REACT ADVANCED LONDON

October

20 & 23, 2023

2 days
In-person & Remote

60+ speakers
presenting on advanced topics

800+ luckies
in London

5K devs
globally remotely

Reserve a spot →

2022 edition →

React Server Components Unleashed: A Deep Dive into Next-Gen Web Development

Maurice de Beijer
@mauricedb

- Maurice de Beijer
- The Problem Solver
- Microsoft MVP
- Freelance developer/instructor
- Currently at https://someday.com/
- Twitter: @mauricedb
- Web: http://www.TheProblemSolver.nl
- E-mail: maurice.de.beijer@gmail.com

Topics

- What are **React Server Components** and why would you care?
- Using **Next.js** and the **App Router**
- Turning a React **Client Component into** a React **Server Component**
- **Updates and caching** with React Server Components
- **Querying the database** from a React Server Component
- **Suspense** & React Server Components
- React Server Components and **streaming**
- **Which components** are really React Server Components?
- Using **React Server Actions**
- **Migrating from pages** to React Server Components with Next.js

Type it out by hand?

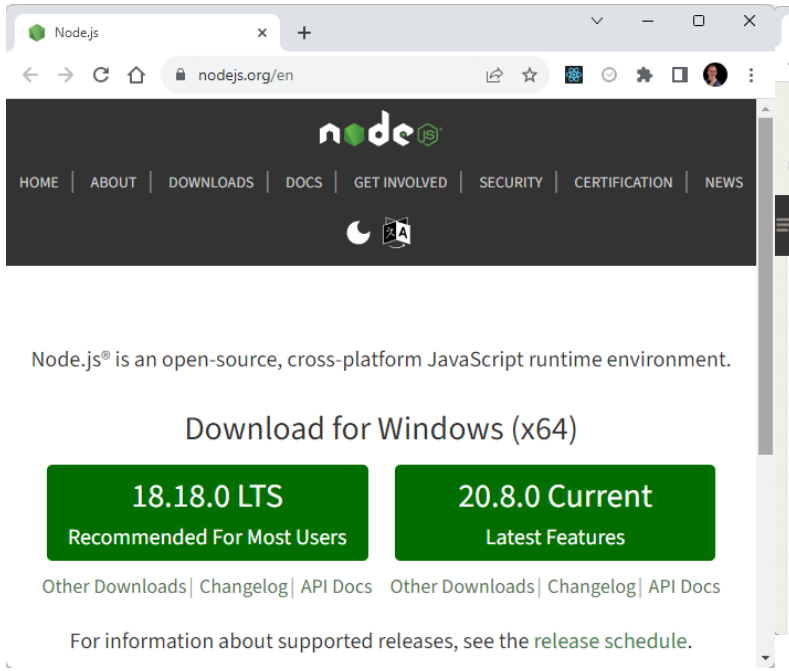
"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"

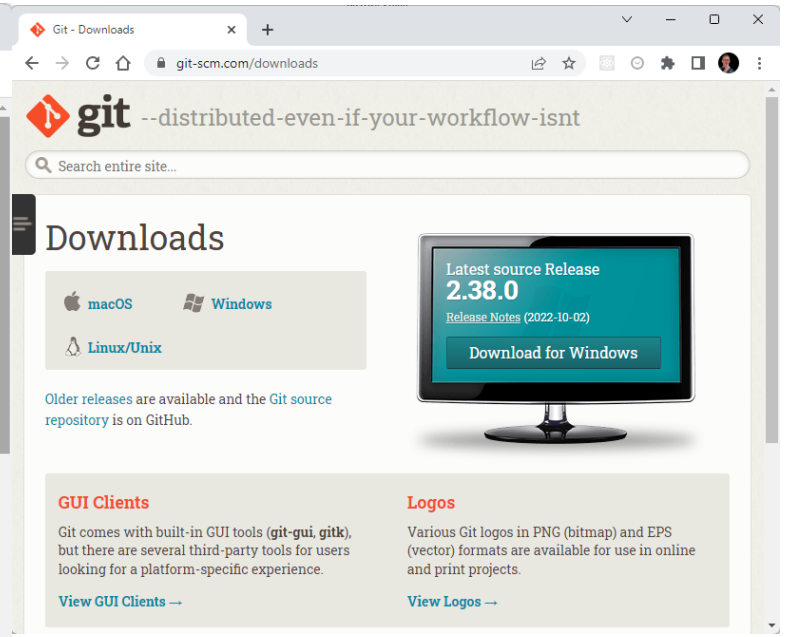
Prerequisites

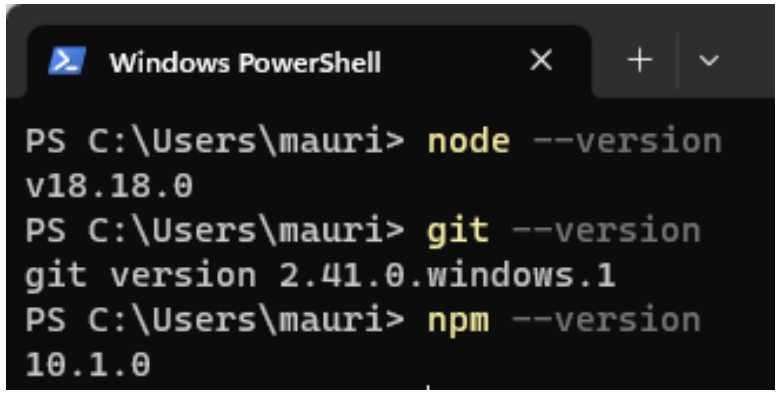
Install Node & NPM

Install the GitHub repository

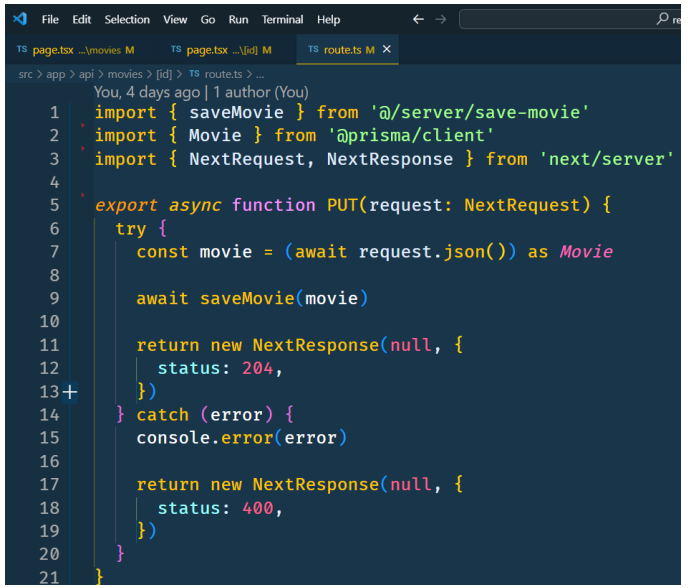
Install Node.js & NPM







Following Along



- Repo: https://github.com/mauricedb/reactadvanced-2023-ws
- Slides: https://bit.ly/reactadvanced-2023-ws

Create a new Next.js app

```
PS C:\Repos> npx create-next-app@latest
√ What is your project named? ... reactadvanced-2023-ws
√ Would you like to use TypeScript? ... No / Yes
√ Would you like to use ESLint? ... No / Yes
√ Would you like to use Tailwind CSS? ... No / Yes
√ Would you like to use `src/` directory? ... No / Yes
√ Would you like to use App Router? (recommended) ... No / Yes
√ Would you like to customize the default import alias? ... No / Yes
Creating a new Next.js app in C:\Repos\reactadvanced-2023-ws.
```

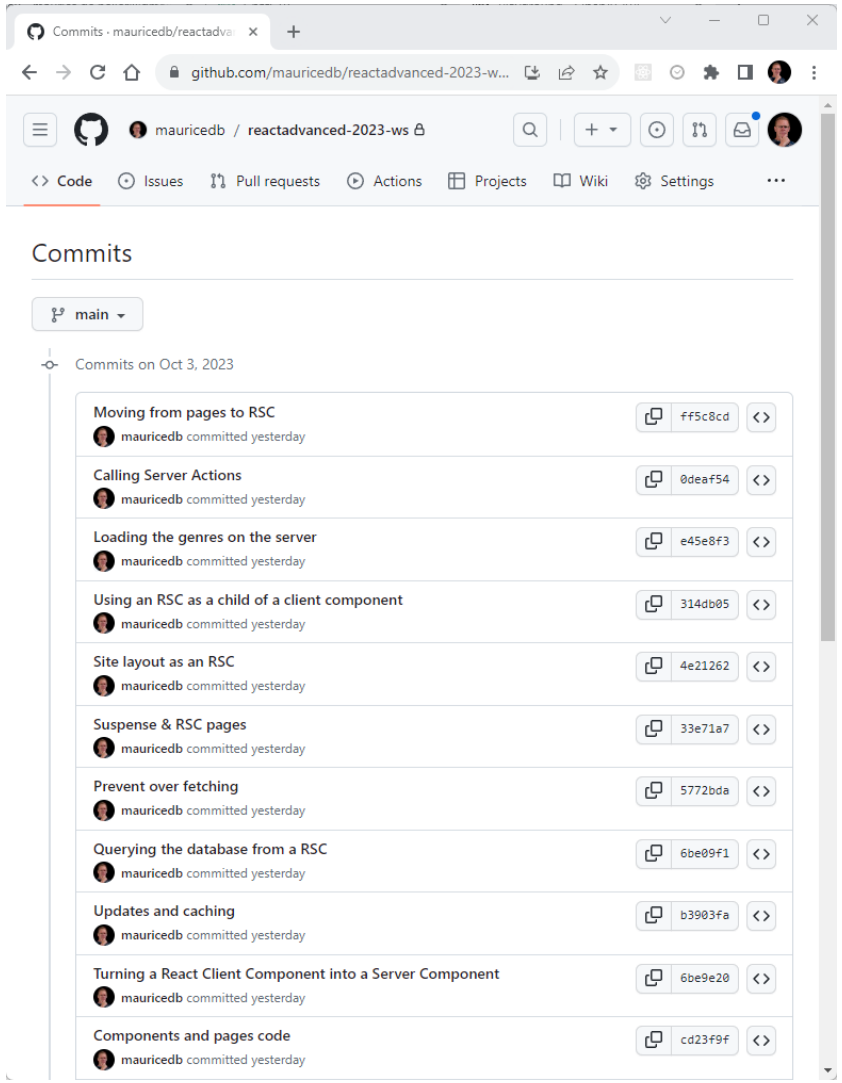
Adding Shadcn support

```
PS C:\Repos\reactadvanced-2023-ws> npx shadcn-ui@latest init
✓ Would you like to use TypeScript (recommended)? ... no / yes
✓ Which style would you like to use? » Default
✓ Which color would you like to use as base color? » Zinc
✓ Where is your global CSS file? ... src\app\globals.css
✓ Would you like to use CSS variables for colors? ... no / yes
✓ Where is your tailwind.config.js located? ... tailwind.config.js
✓ Configure the import alias for components: ... @/components
✓ Configure the import alias for utils: ... @/lib/utils
✓ Are you using React Server Components? ... no / yes
✓ Write configuration to components.json. Proceed? ... yes

✓ Writing components.json...
✓ Initializing project...
✓ Installing dependencies...

Success! Project initialization completed.
```

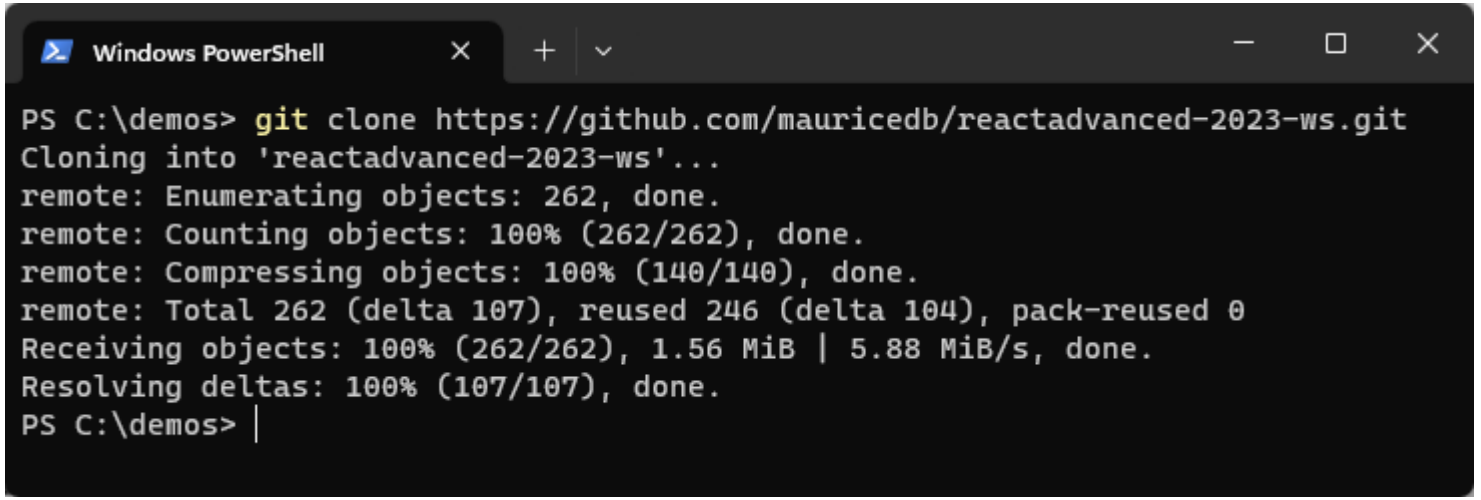
Adding Shadcn components

```
PS C:\Repos\reactadvanced-2023-ws> npx shadcn-ui@latest add button card command dialog form input label popover textarea toast
✔ Ready to install components and dependencies. Proceed? ... yes
✔ Done.
```

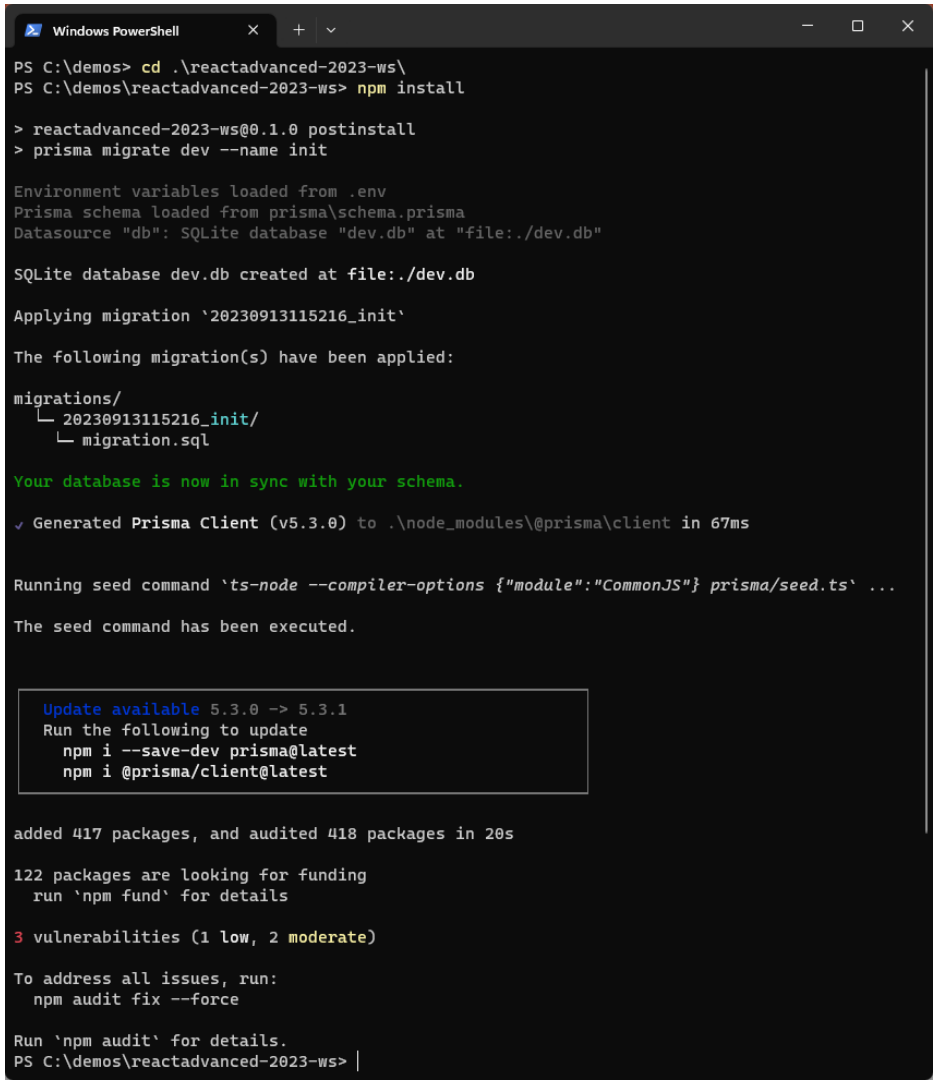
The changes



Clone the GitHub Repository



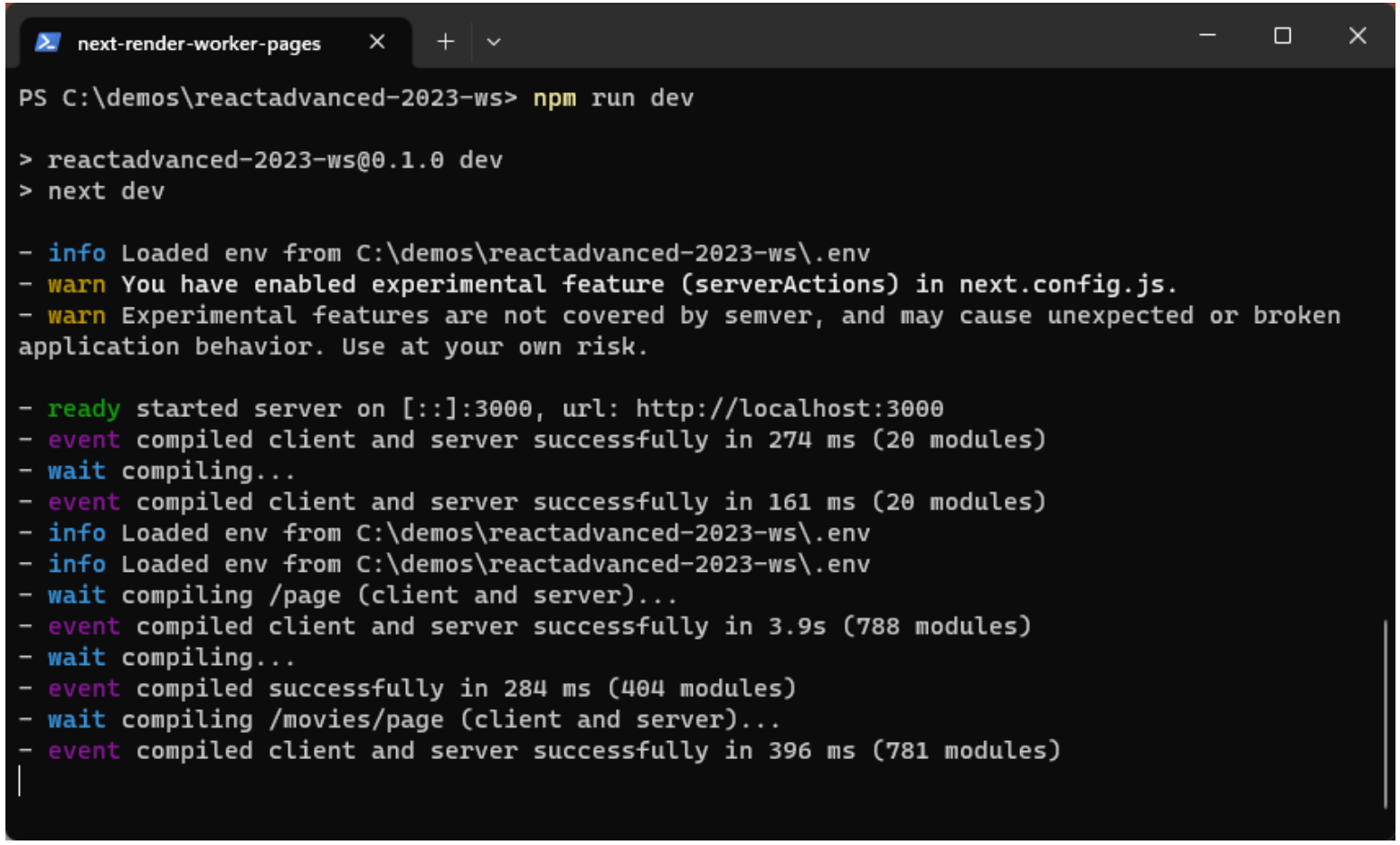
Install NPM Packages



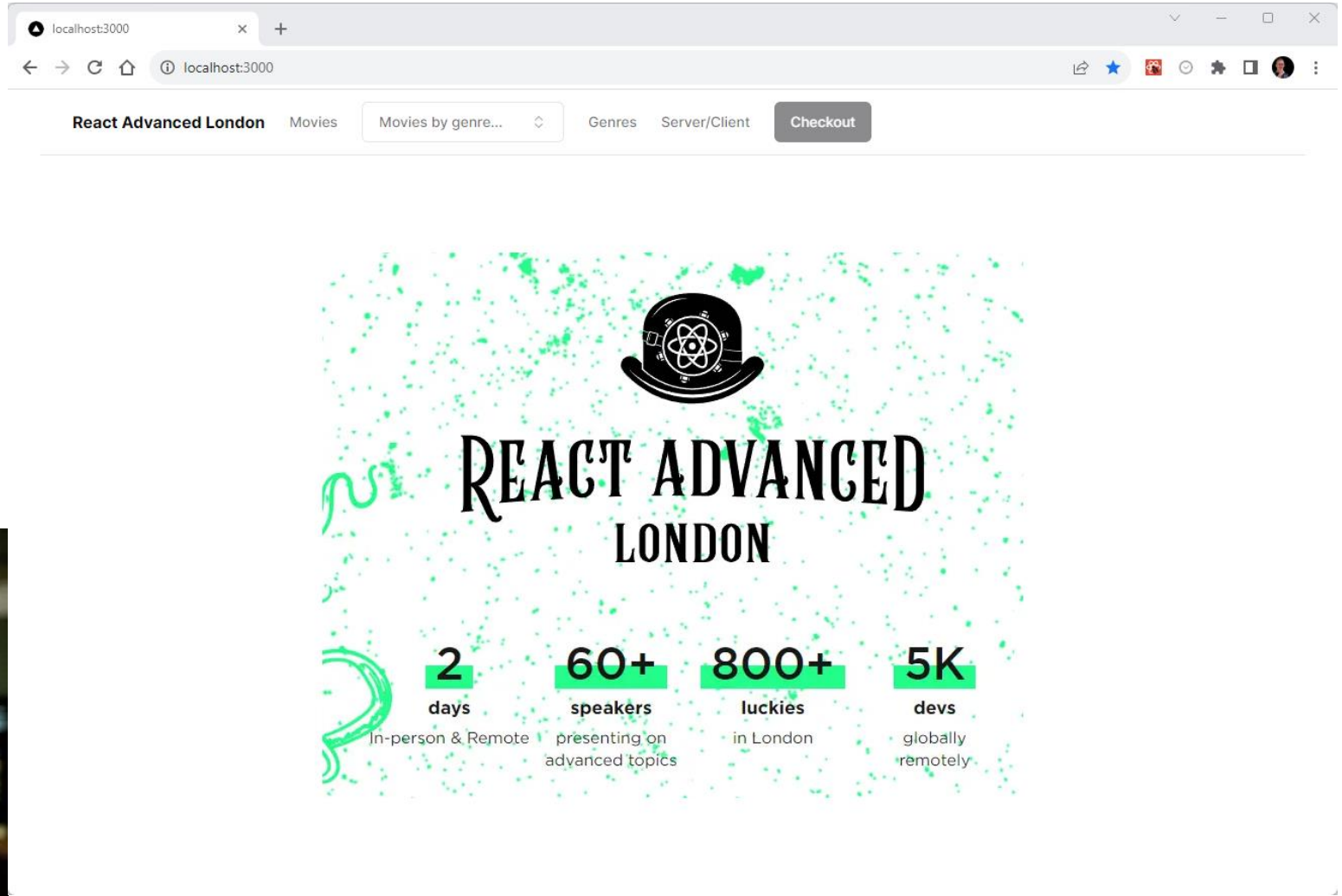
Start branch

- Start with the **00-start** branch
  - `git checkout --track origin/00-start`

Start the application



The application

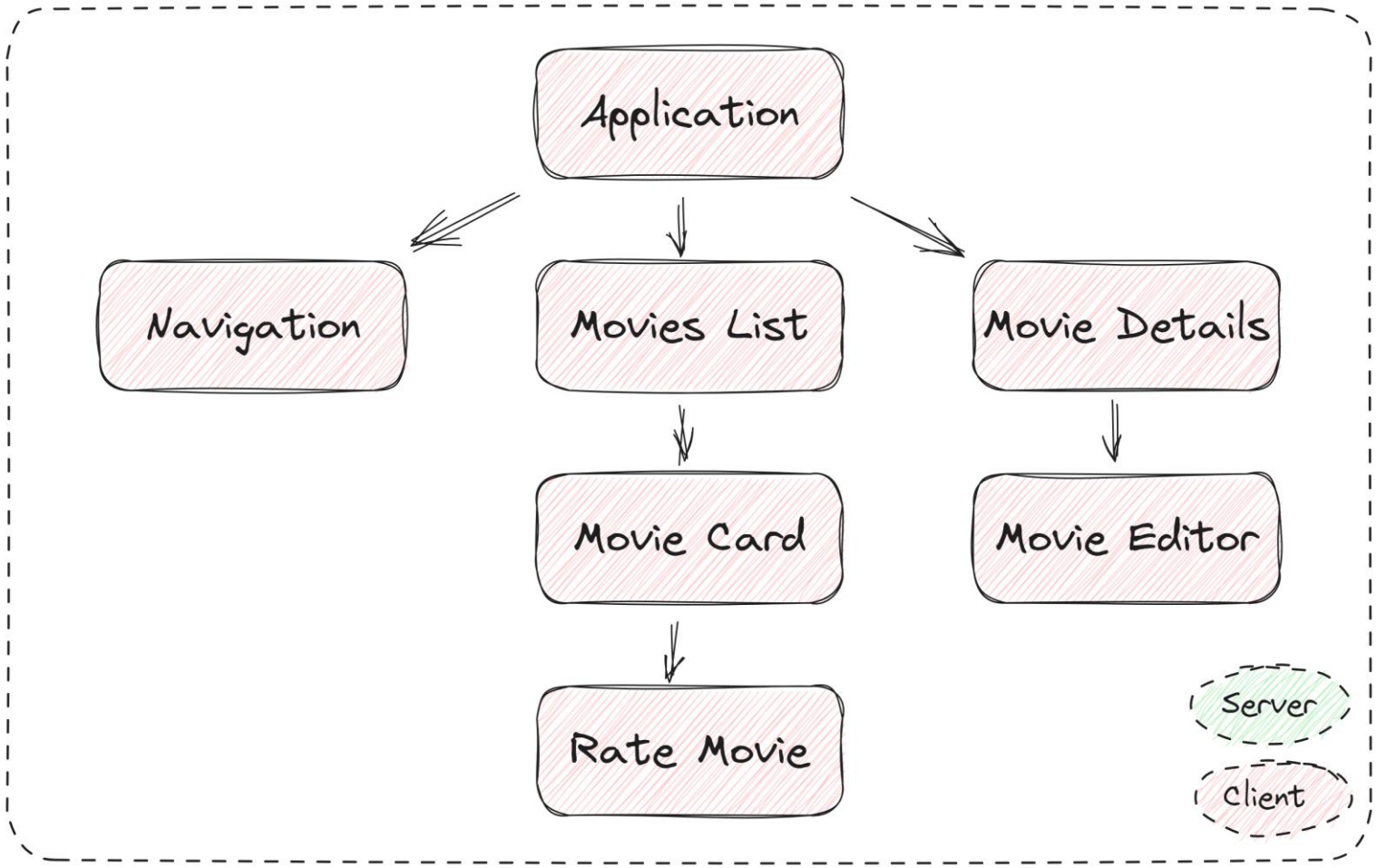


What are React Server Components?

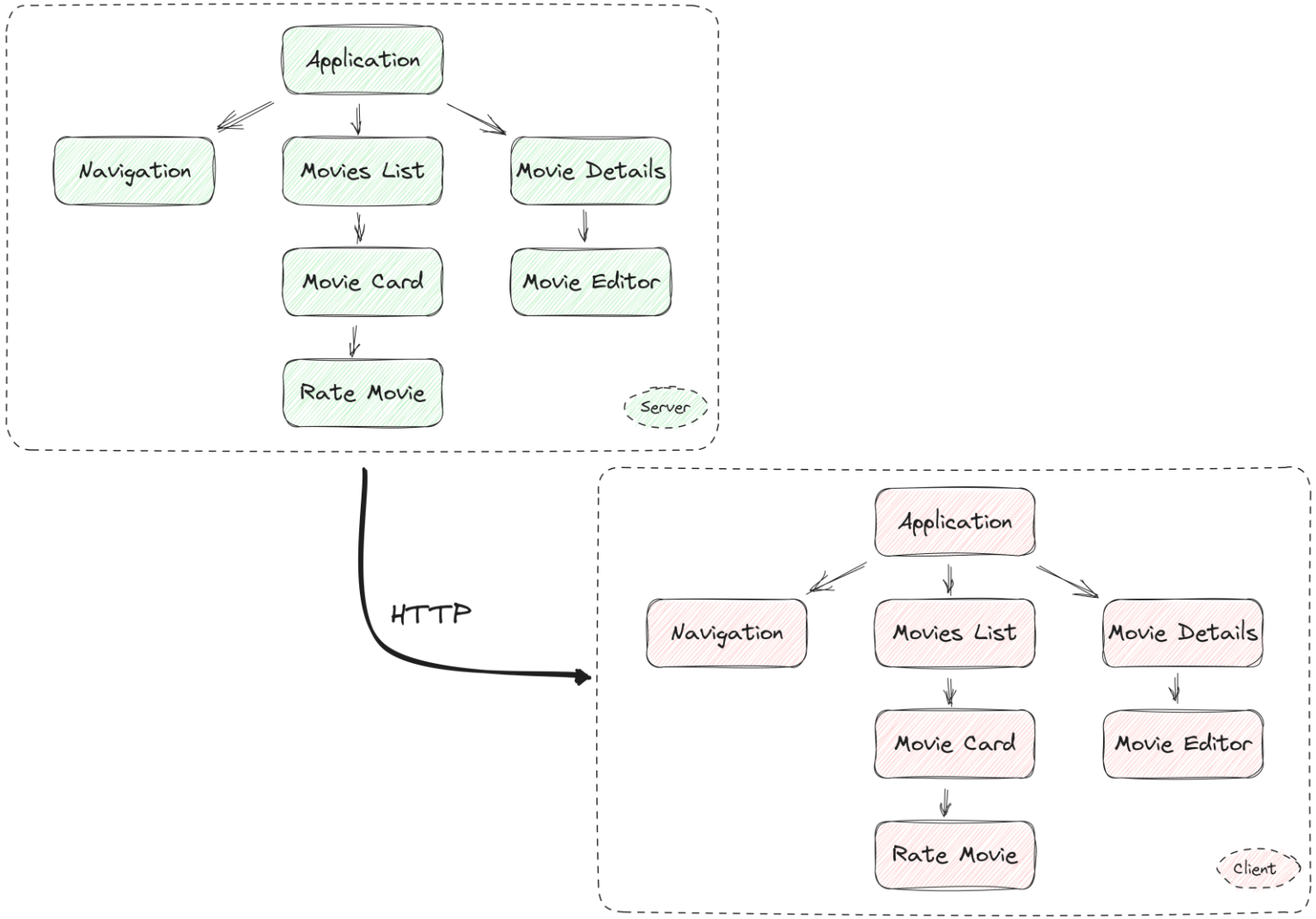
React Server Components

- React Server Components (RSC) **only execute on the server**
  - Traditionally React components always execute in the browser
- RSC are **not the same as Server Side Rendering**
  - With SSR components are executed both on the client and server
- Applications are a **combination of server and client components**
- The result: The back and front-end **code are more integrated**
  - Leading to **better type safety** ☺

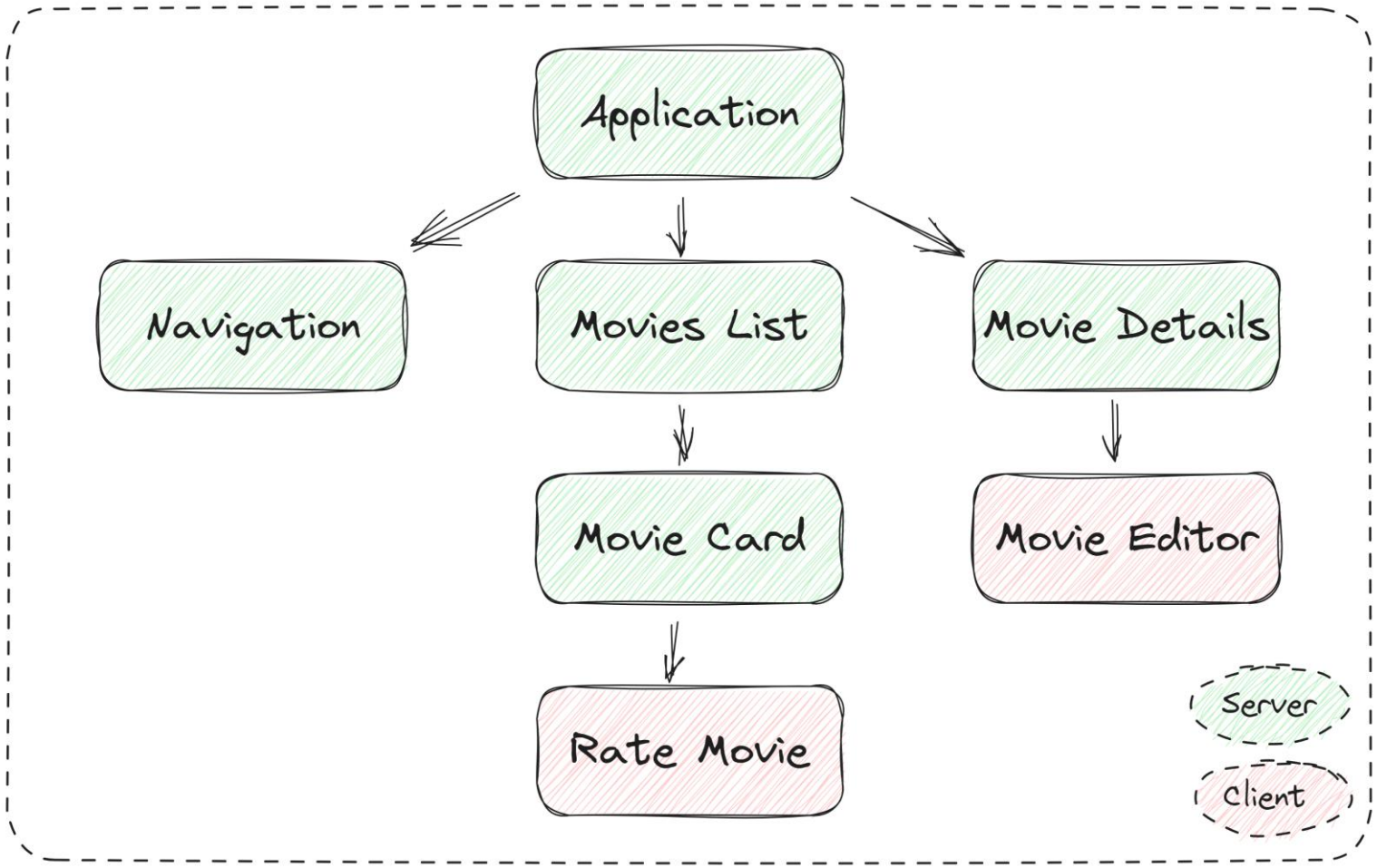
Before RSC


Application
Navigation
Movies List
Movie Details
Movie Card
Movie Editor
Rate Movie
Server
Client

Server Side Rendering



With RSC


Application
Navigation
Movies List
Movie Details
Movie Card
Movie Editor
Rate Movie
Server
Client

React Server Components

- Server components can be **asynchronous**
  - Great to load data from some API
- Server components **render just once**
  - No re-rendering with state changes or event handling
- The server component **code is not send to the browser**
  - Can safely use secure API key's etc.
  - Smaller bundle sizes
- ☞ React Server Components require TypeScript 5.1 ☜

React Server Component

TS genre-loader.tsx ×

src > components > TS genre-loader.tsx > ...

You, 2 weeks ago | 1 author (You)

```tsx
import { prisma } from '@/lib/db'
import { GenreSelector } from '@/components/genre-selector'

export async function GenreLoader() {
  const genres = await prisma.genre.findMany({
    orderBy: { name: 'asc' },
  })

  return <GenreSelector genres={genres} />
}
```

React Client Components

- **Server components can render both server and client components**
  - Client components can only render other client components
- Adding **'use client'** to the top of a component makes it a client component
  - Used as a directive for the bundler to include this in the client JS bundle
- A client component is **still executed on the server** as part of SSR
  - When using Next.js

Next.js and the App Router

Next.js and the App Router

- **React is no longer just a client side library**
  - We need additional server side capabilities
  - As well as additional code bundling options
- **Next.js is the best production option available**
  - Shopify Hydrogen is also an option
  - ☞ Remix 2 doesn't support RSC yet ☜
- There are also more **experimental options**
  - Waku from Daishi Kato
  - React Server Components Demo from the React team

Rendering RSC's

- **React Server Components are only rendered on the server**
  - And shipped to the client as a JSON like structure
  - The React Server Component Payload
- The client then **injects** these JSON objects **into the React tree**
  - Where it would previously have rendered these components themself

- ☞ **React already used a 2 step process** ☜
  - Components render to a virtual DOM
    - Just a series of JavaScript objects
  - Reconciliation maps the virtual DOM to the browser DOM
    - Or an HTML stream in the case or Server Side Rendering

Async transport

- RSC's are **streamed asynchronously** to the client
  - Enables using Suspense boundaries while loading

Code bundling

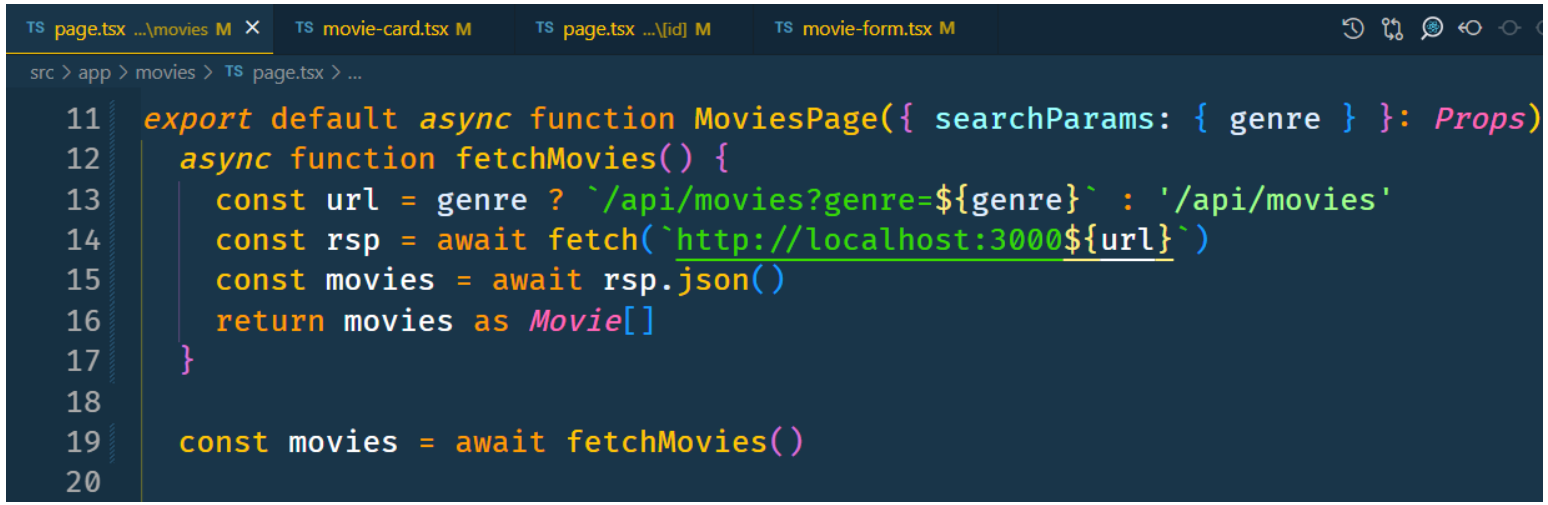
- **Multiple JavaScript bundles** have to be made
  - The client and server have different code bundles
- **Server Component code is never executed on the client**
  - Can use **react-server-dom-webpack** or a similar package

Turning a React Client Component into a Server Component

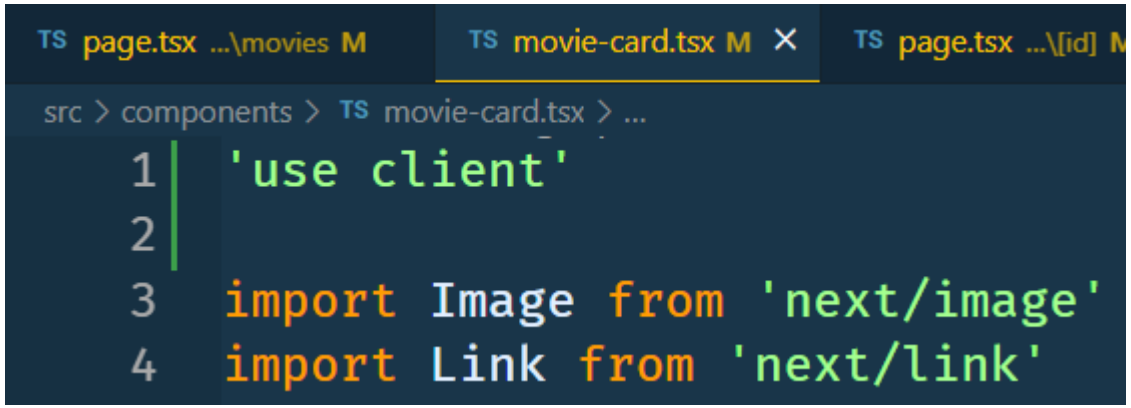
Client Component to Server Component

- **React Server Components normally perform better**
  - Only render once on the server
  - The code doesn't need to be shipped to the browser
- **Can be async and await data** to be fetched
  - No need for a render/effect/re-render cycle in the browser
- **Components that don't need client capabilities should be SRC's**
  - State, effects, browser API's etc. are client requirements

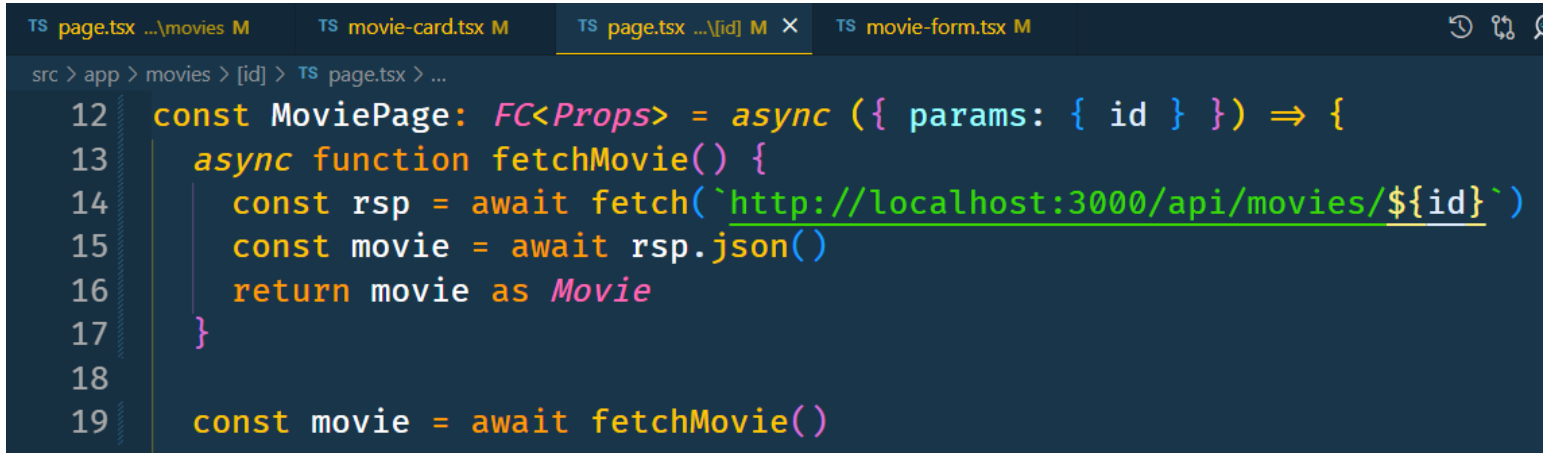
movies /page.tsx



movie-card.tsx



movies/[id]/page.tsx



movie-form.tsx

TS page.tsx ...\movies M | TS movie-card.tsx M | TS page.tsx ...\[id] M | TS movie-form.tsx M

src > components > TS movie-form.tsx > ...

You, 5 minutes ago | 1 author (You)

```tsx
'use client'

import { zodResolver } from '@hookform/resolvers/zod'
import * as z from 'zod'
```

Updates and caching

Updates and caching

- Next.js does a lot of **optimizations using caching**
  - Both on the server and client
- The Next.js uses a **Data Cache** and **Full Router Cache** on the server
  - Use ***export const dynamic = 'force-dynamic'*** to make sure data on the server isn't cached
  - Can also be controlled at the fetch() level
- The Next.js uses a **Router Cache** on the client
  - Dynamically rendered routes are purged after 30 seconds
  - Call ***router.refresh()*** to immediately purge the cache
    - Make sure to use the router from '*next/navigation'*

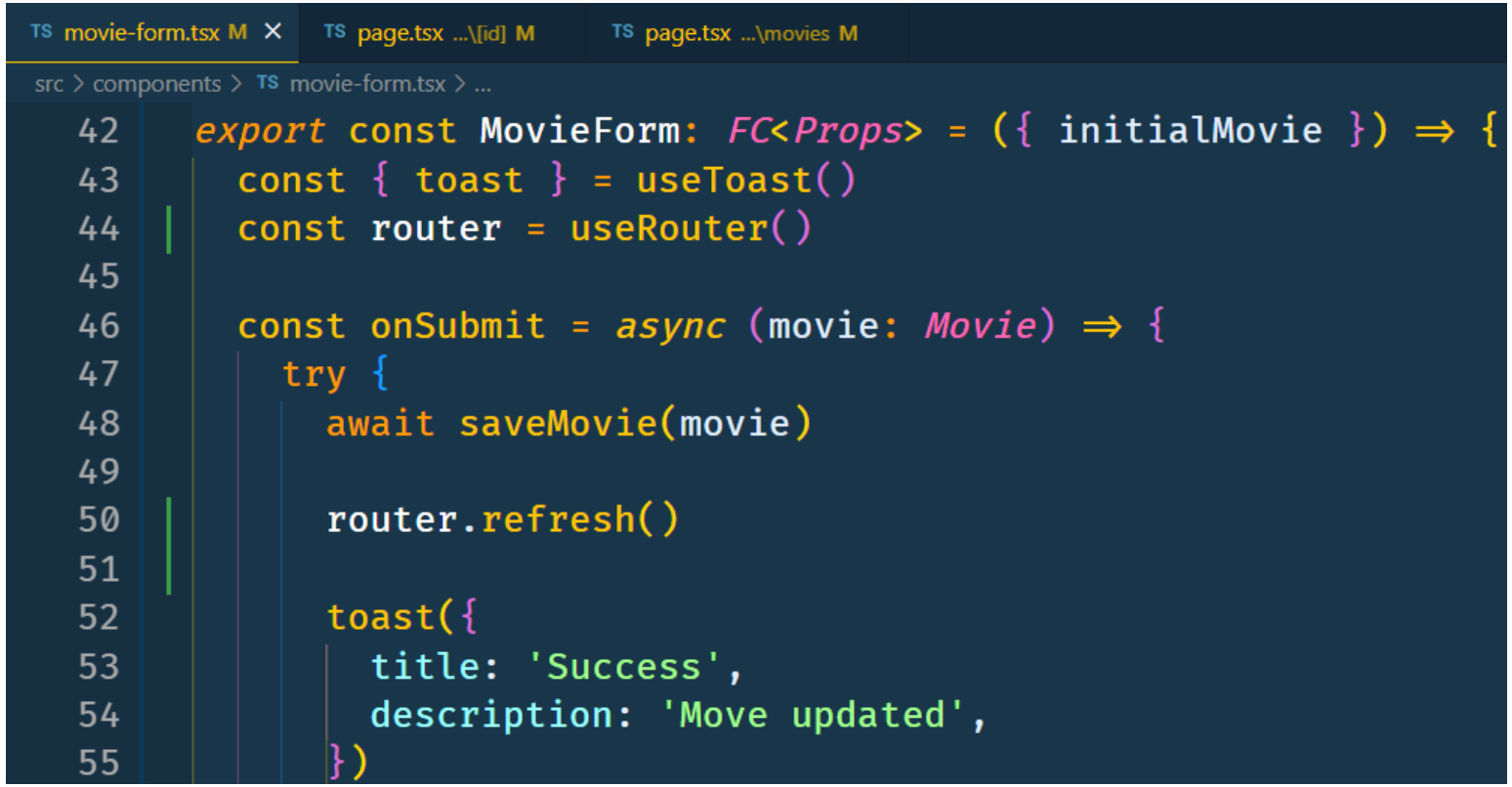
movies/[id]/page.tsx

TS movie-form.tsx M | TS page.tsx ...\[id] M × | TS page.tsx ...\movies M

src > app > movies > [id] > TS page.tsx > ...

```tsx
type Props = {
  params: {
    id: string
  }
}

export const dynamic = 'force-dynamic'

const MoviePage: FC<Props> = async ({ params: { id } }) => {
  async function fetchMovie() {
    const rsp = await fetch(`http://localhost:3000/api/movies/${id}`)
    const movie = await rsp.json()
    return movie as Movie
  }
```

movie-form.tsx



movies /page.tsx

TS movie-form.tsx M | TS page.tsx ...\[id] M | TS page.tsx ...\movies M

src > app > movies > TS page.tsx > ...

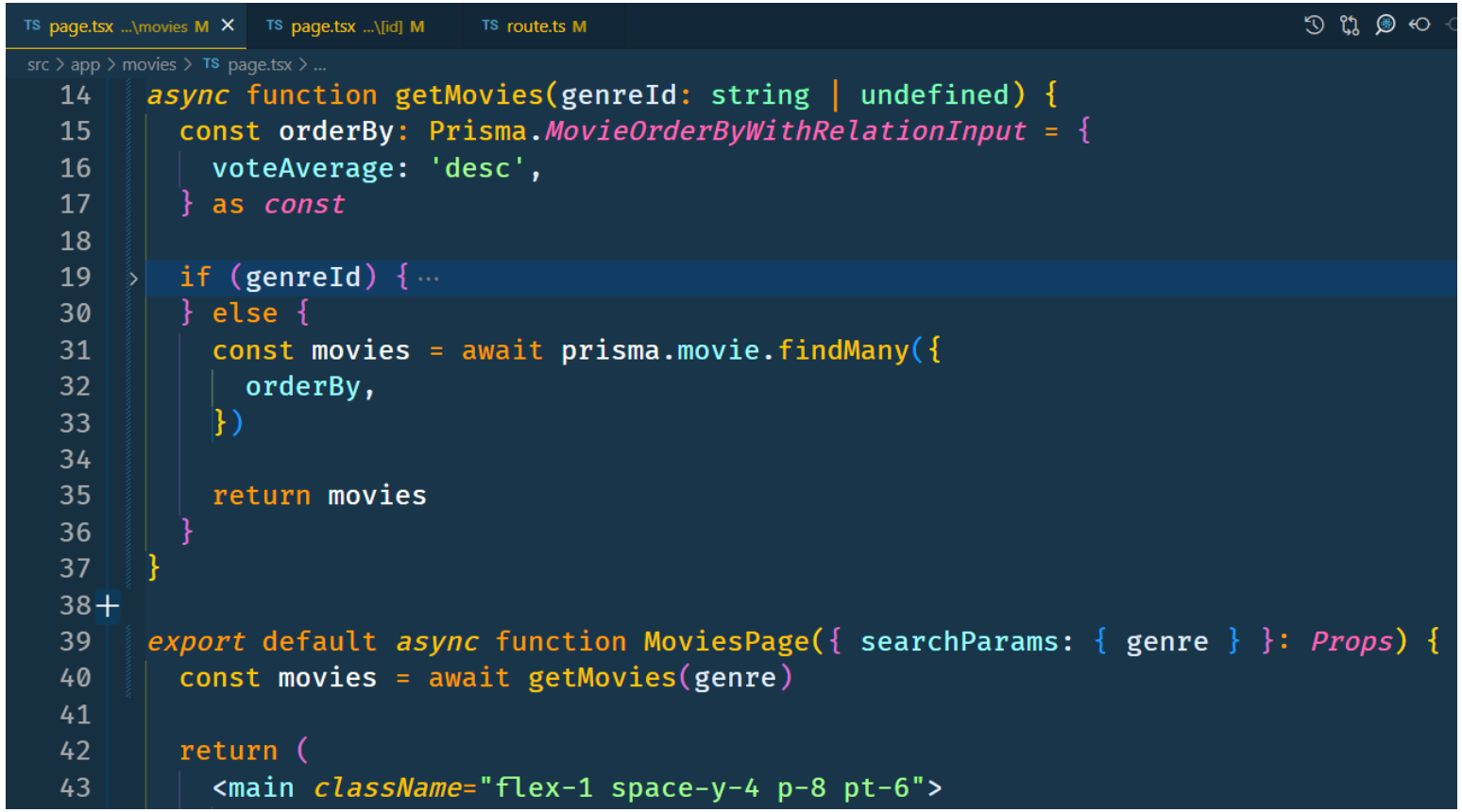
```tsx
type Props = {
  searchParams: {
    genre?: string
  }
}

export const dynamic = 'force-dynamic'

export default async function MoviesPage({ searchParams: { genre } }: Props) {
  async function fetchMovies() {
    const url = genre ? `/api/movies?genre=${genre}` : '/api/movies'
    const rsp = await fetch(`http://localhost:3000${url}`)
    const movies = await rsp.json()
    return movies as Movie[]
  }
```

Querying the database from an RSC

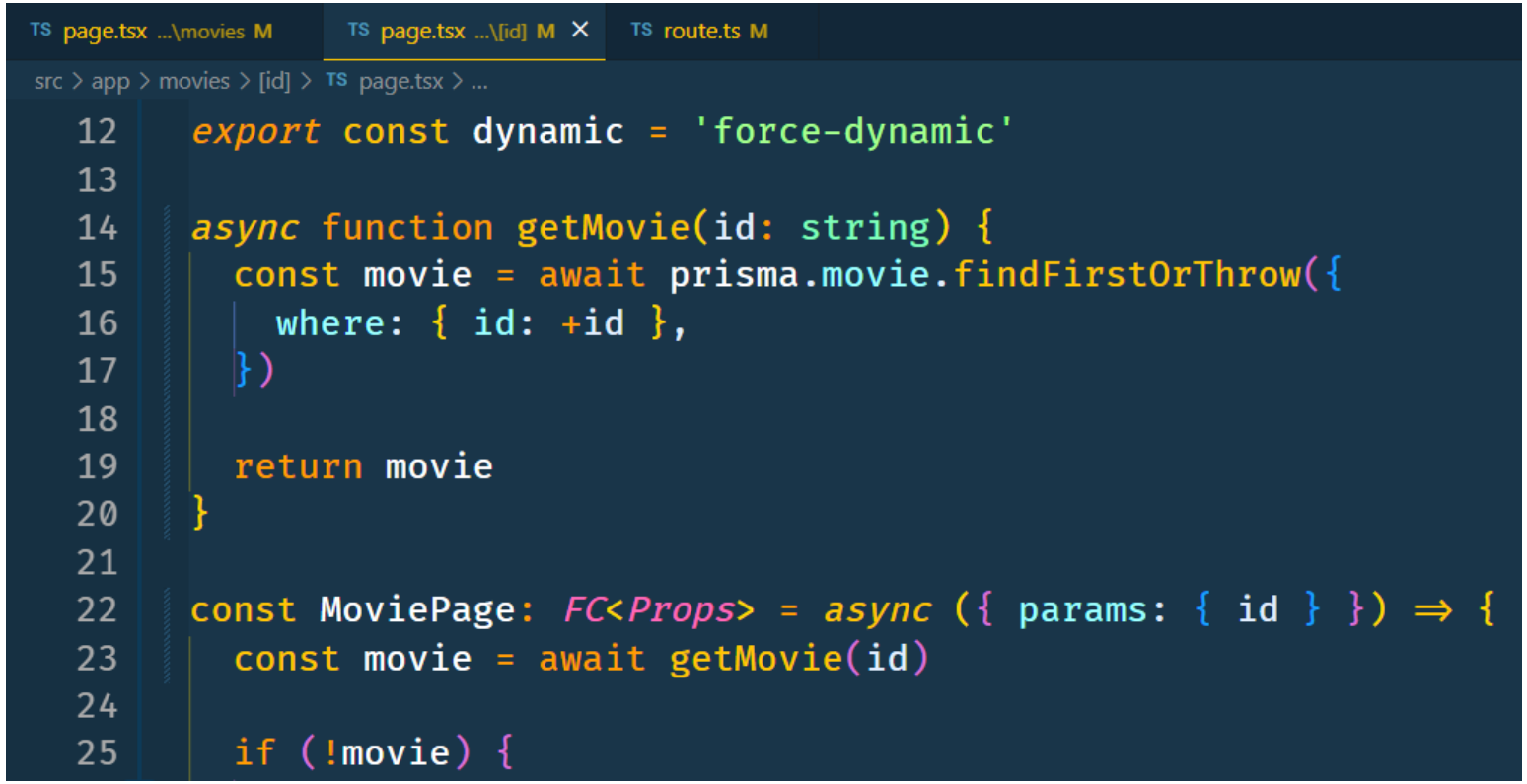
Querying the database from an RSC

- Because an **RSC** only runs on the server we **can use server side code**
  - Query the DB using Prisma directly
  - It's save to use secrets like database connection strings
- **Never executed in the browser**
  - Leads to smaller JavaScript bundle sizes

movies /page.tsx



movies/[id]/page.tsx



api/movies/[id]/route.ts

File Edit Selection View Go Run Terminal Help

page.tsx ...\movies M | page.tsx ...\[id] M | route.ts M

src > app > api > movies > [id] > route.ts > ...

You, 4 days ago | 1 author (You)

```ts
import { saveMovie } from '@/server/save-movie'
import { Movie } from '@prisma/client'
import { NextRequest, NextResponse } from 'next/server'

export async function PUT(request: NextRequest) {
  try {
    const movie = (await request.json()) as Movie

    await saveMovie(movie)

    return new NextResponse(null, {
      status: 204,
    })
  } catch (error) {
    console.error(error)

    return new NextResponse(null, {
      status: 400,
    })
  }
}
```

Prevent over fetching

Prevent over fetching

- **Colocation** of DB queries with components **enables more optimizations**
  - Fetch exactly the right amount of data
  - No more shared REST queries

movies /page.tsx


```tsx
type MovieForCard = ComponentProps<typeof MovieCard>['movie']

export const dynamic = 'force-dynamic'

async function getMovies(genreId: string | undefined): Promise<MovieForCard[]> {
  const orderBy: Prisma.MovieOrderByWithRelationInput = {
    voteAverage: 'desc',
  } as const

  const select = {
    id: true,
    title: true,
    overview: true,
    backdropPath: true,
    voteAverage: true,
    voteCount: true,
  } satisfies Prisma.MovieSelect

  if (genreId) {
    const genre = await prisma.genre.findFirst({
      where: { id: +genreId },
      include: {
        movies: {
          orderBy,
          select,
        },
      },
```

Break time

Suspense & RSC pages

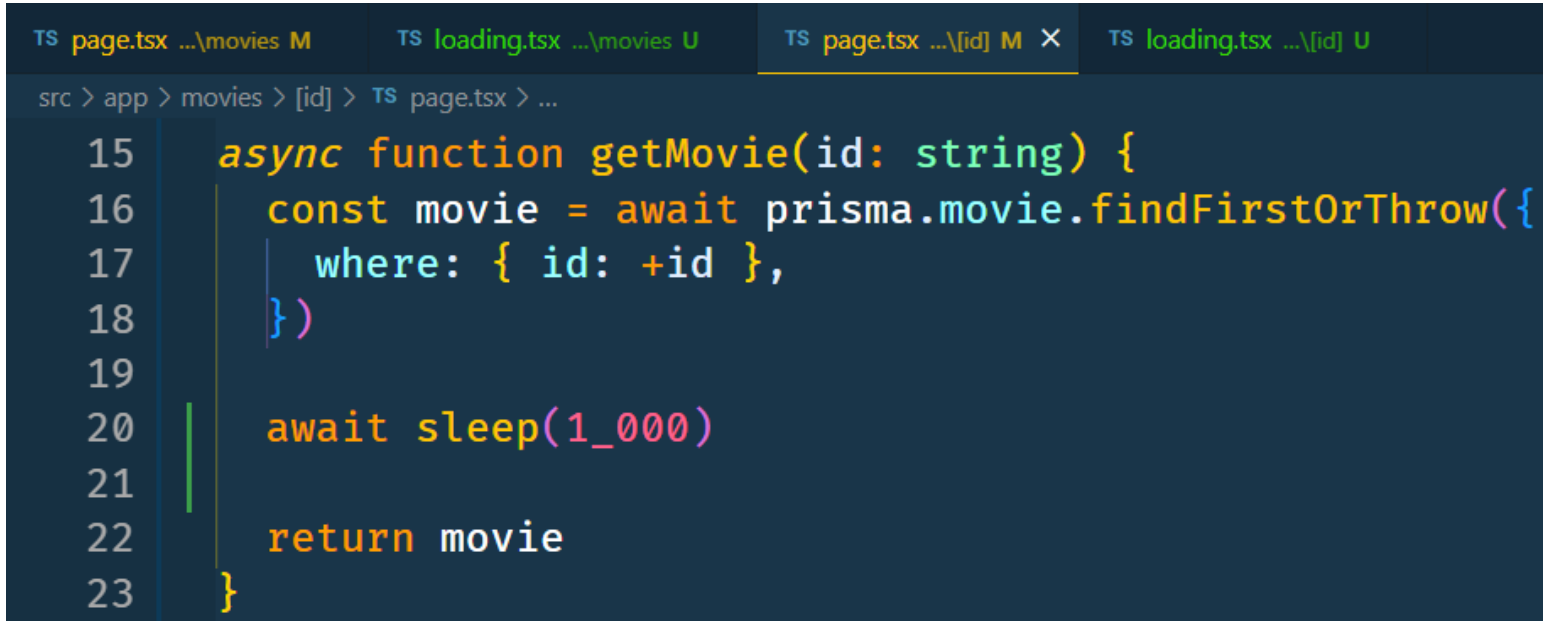
Suspense & RSC pages

- React Server Components are **suspended until they resolve**
  - Can be controlled with **<Suspense /> boundaries**
- Next.js makes it easy to **suspend when rendering an async page**
  - **Add a loading.tsx**
  - They can be nested and the closest loading component will be used

movies /page.tsx

movies /loading.tsx

movies/[id]/page.tsx



movies/[id] /loading.tsx

TS page.tsx ...\movies M | TS loading.tsx ...\movies U | TS page.tsx ...\[id] M | TS loading.tsx ...\[id] U

src > app > movies > [id] > TS loading.tsx > ...

```tsx
export default function LoadingMovie() {
  return (
    <main className="absolute left-1/2 top-2/4 -translate-x-1/2 -translate-y-1/2">
      Loading movie ...
    </main>
  )
}
```

RSC and streaming

RSC and streaming

- **Async React Server Components are streamed to the browser**
  - Using the React Server Component Payload
  - On the client they are suspended until the component resolves
- **Server action responses** can also stream components back
  - After a ***revalidatePath()*** or a ***revalidateTag()***

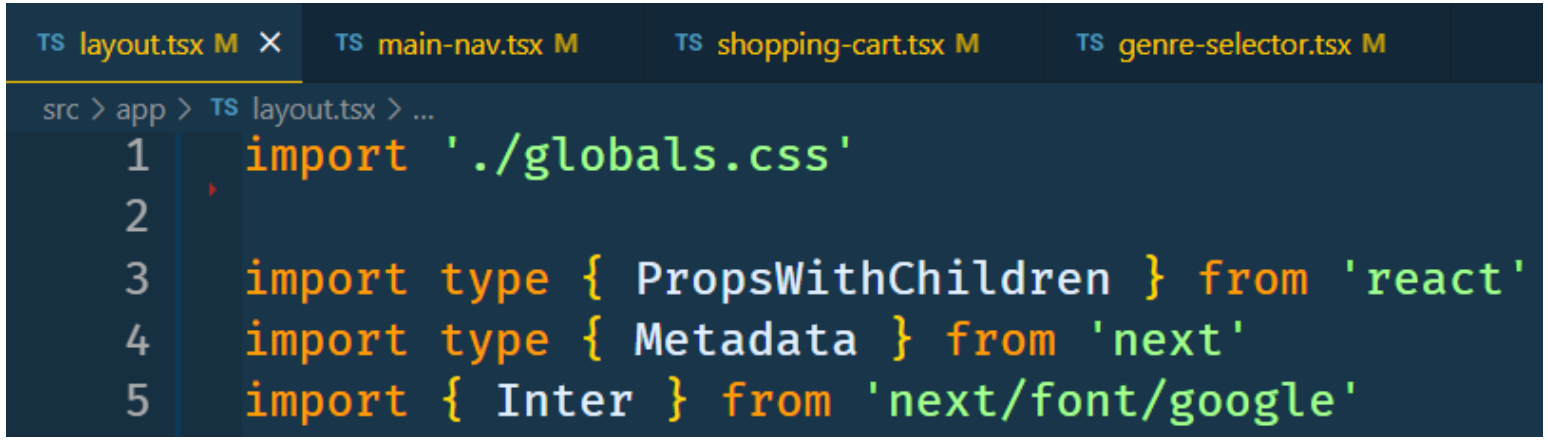
RSC Payload

Site layout as an RSC

Site layout as an RSC

- A **layout.tsx is typically a React Server Component**
  - But can be a client component if required
- **Render server and/or client components** as needed

layout.tsx



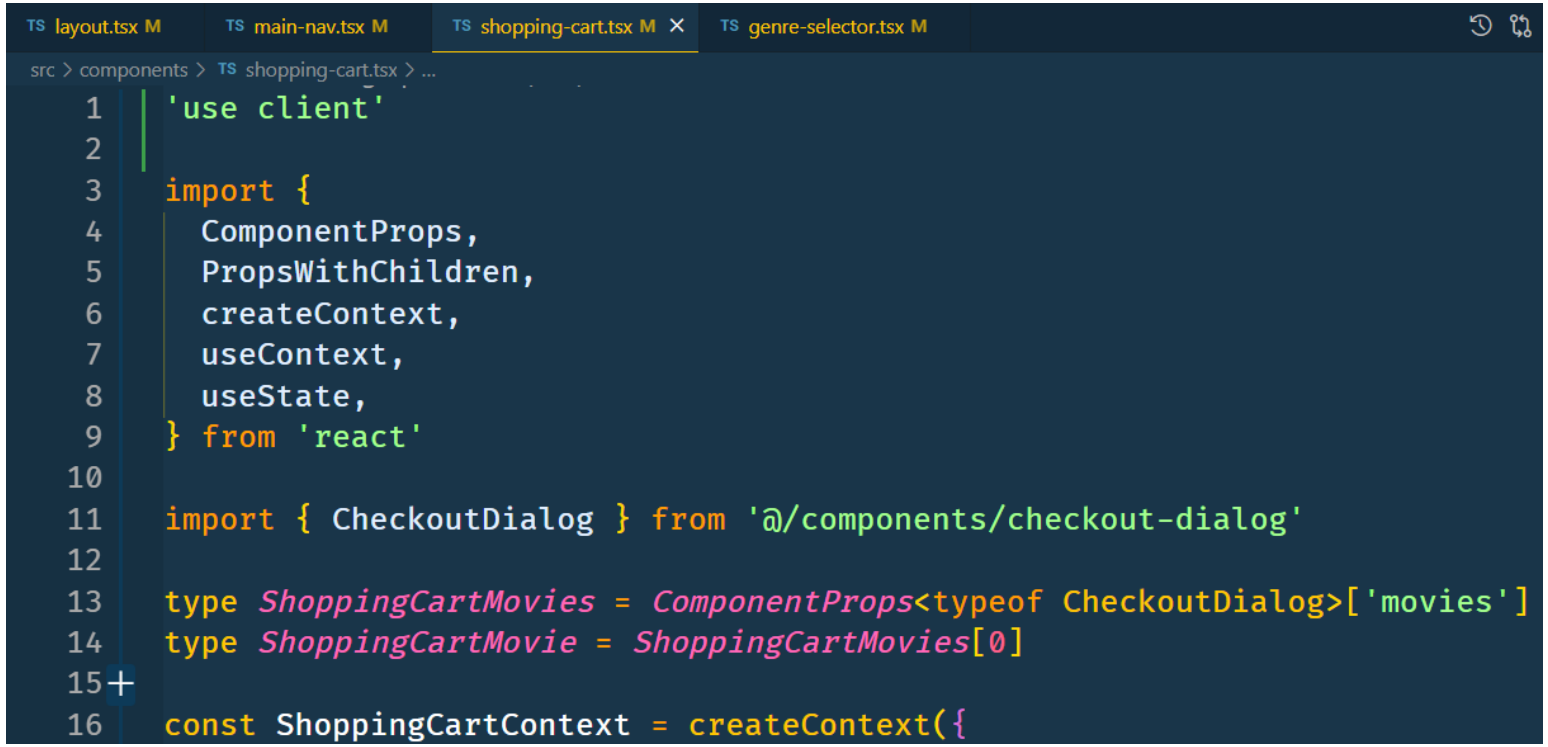
main-nav.tsx

TS layout.tsx M | TS main-nav.tsx M × | TS shopping-cart.tsx M | TS genre-selector.tsx M

src > components > TS main-nav.tsx > ...

```tsx
'use client'

import Link from 'next/link'
import { usePathname, useSearchParams } from 'next/navigation'

import { cn } from '@/lib/utils'
import { Button } from '@/components/ui/button'
import { useShoppingCart } from './shopping-cart'
import { GenreSelector } from './genre-selector'

export function MainNav() {
  const { itemCount, checkout } = useShoppingCart()
  const pathname = usePathname()
  const searchParams = useSearchParams()
  const hasGenreParam = searchParams?.has('genre')
```

shopping-cart.tsx



genre-selector.tsx

TS layout.tsx M | TS main-nav.tsx M | TS shopping-cart.tsx M | TS genre-selector.tsx M

src > components > TS genre-selector.tsx > ...

```tsx
'use client'

import { useEffect, useState } from 'react'
import { Check, ChevronsUpDown } from 'lucide-react'
import { useRouter, useSearchParams } from 'next/navigation'
import { Genre } from '@prisma/client'

import { cn } from '@/lib/utils'
import { Button } from '@/components/ui/button'
import { Command, CommandGroup, CommandItem } from '@/components/ui/command'
import {
  Popover,
  PopoverContent,
  PopoverTrigger,
} from '@/components/ui/popover'

export function GenreSelector() {
```

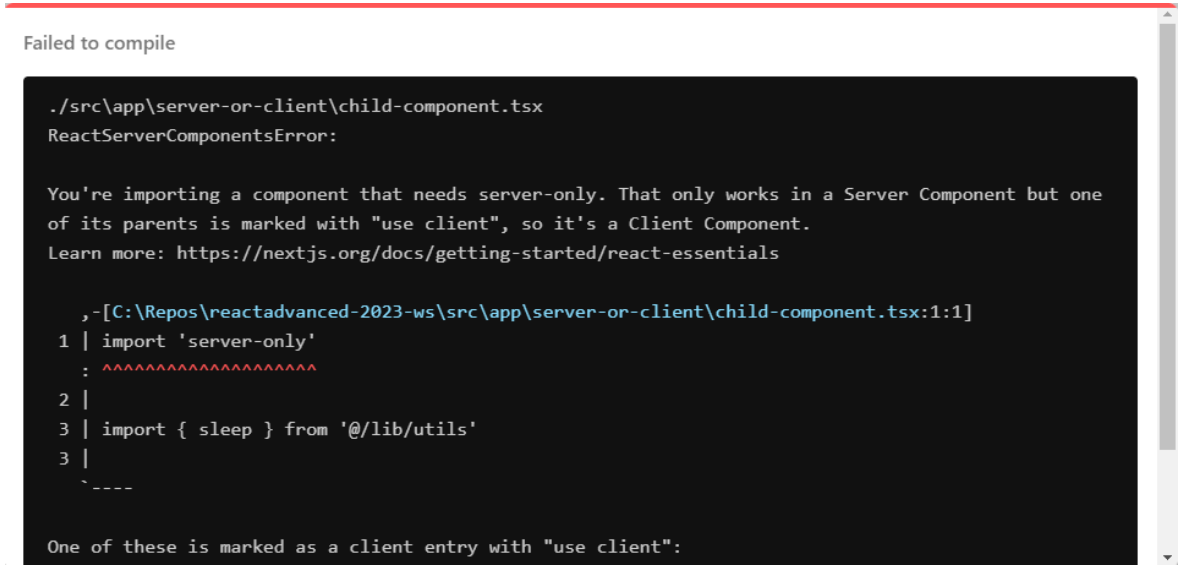
What is a server component?

What is a server component?

- What is a server component and what is not?
  - **Client components are marked with 'use client'**
- But **not all other components are server components**
  - With a component without 'use client' it depends on their parents
- If a component is a client component
  - Then **all components it renders are also client components**
- ☞ There is ***no 'use server'*** for server components ☜
  - The ***'use server'*** directive exists but is used for Server Actions
  - But there is a ***server-only*** NPM package

server-only

- Import the **server-only** NPM package
  - With components that must run on the server



Using an RSC as a child of a client component

- **A client component can have a server component as a child**
  - As long as it doesn't render it
- **Render the child server component** from another server component
  - 💡 And pass it as a children prop into the client component 💡

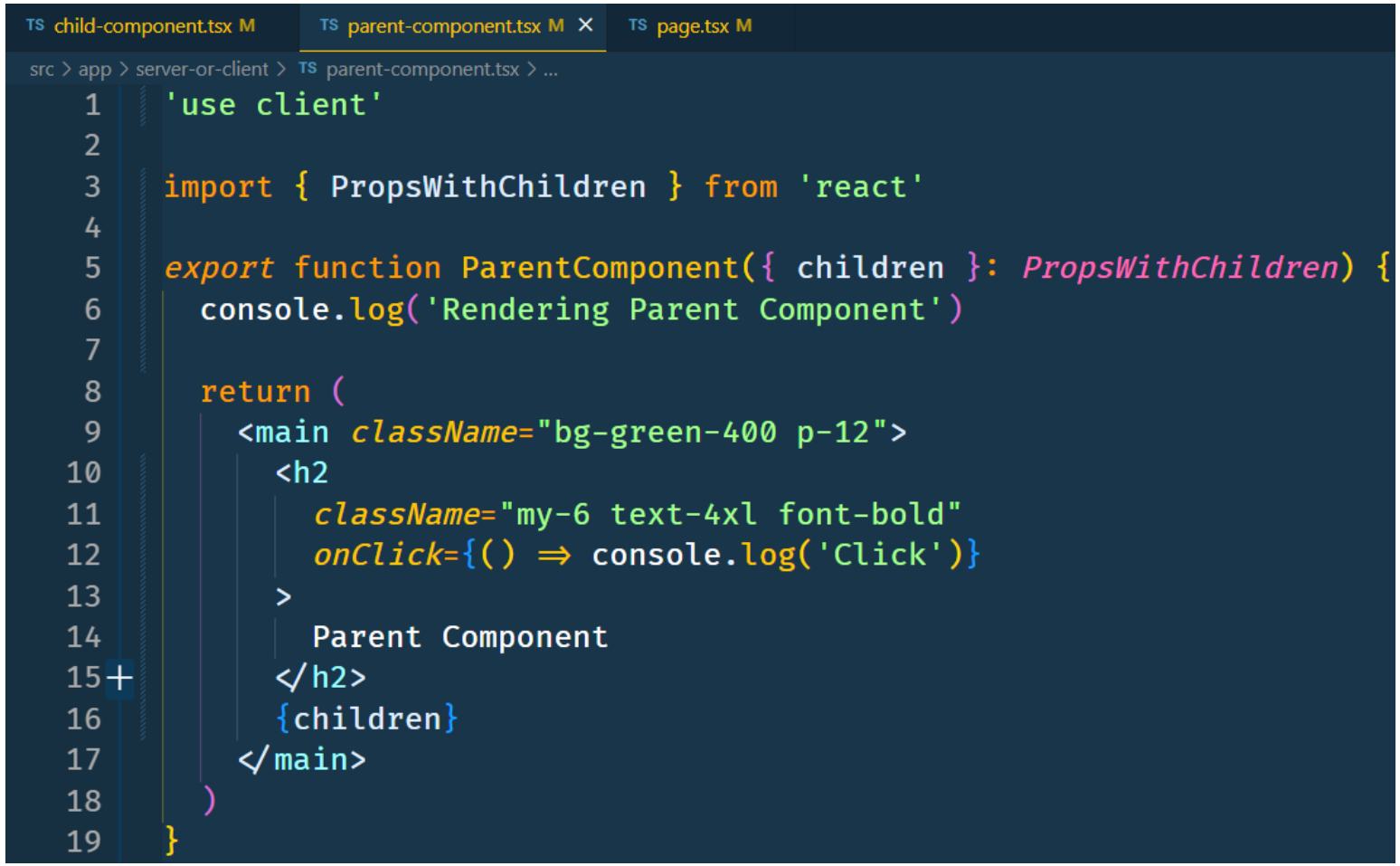
child-component.tsx

TS child-component.tsx M | TS parent-component.tsx M | TS page.tsx M

src > app > server-or-client > TS child-component.tsx > ...

```tsx
import 'server-only'

import { sleep } from '@/lib/utils'

export async function ChildComponent() {
  console.log('Rendering Child Component')

  await sleep(100)

  return (
    <main className="bg-red-400 p-12">
      <h2 className="my-6 text-4xl font-bold">Child Component</h2>
    </main>
  )
}
```

parent-component.tsx



server-or-client /page.tsx

TS child-component.tsx M | TS parent-component.tsx M | TS page.tsx M

src > app > server-or-client > TS page.tsx > ...

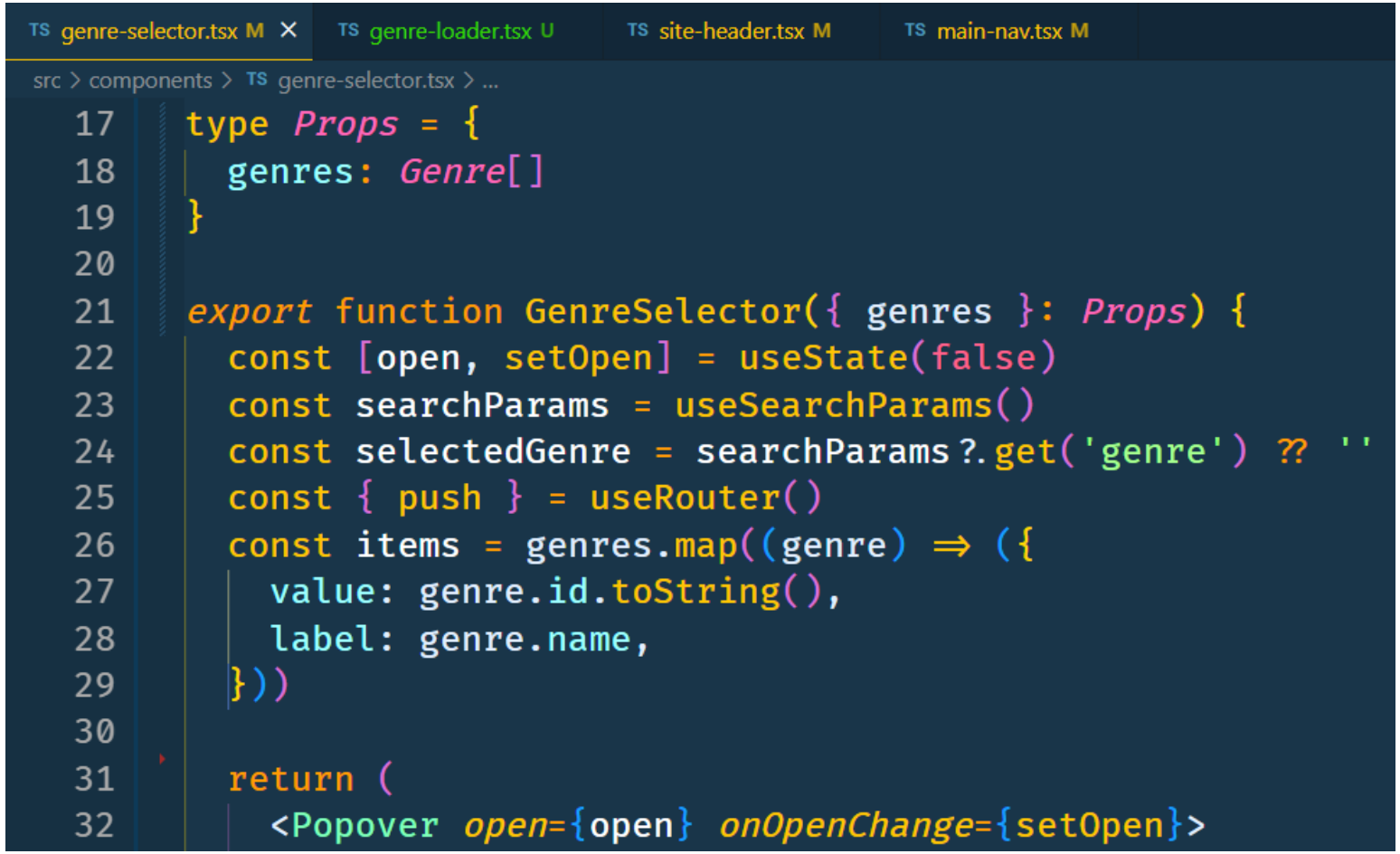
```tsx
import { ParentComponent } from './parent-component'
import { ChildComponent } from './child-component'

export default function ServerOrClient() {
  console.log('Rendering Page')

  return (
    <main className="bg-blue-400 p-12">
      <h1 className="my-6 text-4xl font-bold">
        Render on the server or client
      </h1>
      <ParentComponent>
        <ChildComponent />
      </ParentComponent>
    </main>
  )
}
```

Loading the genres on the server

Loading the genres on the server

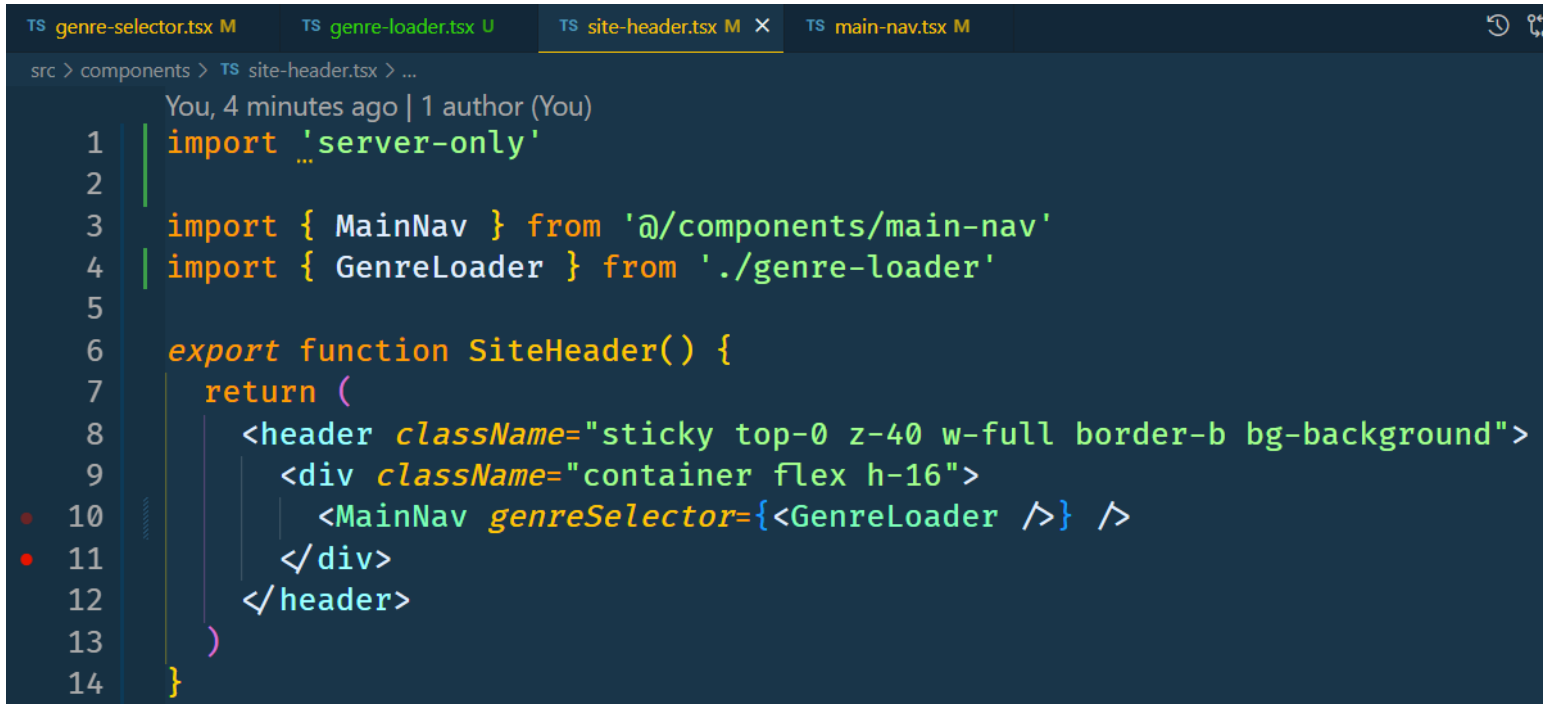
- **Splitting the GenreSelector** in a client and a server component
  - **Client component for interactivity**
  - **Server component for data loading**
- The **MainNav component** still needs to be a client component
  - The **GenreSelector/Loader can be injected as a prop**

genre-selector.tsx

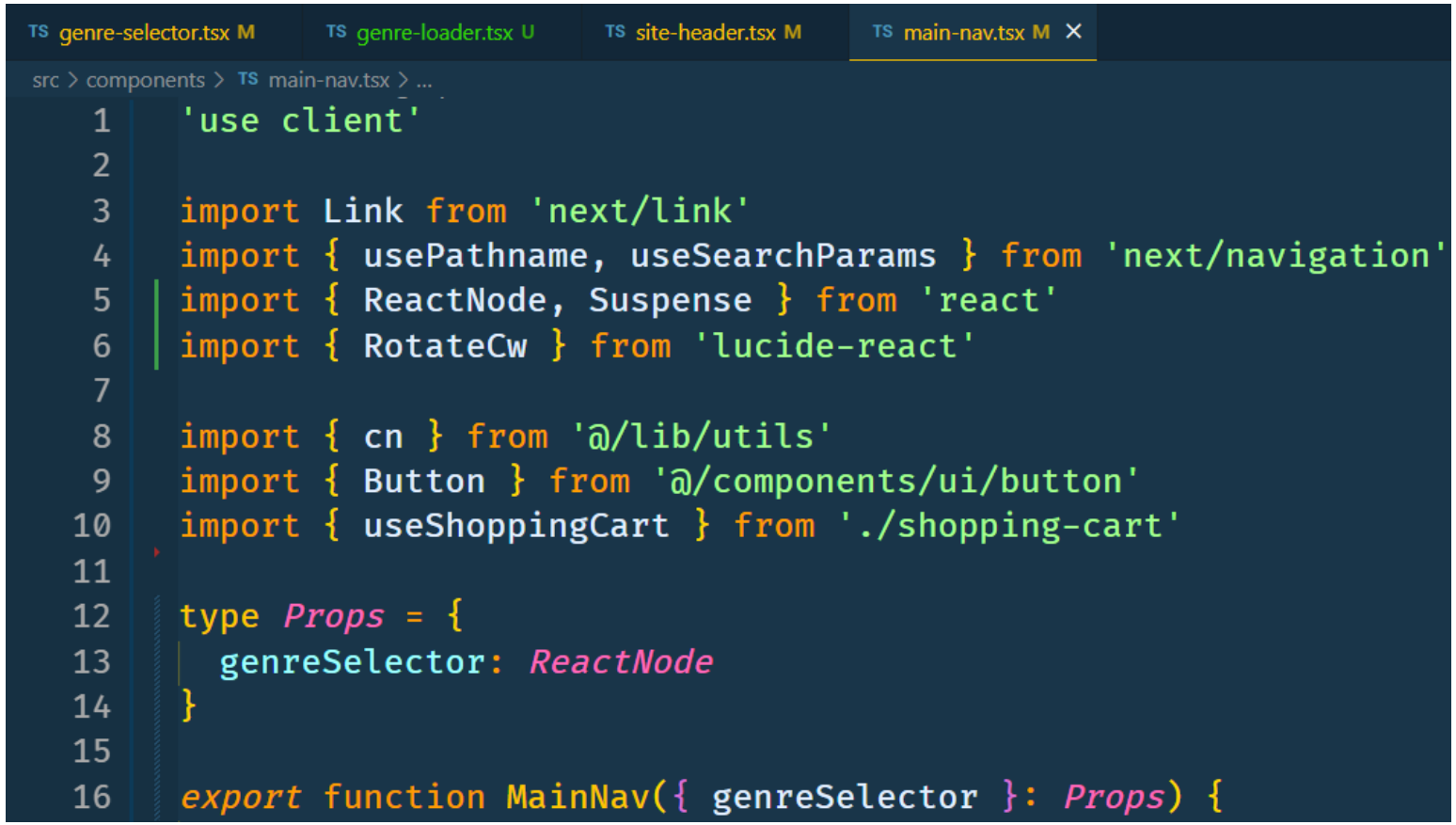


genre-loader.tsx

site-header.tsx



main-nav.tsx



main-nav.tsx

TS genre-selector.tsx M | TS genre-loader.tsx U | TS site-header.tsx M | TS main-nav.tsx M

src > components > TS main-nav.tsx > ...

```tsx
47              <Suspense
48                fallback={
49                  <RotateCw className="animate-spin text-foreground/60" size={20} />
50                }
51              >
52                {genreSelector}
53              </Suspense>
```

Calling Server Actions

Calling Server Actions

- **React Server Actions** are functions that we can **call on the client**
  - But then **execute on the server**
- Add the ***'use server'*** annotation
  - Can be at the top of a file or a single function
  - Not related to server components
- Can be passed as **the action of a client side <form />**
  - The forms data is passed as a **FormData** parameter
  - Even works if JavaScript is disabled ☺
- Can also be **called as a normal asynchronous function**
  - The network request is handled for you
- Requires the Next.js **experimental *serverActions*** flag to be set

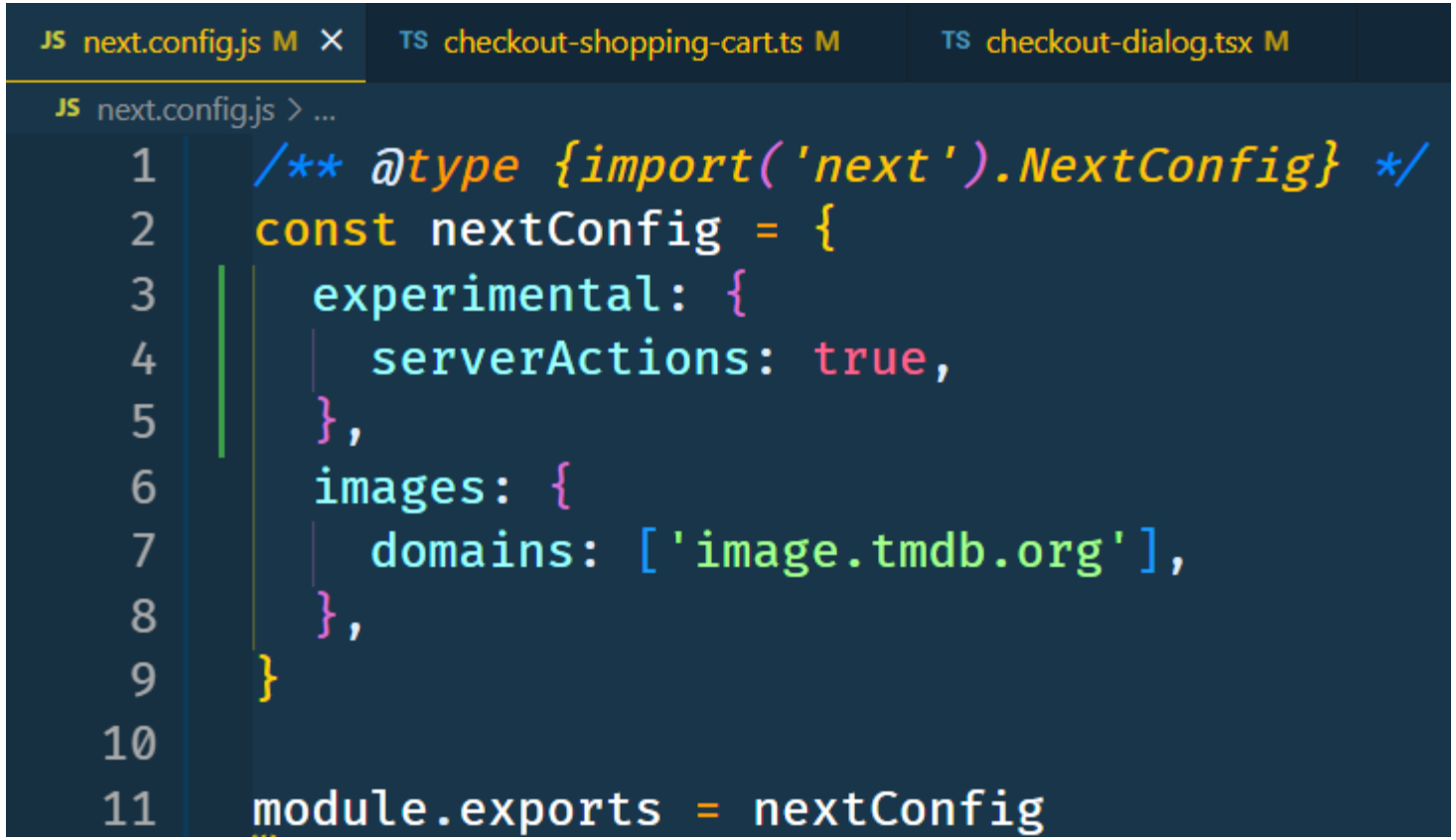
Submitting a form

page.tsx M

src > app > page.tsx > ...

```tsx
export default function AddUserPage() {
  const handleSubmit = async (formData: FormData) => {
    'use server';
    console.log('handleSubmit', formData);
  };

  return (
    <div className="m-auto my-10 w-1/3">
      <form action={handleSubmit} className="space-y-4">
        <h1>Add a new user</h1>
        <div className="grid grid-cols-2 gap-4">
          <div className="space-y-2">
            <Label htmlFor="first-name">First name</Label>
            <Input name="first-name" id="first-name" />
          </div>
          <div className="space-y-2">
            <Label htmlFor="last-name">Last name</Label>
            <Input name="last-name" id="last-name" />
          </div>
        </div>
```

next.config.js



checkout-shopping-cart.ts

JS next.config.js M | TS checkout-shopping-cart.ts M × | TS checkout-dialog.tsx M

src > server > TS checkout-shopping-cart.ts > ...

```ts
'use server'

import { Movie } from '@prisma/client'

type ShoppingCartMovie = Pick<Movie, 'id' | 'title'>

type Cart = {
  account: string
  customerName: string
  movies: ShoppingCartMovie[]
}

export async function checkoutShoppingCart({
  account,
  customerName,
  movies,
}: Cart) {
```

checkout-dialog.tsx

JS next.config.js M | TS checkout-shopping-cart.ts M | TS checkout-dialog.tsx M

src > components > TS checkout-dialog.tsx > ...

```tsx
53    const onSubmit = async (data: CheckoutForm) => {
54      try {
55        await checkoutShoppingCart({
56          account: data.account,
57          customerName: data.name,
58          movies,
59        })
60        toast({
61          title: 'Success',
62          description: 'Checkout completed',
63        })
```

Moving from pages to RSC with Next.js

Moving from pages to RSC

- React Server Components require the **App Router** in Next.js
  - The App Router has many other benefits
- Each route is **a folder with a page.tsx**
  - For example app/movies/page.tsx
- Other files in the folder are **not reachable** as routes
  - Co-locate route specific components to the same folder if you want
- **Dynamic routes** are done with **square bracket in the folder** name
  - For example app/movies/[movieId]/page.tsx

Moving from pages to RSC

- **Replace** all instances of **next/router** with **next/navigation**

Failed to compile

```
./src\app\genres\page.tsx
ReactServerComponentsError:

You have a Server Component that imports next/router. Use next/navigation instead.
Learn more: https://nextjs.org/docs/app/api-reference/functions/use-router

Import trace:
  ./src\app\genres\page.tsx
```

This error occurred during the build process and can only be dismissed by fixing the error.

genres /page.tsx

genres-list.tsx

page.tsx ...\genres U | page.tsx ...\[id] U | genres-list.tsx U

src > app > genres > genres-list.tsx > ...

```tsx
'use client'

import { Button } from '@/components/ui/button'
import { Genre } from '@prisma/client'
import { useRouter } from 'next/navigation'
import { FC } from 'react'

type Props = {
  genres: Genre[]
}

export const GenresList: FC<Props> = ({ genres }) => {
  const router = useRouter()

  return (
    <main className="flex-1 space-y-4 p-8 pt-6">
      <h2 className="text-3xl font-bold tracking-tight">Genres</h2>
```

genres/[id] /page.tsx

TS page.tsx ...\genres U | TS page.tsx ...\[id] U | TS genres-list.tsx U

src > app > genres > [id] > TS page.tsx > ...

```tsx
import 'server-only'

import { prisma } from '@/lib/db'

type Props = {
  params: {
    id: string
  }
}

async function GenrePage({ params: { id } }: Props) {
  const genre = await prisma.genre.findFirstOrThrow({
    where: { id: Number(id) },
  })

  return (
    <main className="flex-1 space-y-4 p-8 pt-6">
      <h2 className="text-3xl font-bold tracking-tight">Genre</h2>
      <div className="flex w-72 flex-col gap-6 px-6">
        <div>ID: {genre.id}</div>
        <div>Name: {genre.name}</div>
      </div>
    </main>
  )
}

export default GenrePage
```

Recommendations with React Server Components

Recommendations

- **Start with Shared** components
  - Can run on the server or client as needed
  - Will default to act as Server Components
- Switch to **Server only components if needed**
  - When you need to use server side capabilities
- Only use **Client only components when absolutely needed**
  - Local state or side effects
  - Interactivity
  - Required browser API's
- Learn all about the **new capabilities of Next.js**
  - App Router
  - Caching

Conclusion

- React Server Components are a **great new addition to React**
  - Helps with keeping the client more responsive
  - Makes the application architecture easier
- **Use Next.js and the App Router**
  - Because you need a server
- React **Client Components**
  - Are components with state and interactivity and require 'use client'
- Control caching of React **Server Components**
  - Because Next.js is quite aggressive about caching
- **React Server Components are streamed**
  - And use Suspense boundaries until they are done
- **Server Actions** are a great way to call back into the server
  - They also update the invalidated server components on the client

Thank you for joining

Share your thoughts