✗	✓
Use State and Lifecycle Effects ( <code>useState()</code> , <code>useReducer()</code> , <code>useEffect()</code> , etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use <a href="#">React Class components</a>	✗	✓

## Demos:

- [Client component demo \(/client-component\)](/client-component)
- with [Counter component](#)

For more, see [Server Components vs. Client Components](#)  
Nice explanation [Ariel Shulman - WebExpo](#)

## SSR & SSG

- SSR: Server Side Rendering (on request)
- SSG: Static Site Generation (at build time)
- Use `generateStaticParams` for SSG

### Demos:

- [Dashboard \(SSR\) \(/dashboard\)](/dashboard) - must be logged
- [Blog \(SSG\) \(/blog\)](/blog)

## The `fetch` function

- Native fetch in server components
- Automatic caching

### Demos:

- [Client data fetching \(/client-data-fetching\)](/client-data-fetching)

# Caching

- By default, fetch requests are **not** cached in Next.js 15+ ([docs](#))
- Enable caching explicitly with `cache: 'force-cache'` or `next.revalidate` for ISR:

```
// Static caching
fetch(url, { cache: 'force-cache' });

// ISR (Incremental Static Regeneration)
fetch(url, { next: { revalidate: 3600 } });
```

- Use `cache: 'no-store'` to always fetch fresh data (SSR)

## Controlling static/dynamic rendering with `dynamic` export:

- In App Router, you can control how a route is rendered and cached using the `dynamic` export at the top of your file:

```
// page.tsx, layout.tsx, or route.ts
export const dynamic = 'auto'; // (default) Next.js decides based on usage
// or
export const dynamic = 'force-static'; // always statically render and cache
// or
export const dynamic = 'force-dynamic'; // always render on the server, no cache
```

- **'auto'** (default): Next.js chooses static or dynamic based on your code (e.g. use of `fetch`, cookies, headers).
- **'force-static'**: Forces static rendering and caching, even if you use dynamic code (errors if truly dynamic).
- **'force-dynamic'**: Forces server-side rendering on every request, disables all caching.



Use these options to fine-tune performance and cache behavior for each route.

[Dashboard page example \(/dashboard\)](/dashboard) - see build report

[Docs: Static and Dynamic Rendering](#)

### Demos:

- [Caching demo\(/caching-demo/\)](/caching-demo/)

## Note on combining `dynamic` and `revalidate`:

- `revalidate` only has an effect when the route is statically rendered (default or `force-static`).
- If you set `dynamic = 'force-dynamic'`, any `revalidate` value is ignored—every request is always rendered on the server, no cache.
- Example:

```
export const dynamic = 'force-dynamic';  
export const revalidate = 60; // This will be ignored
```

- Use `revalidate` for Incremental Static Regeneration (ISR) with static routes, not with `force-dynamic`.

## Server Actions

- **Server Actions** let you run server-side code directly from your React components—no need to create a separate API route.
- Useful for mutations (create, update, delete), form submissions, and cache invalidation.
- Secure: code runs only on the server, never sent to the client.
- Can be called from forms or programmatically.
- Great for progressive enhancement (forms work even without JS).

## Minimal example:

```
// In a server component
'use server';

export async function createUser(formData) {
  // Save to DB, revalidate cache, etc.
}

// In your page/component
<form action={createUser}>
  <input name="name" />
  <button type="submit">Create</button>
</form>;
```

- **Examples:**
  - [User form server action](#) - can be combined with [useActionState](#)
  - [Database demo server actions](#)
- You can also call `revalidateTag` or `revalidatePath` inside a server action after a mutation.
- [Docs: Server Actions](#)

## On-demand Revalidation: `revalidateTag` and `revalidatePath`

- `revalidateTag(tag)` lets you purge the cache for a specific tag on demand. Use it after a mutation (e.g., creating or updating a record) to ensure that data with this tag is refetched on the next request.

- Add tags when fetching data:

```
fetch(url, { next: { tags: ['users'] } });
```

- After a mutation, call on the server:

```
import { revalidateTag } from 'next/cache';  
revalidateTag('users');
```

- 
- `revalidatePath(path)` purges the cache for a specific path (page or API endpoint). Use it when you want to revalidate a particular page after a data change.

- See examples in the project:
  - [User Form Server Action \(revalidateTag\) \(/user-form-server-action\)](#)
  - [Database Demo \(revalidatePath\) \(/database-demo\)](#)
- More in the docs: [revalidateTag](#), [revalidatePath](#)

# Connecting to Database and Filesystem

- Use Prisma for DB, Node APIs for filesystem

## Demos:

- [Database demo \(/database-demo\)](#)
- [Prisma schema](#)

**Tip:** To explore and edit your database visually, you can use Prisma Studio:

```
yarn prisma studio
```



# API Routes

- In the App Router, API routes are implemented as **Route Handlers** using `route.ts` (or `route.js`) files inside the `app` directory.
- Each route handler can export HTTP methods as functions: `GET`, `POST`, `PUT`, `DELETE`, etc.
- You can use either the default Node.js runtime or opt-in to the Edge runtime by exporting `export const runtime = 'edge'`.
- Use the `NextRequest` object to access request data (body, query, headers, cookies).
- Use the `NextResponse` object to send responses.

- Route handlers are colocated with your routes, and support dynamic segments, catch-all, and route groups.
- **Difference from Pages Router:** No need for `api/` prefix in the URL, and you have full control over HTTP methods and runtime.
- [Official documentation: Route Handlers](#)

## Example: Basic GET and POST handler

```
// app/api/hello/route.ts
import { NextRequest, NextResponse } from 'next/server';

export async function GET(request: NextRequest) {
  return NextResponse.json({ message: 'Hello from App Router!' });
}

export async function POST(request: NextRequest) {
  const data = await request.json();
  // process data...
  return NextResponse.json({ received: data });
}
```

- For dynamic API routes, use `[param]` or `[...slug]` in the folder name, just like for pages.
- You can also use middleware and Edge runtime for advanced use cases.

# Route Groups & Segmented Sections

- In the App Router, you can separate sections using parentheses folders, e.g., `(marketing)`.
- Allows you to separate, for example, public and internal parts of the site, or marketing pages.
- **Example:**
  - [Marketing group \(/about\)](#)
- [Official documentation](#)

# Parallel Routes & Slots

- Parallel routes allow you to render multiple independent parts of the page (slots) at the same time.
- Each slot is a folder starting with `@` (e.g., `@feed`, `@notifications`).
- Slots can also be nested.
- **Examples:**
  - [Main parallel-demo \(/parallel-demo\)](/parallel-demo)
  - [Feed slot \(/parallel-demo\)](/parallel-demo)
  - [Notifications slot \(/parallel-demo\)](/parallel-demo)
  - [Nested slot feed/archive \(/parallel-demo/archive\)](/parallel-demo/archive)
- [Official documentation](#)

# Conditional Routes & Slots

- Conditional routes allow you to dynamically change content based on a segment (e.g., user role).
- Slots like `@admin` , `@user` within a dynamic folder `[role]` .
- **Examples:**
  - [Conditional routes demo \(/conditional-routes-demo/user\)](/conditional-routes-demo/user)
  - [Admin slot \(/conditional-routes-demo/admin\)](/conditional-routes-demo/admin)
  - [User slot \(/conditional-routes-demo/user\)](/conditional-routes-demo/user)
- [Official documentation](#)

# Progressive Enhancement

- Progressive enhancement means the form works even without JavaScript – validation and processing happen on the server.
- In Next.js, you can combine this with Server Actions.
- Benefits: better accessibility, SEO, fallback for older browsers.
- **Examples:**
  - [Progressive enhancement form \(/progressive-enhancement-form\)](/progressive-enhancement-form)
  - [Server action](#)
- [Official documentation](#)

# Error Boundaries & Error Handling

- Next.js App Router has special files for error boundaries:
  - `error.tsx` – error boundary for a specific route
  - `global-error.tsx` – global error boundary for the whole app ([global-error.tsx](#))
  - `not-found.tsx` – page for 404 errors ([not-found.tsx](#))
- **Examples:**
  - [error.tsx](#)
  - [global-error.tsx](#)
  - [not-found.tsx/not-found](#))
- [Official documentation](#)



## 4. Middleware

- Code that runs before a request is completed
- `src/app/middleware.ts`
- Use for auth, redirects, logging

### Demos:

- [Middleware file](#)
- [Middleware demo page \(/middleware-demo\)](#)

# 5. Environments

## Environment Variables

- Next.js supports environment variables via `.env` files in the project root.
- Supported files (loaded in this order):
  - i. `.env.${(NODE_ENV)}.local`
  - ii. `.env.local` (not loaded in test)
  - iii. `.env.${(NODE_ENV)}`
  - iv. `.env`

- Variables prefixed with `NEXT_PUBLIC_` are exposed to the browser (client-side). Others are only available on the server.
- Example `.env` :

```
DATABASE_URL=postgres://user:pass@localhost:5432/db  
NEXT_PUBLIC_API_URL=https://api.example.com
```

- Usage in code:

```
// Server only
const dbUrl = process.env.DATABASE_URL;
// Client or server
const apiUrl = process.env.NEXT_PUBLIC_API_URL;
```

- Variables are inlined at build time for the client. For runtime values, use server-side code or API endpoints.
- You can reference other variables in `.env` using `$VAR_NAME`.
- For advanced use (e.g. loading env in scripts), use `@next/env`:

```
import { loadEnvConfig } from '@next/env';
loadEnvConfig(process.cwd());
```

[Official docs: Environment Variables](#)

# Passing Data Between Environments

When deploying Next.js in Docker or other environments, you often need to pass environment variables securely and correctly.

- **.env files:** Place `.env` , `.env.production` , etc. in the project root. These are loaded automatically at build/start time.
- **Server-side variables** (e.g. `DATABASE_URL` ) are read at runtime and can be passed when starting the container:

```
docker run -e DATABASE_URL=postgres://user:pass@host/db my-next-app  
# or  
docker run --env-file .env my-next-app
```

- **Client-side variables** (must start with `NEXT_PUBLIC_`) are inlined at build time. They must be set before running `next build`:

```
# Example Dockerfile snippet
ENV NEXT_PUBLIC_API_URL=https://api.example.com
RUN yarn build
```

- **Build-time vs. Run-time:**
  - Server envs can be changed at container start.
  - Client envs are "baked in" at build time—changing them later requires a rebuild.
- **Best practice:**
  - Use server envs for secrets and runtime config.
  - Use client envs only for values that can be public and are known at build time.

## Example: Dockerfile for Next.js

```
# Install dependencies only when needed
FROM node:18-alpine AS deps
WORKDIR /app
COPY package.json yarn.lock ./
RUN yarn install --frozen-lockfile

# Build the app
FROM node:18-alpine AS builder
WORKDIR /app
COPY . .
COPY --from=deps /app/node_modules ./node_modules
# Set build-time envs (client-side)
ENV NEXT_PUBLIC_API_URL=https://api.example.com
RUN yarn build

# Production image
FROM node:18-alpine AS runner
WORKDIR /app
ENV NODE_ENV=production
COPY --from=builder /app/.next ./next
COPY --from=builder /app/public ./public
COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/node_modules ./node_modules

# Set runtime envs (server-side)
ENV DATABASE_URL=postgres://user:pass@host/db

EXPOSE 3000
CMD ["yarn", "start"]
```



- Set `NEXT_PUBLIC_` variables before `yarn build` (build-time, client-side)
- Set server-side variables (like `DATABASE_URL` ) at runtime with `docker run -e ...`

### Official docs:

- [Environment Variables](#)
- [Docker deployment](#)

## Node.js vs. Edge

- Next.js can run on Node.js or Edge runtime
- Use `export const runtime = 'edge'` in a route/middleware

### Demo:

- [Runtime demo \(/runtime-demo\)](/runtime-demo)

# Different Types of API Routes (Edge/Node, request info)

- Next.js allows you to write API routes for different runtimes:
  - **Edge runtime:** faster, limited API (e.g., no fs)
  - **Node runtime:** full access to Node.js API
- You can also get request info (headers, cookies, etc.)
- **Examples:**
  - [API route Edge](#)
  - [API route Node](#)
  - [API request info](#)
- [Official documentation](#)

## 6. Configuration & Instrumentation

# Next Config

Next.js allows you to configure various aspects of your application using the `next.config.js` or `next.config.ts` file.

- Mode settings (strict mode, experimental features)
- Image, headers, file types, build options
- **assetPrefix**: Set a custom prefix for serving static assets (e.g. from a CDN).
- **basePath**: Serve your app from a subpath (e.g. `/docs` or `/app` ).
- **Redirects and Rewrites**

## Example:

```
// next.config.ts
const nextConfig: NextConfig = {
  assetPrefix: 'https://cdn.example.com', // serve static assets from CDN
  basePath: '/docs', // app will be served from https://yourdomain.com/docs
};
```

Use `assetPrefix` when deploying static assets to a CDN. Use `basePath` when your app is not at the root of the domain.

## Redirects and Rewrites

- **Redirects** let you send users from one URL to another (e.g., when migrating content or changing site structure).
- **Rewrites** let you map an incoming request path to a different destination path on the server, without changing the URL in the browser (e.g., for API proxying or pretty URLs).

## Example from the project:

```
// apps/next-guide-app/next.config.ts
// Redirects
async redirects() {
  return [
    {
      source: '/old-blog',
      destination: '/blog',
      permanent: true,
    },
  ];
},
// Rewrites
async rewrites() {
  return [
    {
      source: '/api/legacy/:path*',
      destination: '/api/:path*',
    },
  ];
},
```



# Instrumentation

- Instrumentation allows you to monitor performance, log, or connect OpenTelemetry.
- In Next.js, add an `instrumentation.js` or `instrumentation.ts` file to `app/`.
- Runs only on the server at process startup.
- **Examples:**
  - [instrumentation.js](#)
  - [Instrumentation demo page](#)
- [Official documentation](#)
- **Demos:**
  - [Instrumentation demo \(/instrumentation-demo\)](#)
  - [instrumentation.js example](#)

## 7. Extra

- **Styling:** See [globals.css](#)
- **MDX:**
  - [MDX demo](#)
  - [MDX layout \(/mdx-demo\)](#)
  - [MDX components \(/mdx-demo\)](#)

- Image Component:

- [Image demo \(/image-demo\)](/image-demo)

**Note:** If you want to load images from external domains, you must add those domains to `remotePatterns` in your `next.config.ts`:

```
// next.config.ts
images: {
  remotePatterns: [
    {
      protocol: 'https',
      hostname: 'images.unsplash.com',
    },
  ],
},
```

Otherwise, images from those domains will not load and Next.js will show an error.

- **Shared Components:**

- Counter component
- Use a `/components` or `/_components` (private - non-routable) folder for reusable UI (best practice)

- **Public Assets:**

- Use the `/public` folder for static assets (images, favicon, etc.) ([public/](#))

# What was not covered

Some advanced or less common Next.js topics are not covered in detail in this presentation:

- Internationalization (i18n) – [Docs](#)
- Route Interception & Modals – [Docs](#)
- Partial Prerendering – [Docs](#)
- Advanced Metadata & SEO (OpenGraph, dynamic metadata) – [Docs](#)
- Testing (unit, e2e, integration) – [Docs](#)
- Analytics & Monitoring – [Docs](#)

- Deployment to Vercel/Netlify/Cloud – [Docs](#)
- Security, CORS, API rate limiting – [Docs](#), [CORS Docs](#)
- Custom Webpack/Babel config – [Docs](#)
- Static Export ( `next export` ) – [Docs](#)
- Multi-zones – [Docs](#)

For a full overview, see the [Next.js Documentation](#).

## Q&A / Discussion

- What challenges have you faced with Next.js?
- Which feature are you most excited to try?
- Any questions about the examples or exercises?

# Thank You!

**Martin Krištof**

[GitHub Repo](#)

[My Website](#)