Module 4: Next.js

App Router, Server Components & Server Actions

Adrián Catalán

adriancatalan@galileo.edu

Agenda

1. **Project: EventPass**
2. **App Router & Server Components**
3. **Server Actions (No manual API)**
4. **Streaming & Suspense**
5. **Deep Dive**
6. **Challenge Lab**

EventPass App

We are building an Event Management platform with Next.js 16.

Core Requirements:

1. **Event Listing**: Display events with filtering options.
2. **Event Details**: Show full event information.
3. **Registration**: Allow users to register for events.
4. **Event Creation**: Form to create new events.


center

Why Next.js?

React is a library. Next.js is a **framework** built on React.

Feature	React (Vite)	Next.js
Routing	Manual (react-router)	Built-in (file-based)
Rendering	Client only	Server + Client
Data Fetching	Client side	Server side
SEO	Challenging	Built-in
API Routes	Separate backend	Integrated

2. Next.js App Router

File-Based Routing

The file structure **is** the routing structure.

```
app/
├── page.tsx              → /
├── layout.tsx            → Shared layout
├── events/
│   ├── page.tsx          → /events
│   ├── new/
│   │   └── page.tsx      → /events/new
│   └── [id]/
│       └── page.tsx      → /events/123 (dynamic)
├── loading.tsx           → Loading UI
├── error.tsx             → Error UI
└── not-found.tsx         → 404 page
```

Special Files

File	Purpose
`page.tsx`	UI for a route (required for route to work)
`layout.tsx`	Shared UI wrapper (persists across navigation)
`loading.tsx`	Loading UI (Suspense boundary)
`error.tsx`	Error UI (Error boundary)
`not-found.tsx`	404 page
`route.ts`	API endpoint

Root Layout

Required file that wraps all pages.

```
// app/layout.tsx
export default function RootLayout({
    children
}: {
    children: React.ReactNode
}) {
    return (
        <html lang="es">
            <body>
                <Header />
                <main>{children}</main>
                <Footer />
            </body>
        </html>
    );
}
```

Dynamic Routes

Capture URL segments as parameters.

```
// app/events/[id]/page.tsx
interface PageProps {
    params: Promise<{ id: string }>;
}

export default async function EventPage({ params }: PageProps) {
    const { id } = await params;
    const event = await getEventById(id);

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

    return (
        <div>
            <h1>{event.title}</h1>
            <p>{event.description}</p>
        </div>
    );
```

Loading UI

Automatic loading states with Suspense.

```
// app/events/loading.tsx
export default function Loading() {
    return (
        <div className="flex items-center justify-center h-64">
            <div className="animate-spin rounded-full h-12 w-12 border-b-2 border-blue-500" />
        </div>
    );
}
```

Next.js wraps your page in a `<Suspense>` boundary automatically.

```
// Conceptually what happens:
<Suspense fallback={<Loading />}>
    <EventsPage />
</Suspense>
```

Error Handling

Catch errors at route boundaries.

```tsx
// app/events/error.tsx
'use client'; // Error components must be Client Components

interface ErrorProps {
    error: Error;
    reset: () => void;
}

export default function Error({ error, reset }: ErrorProps) {
    return (
        <div className="text-center py-8">
            <h2>Something went wrong</h2>
            <p>{error.message}</p>
            <button onClick={reset}>
                Try again
            </button>
        </div>
    );
```

Link Component

Client-side navigation with prefetching.

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

function Navigation() {
    return (
        <nav>
            <Link href="/">Home</Link>
            <Link href="/events">Events</Link>
            <Link href="/events/new">Create Event</Link>

            {/* Dynamic link */}
            <Link href={`/events/${event.id}`}>
                View Details
            </Link>
        </nav>
    );
}
```

Search Params

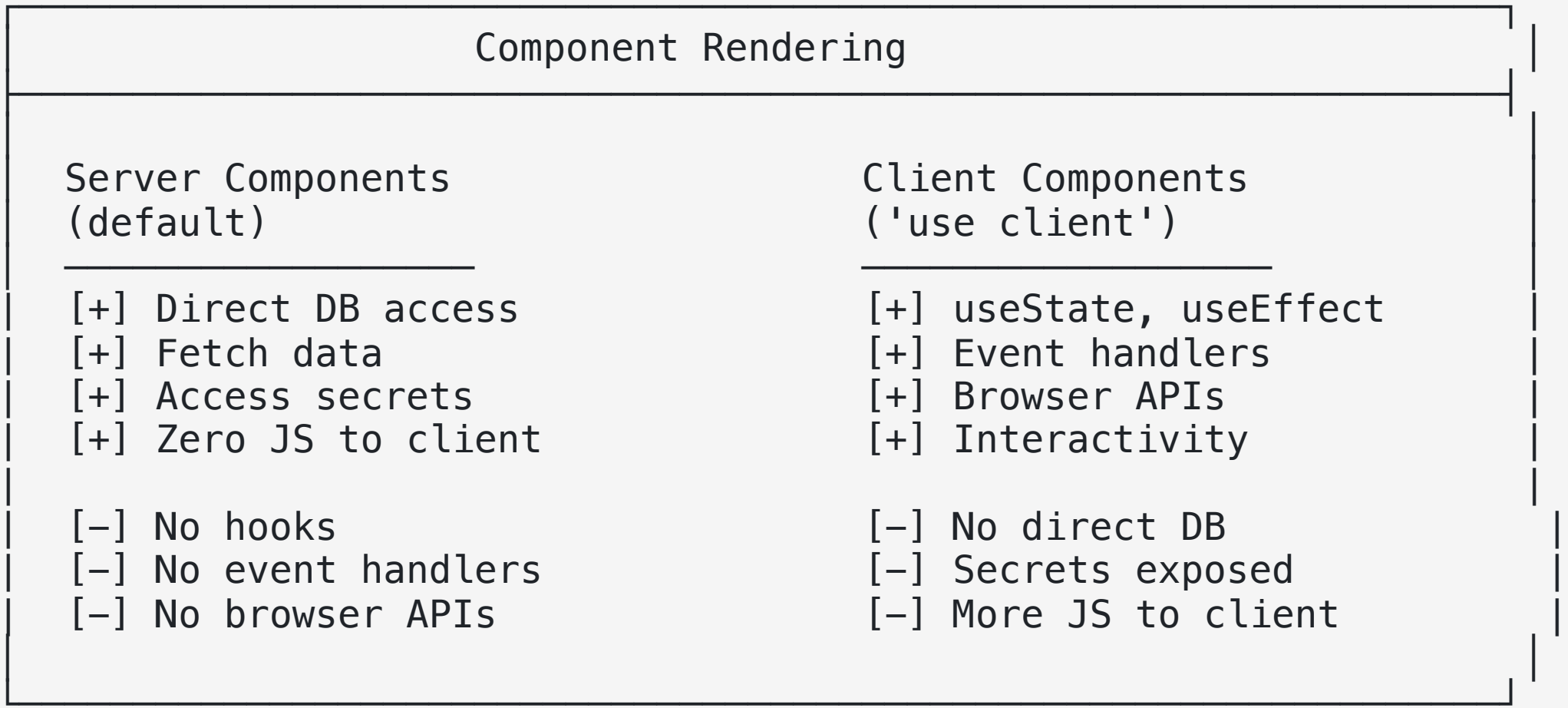
Access URL query parameters.

```
// /events?category=music&status=published

interface PageProps {
    searchParams: Promise<{
        category?: string;
        status?: string;
    }>;
}

export default async function EventsPage({ searchParams }: PageProps) {
    const { category, status } = await searchParams;

    const events = await getEvents({
        category,
        status
    });

    return <EventList events={events} />;
```

3. Server Components

Client vs Server Components

Next.js 16 defaults to **Server Components**.



Server Component Example

Fetch data directly in the component.

```
// app/events/page.tsx
// This is a Server Component by default

import { getEvents } from '@/data/events';

export default async function EventsPage() {
    // This runs on the server, not the browser
    const events = await getEvents();

    return (
        <div className="grid grid-cols-3 gap-4">
            {events.map(event => (
                <EventCard key={event.id} event={event} />
            ))}
        </div>
    );
}
```

Client Component Example

For interactivity, add `'use client'` directive.

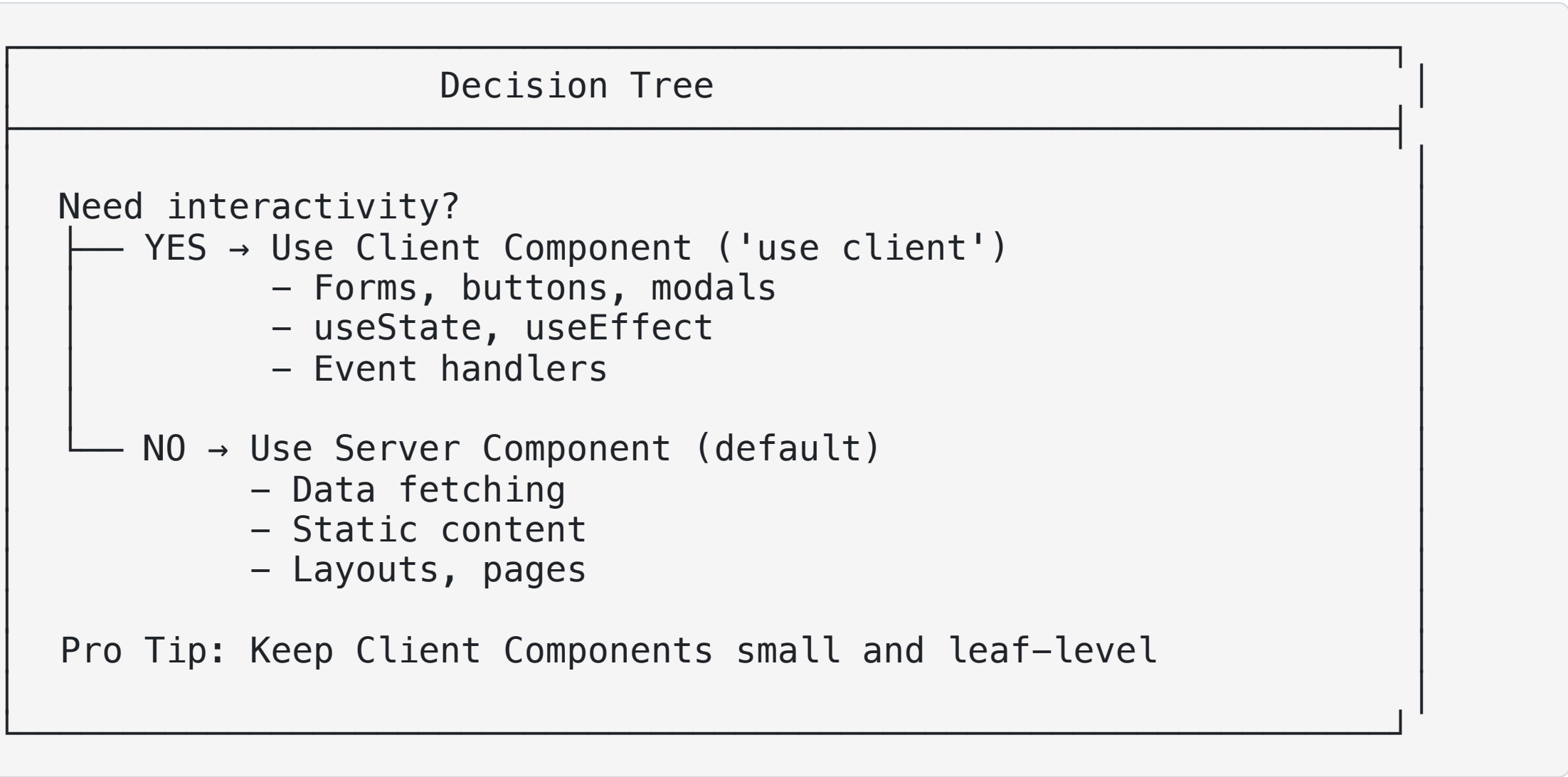
```tsx
// components/RegisterButton.tsx
'use client';

import { useState } from 'react';

export function RegisterButton({ eventId }: { eventId: string }) {
    const [isRegistered, setIsRegistered] = useState(false);

    const handleClick = async () => {
        await registerForEvent(eventId);
        setIsRegistered(true);
    };

    return (
        <button onClick={handleClick} disabled={isRegistered}>
            {isRegistered ? 'Registered' : 'Register'}
        </button>
    );
```

Composition Pattern

Server Components can render Client Components.

```
// app/events/[id]/page.tsx (Server Component)
import { RegisterButton } from '@/components/RegisterButton';

export default async function EventPage({ params }: PageProps) {
    const { id } = await params;
    const event = await getEventById(id);

    return (
        <div>
            <h1>{event.title}</h1>
            <p>{event.description}</p>

            {/* Client Component for interactivity */}
            <RegisterButton eventId={id} />
        </div>
    );
}
```

When to Use Each



4. Server Actions

What are Server Actions?

Functions that run on the server, called from the client.

```
// actions/eventActions.ts
'use server';

import { revalidatePath } from 'next/cache';

export async function createEvent(formData: FormData) {
    const title = formData.get('title') as string;
    const description = formData.get('description') as string;

    // This runs on the server
    await db.event.create({
        data: { title, description }
    });

    // Revalidate the events page cache
    revalidatePath('/events');
}
```

Using Actions in Forms

Native HTML form integration.

```
// components/EventForm.tsx
import { createEvent } from '@/actions/eventActions';

export function EventForm() {
    return (
        <form action={createEvent}>
            <input name="title" required />
            <textarea name="description" required />
            <button type="submit">
                Create Event
            </button>
        </form>
    );
}
```

No JavaScript needed - works with JS disabled!

useActionState Hook

Handle loading and error states.

```
'use client';

import { useActionState } from 'react';
import { createEvent } from '@/actions/eventActions';

export function EventForm() {
    const [state, formAction, isPending] = useActionState(
        createEvent,
        { error: null }
    );

    return (
        <form action={formAction}>
            <input name="title" disabled={isPending} />

            {state.error && <p className="error">{state.error}</p>}

            <button disabled={isPending}>
                {isPending ? 'Creating...' : 'Create Event'}
            </button>
        </form>
    );
```

Action with Validation

Return errors to the client.

```
// actions/eventActions.ts
'use server';

import { z } from 'zod';

const EventSchema = z.object({
    title: z.string().min(3),
    description: z.string().min(10)
});

export async function createEvent(prevState: any, formData: FormData) {
    const result = EventSchema.safeParse({
        title: formData.get('title'),
        description: formData.get('description')
    });

    if (!result.success) {
        return { success: false, message: result.error.issues[0].message };
    }

    await db.event.create({ data: result.data });
    revalidatePath('/events');

    // Return success state instead of redirecting
```

useFormStatus Hook

Show pending state in submit button.

```
'use client';

import { useFormStatus } from 'react-dom';

function SubmitButton() {
    const { pending } = useFormStatus();

    return (
        <button type="submit" disabled={pending}>
            {pending ? (
                <>
                    <Spinner /> Saving...
                </>
            ) : (
                'Save Event'
            )}
        </button>
    );
}

// Usage in form
<form action={createEvent}>
    <input name="title" />
```

useOptimistic Hook

Update UI before server confirms.

```
'use client';

import { useOptimistic } from 'react';

export function RegisterButton({ event }: { event: Event }) {
    const [optimisticCount, addOptimistic] = useOptimistic(
        event.registeredCount,
        (current, _) => current + 1
    );

    async function handleRegister() {
        addOptimistic(null); // Immediately show +1
        await registerForEvent(event.id); // Then confirm with server
    }

    return (
        <button onClick={handleRegister}>
            Register ({optimisticCount} / {event.capacity})
        </button>
    );
```

Revalidation Strategies

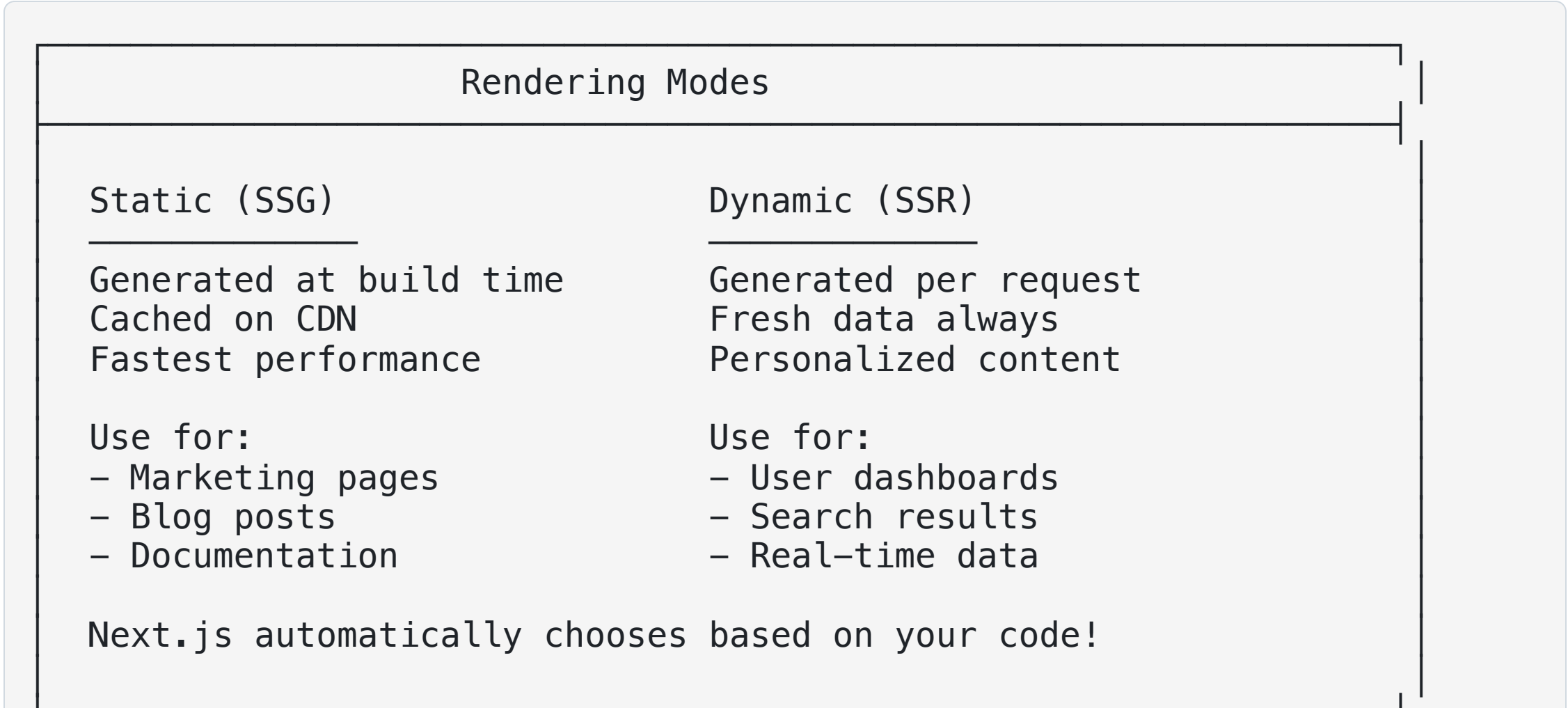
Keep data fresh after mutations.

```
'use server';

import { revalidatePath, revalidateTag } from 'next/cache';

export async function createEvent(data: EventData) {
    await db.event.create({ data });

    // Option 1: Revalidate specific path
    revalidatePath('/events');

    // Option 2: Revalidate by tag
    revalidateTag('events');

    // Option 3: Redirect (implies revalidation)
    redirect('/events');
}

// Using tags in fetch
const events = await fetch('/api/events', {
    next: { tags: ['events'] }
```

5. Deep Dive

1. Rendering Strategies

Next.js supports multiple rendering modes.



2. Static vs Dynamic Detection

Next.js analyzes your code to decide rendering strategy.

```
// STATIC - No dynamic functions used
export default async function Page() {
    const posts = await getPosts(); // Cached at build
    return <PostList posts={posts} />;
}

// DYNAMIC - Uses searchParams or cookies
export default async function Page({ searchParams }: Props) {
    const { q } = await searchParams; // Forces dynamic
    const results = await search(q);
    return <Results results={results} />;
}

// Force dynamic
export const dynamic = 'force-dynamic';
```

3. Streaming & Suspense

Progressive rendering with React Suspense.

```
// app/events/page.tsx
import { Suspense } from 'react';

export default function EventsPage() {
    return (
        <div>
            <h1>Events</h1>

            {/* This streams independently */}
            <Suspense fallback={<EventsSkeleton />}>
                <EventList />
            </Suspense>

            {/* This can load separately */}
            <Suspense fallback={<StatsSkeleton />}>
                <EventStats />
            </Suspense>
        </div>
    );
```

4. Parallel Data Fetching

Fetch data in parallel to reduce waterfall.

```
// BAD: Sequential (slow)
export default async function Page({ params }: Props) {
    const { id } = await params;
    const event = await getEvent(id);          // Wait...
    const comments = await getComments(id);  // Then wait...
    const related = await getRelated(id);    // Then wait...
}

// GOOD: Parallel (fast)
export default async function Page({ params }: Props) {
    const { id } = await params;

    const [event, comments, related] = await Promise.all([
        getEvent(id),
        getComments(id),
        getRelated(id)
    ]);
}
```

5. Server Action Security

Server Actions are endpoints - treat them as such.

```
'use server';

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

export async function deleteEvent(eventId: string) {
    // 1. Always authenticate
    const user = await auth();
    if (!user) throw new Error('Unauthorized');

    // 2. Always validate input
    if (!eventId || typeof eventId !== 'string') {
        throw new Error('Invalid event ID');
    }

    // 3. Always authorize
    const event = await db.event.findUnique({ where: { id: eventId } });
    if (event?.organizerId !== user.id) {
        throw new Error('Forbidden');
    }

    // 4. Then perform action
    await db.event.delete({ where: { id: eventId } });
```

6. Challenge Lab

Practice & Application

Part 1: Event Filters with URL State

Context:
Users want to filter events by category and status. Filters should persist in the URL for sharing and bookmarking.

Your Task:
Implement URL-based filtering that:

- Uses searchParams for filter state
- Updates URL without full page reload
- Shows active filters visually
- Clears filters with a reset button

Files to Modify:

- `app/events/page.tsx`

Part 1: Definition of Done

Criteria	Description
URL updates	Selecting filter changes URL (`?category=music`)
Filters persist	Refreshing page keeps filters
Server filtering	Filters applied on server, not client
Active state	Selected filters visually highlighted
Reset button	"Clear all" resets to no filters
Combined filters	Multiple filters work together
Empty state	"No events found" message when empty

Part 2: Optimistic Event Registration

Context:
When users click "Register", there's a delay before the count updates. Implement optimistic UI for instant feedback.

Your Task:
Implement optimistic updates that:

- Shows updated count immediately on click
- Disables button during registration
- Reverts on error
- Shows success/error feedback

Files to Modify:

- `components/RegisterButton.tsx`

Part 2: Definition of Done

Criteria	Description
Instant feedback	Count increases immediately on click
useOptimistic used	Implements React 19 useOptimistic hook
Button disabled	Cannot click while registering
Error handling	Reverts count if server fails
Success message	Shows confirmation after success
Capacity check	Cannot register if event is full
Loading indicator	Shows spinner during action

Resources & Wrap-up

Resources

Next.js

- Next.js Documentation
- App Router Guide
- Server Components
- Server Actions
- Next.js Learn Course

React 19

- React 19 Blog Post
- useActionState Hook
- useFormStatus Hook
- useOptimistic Hook

Recommended Articles

Next.js App Router

- Understanding the App Router - Vercel
- Data Fetching Patterns - Vercel
- Caching in Next.js - Vercel

Server Components

- Making Sense of RSC - Josh Comeau
- RSC From Scratch - React Working Group
- When to Use Server vs Client - Vercel