REACT ADVANCED LONDON

2 days
In-person & Remote

50+ speakers
presenting on advanced topics

800+ luckies
in London

5K devs
globally remotely

October
25 & 28, 2024

Reserve a spot →

Attend remotely →

Mastering React Server Components and Server Actions in React 19

Maurice de Beijer
@mauricedb

- Maurice de Beijer
- The Problem Solver
- Freelance developer/instructor
- Currently at https://someday.com/
- Twitter: @mauricedb
- Web: https://www.theproblemsolver.dev/
- 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**

Type it out by hand?

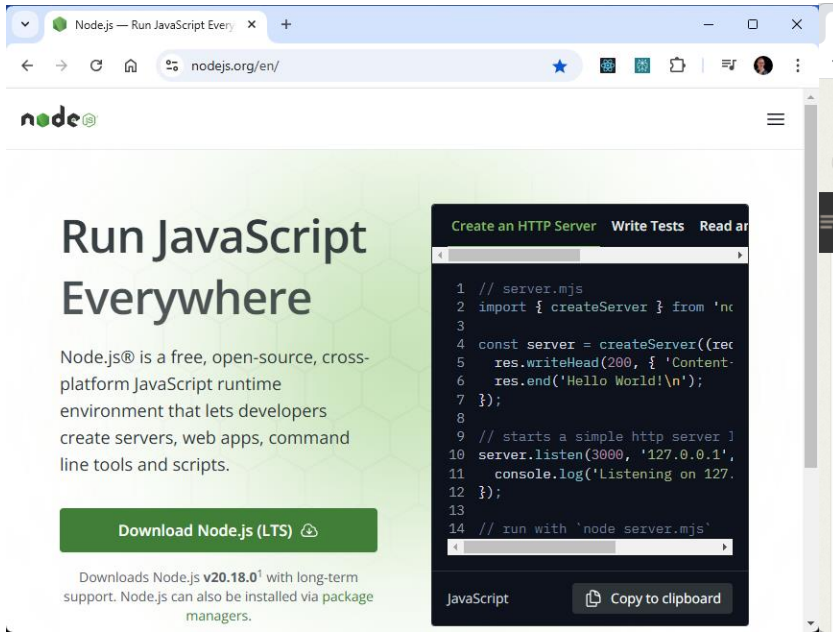
"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"

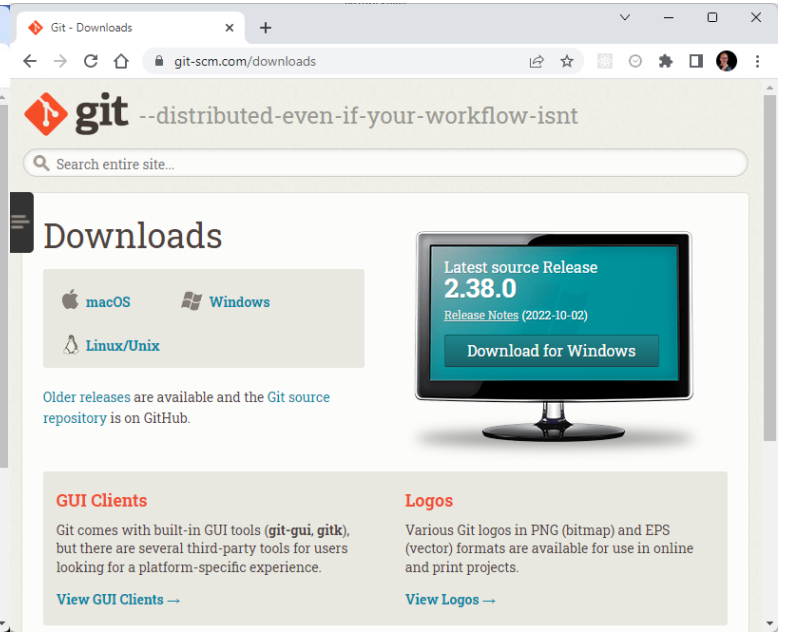
Prerequisites

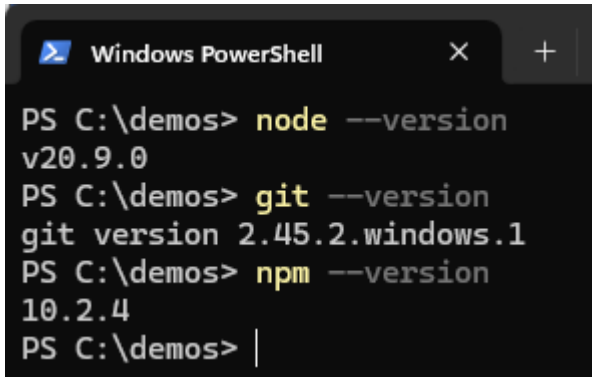
Install Node & NPM

Install the GitHub repository

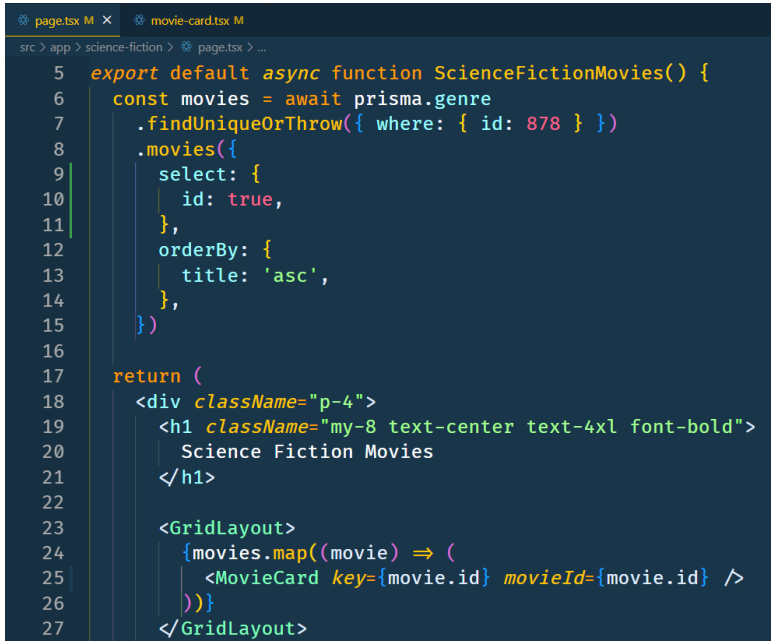
Install Node.js & NPM







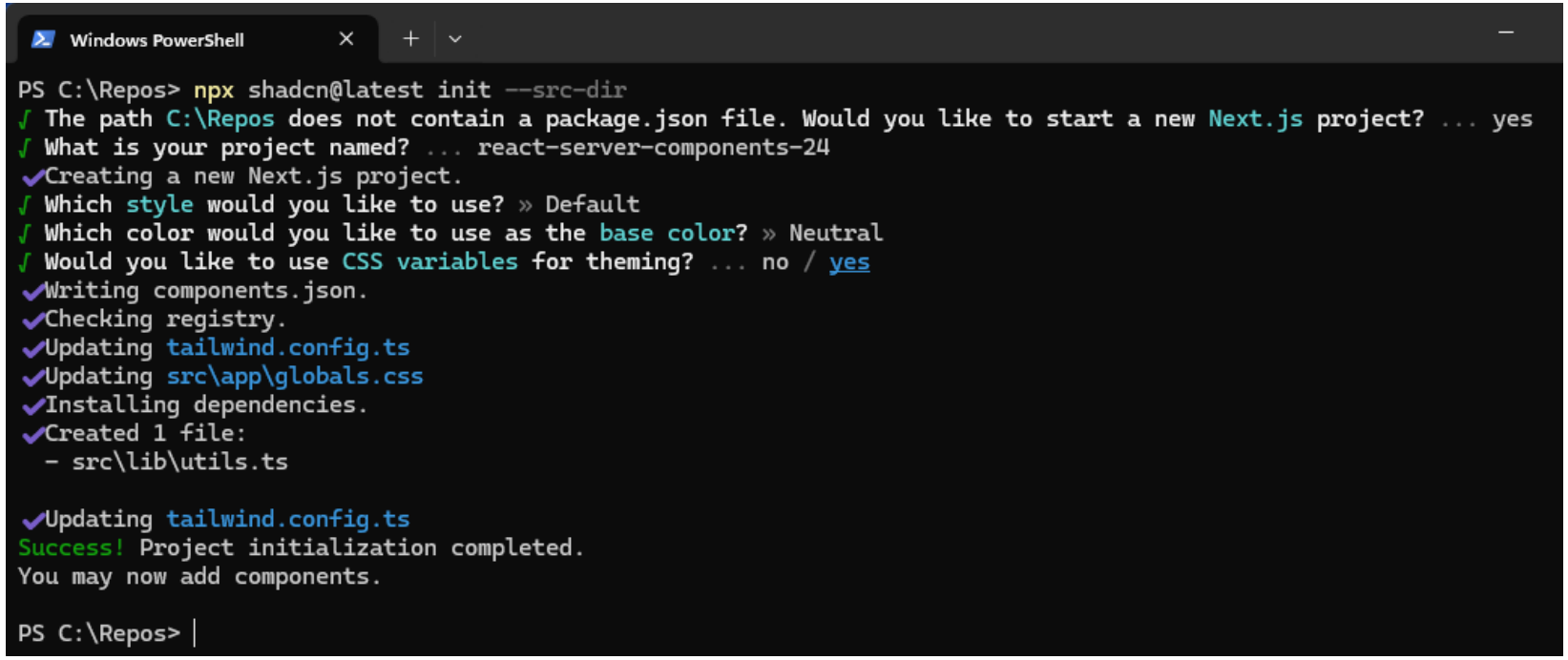
Following Along



- Repo: https://github.com/mauricedb/react-server-components-24
- Slides: https://www.theproblemsolver.dev/docs/react-advanced-2024.pdf

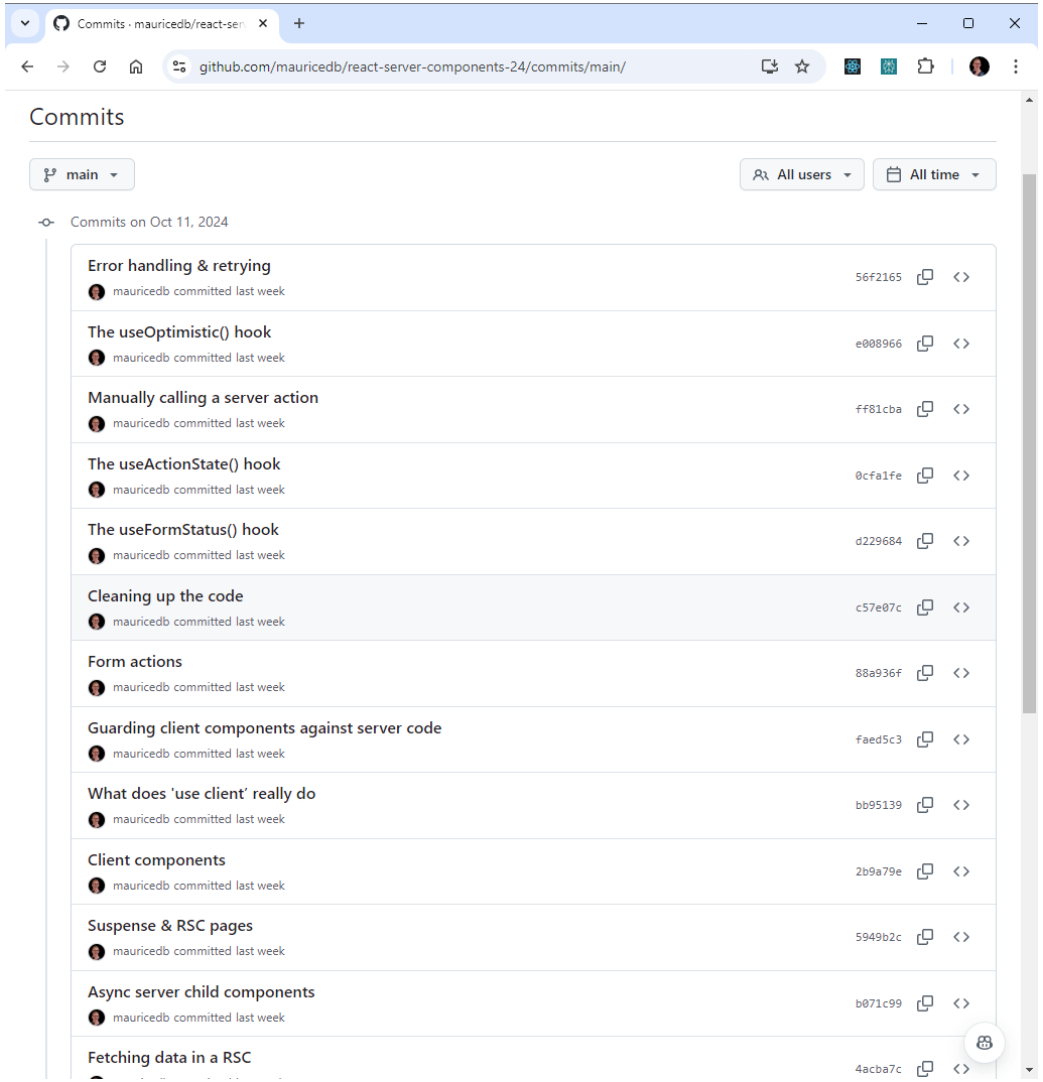
Create a new Next.js app with shadcn/ui

- npx shadcn@latest init --src-dir

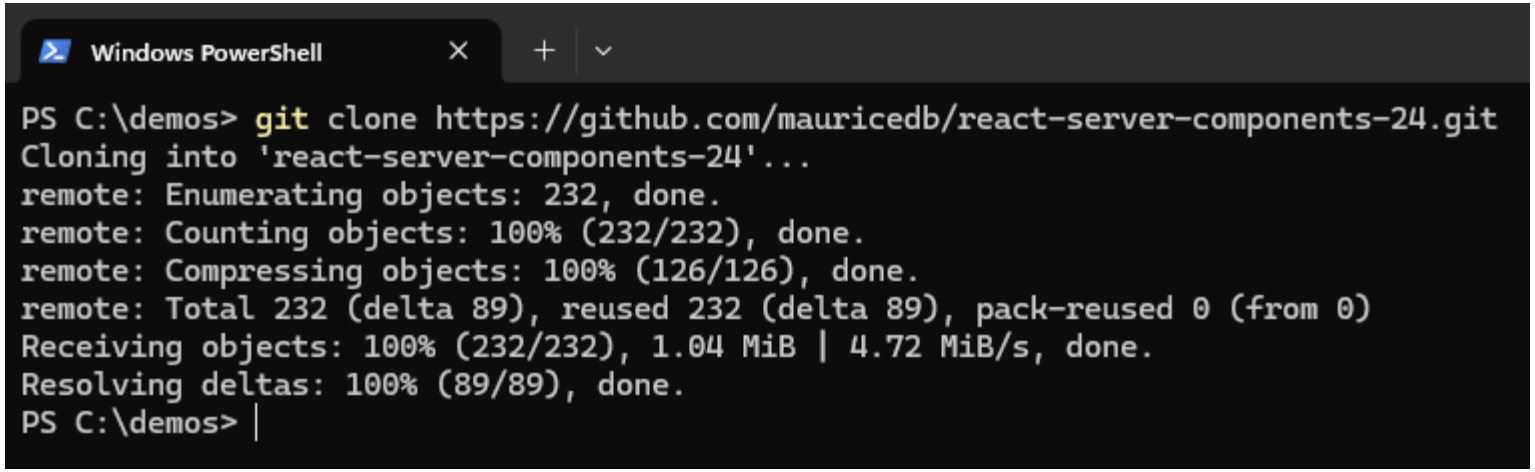


The changes


MAKE IT SO
memegenerator.net

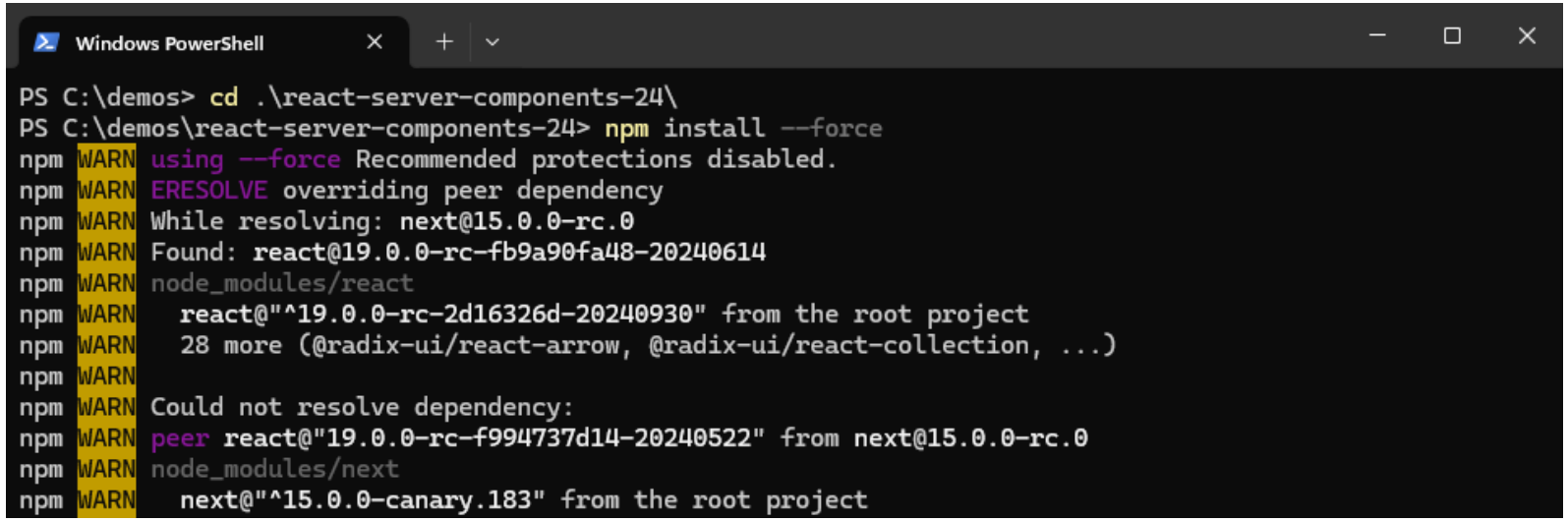

Commits · mauricedb/react-ser
github.com/mauricedb/react-server-components-24/commits/main/
Commits
main
All users
All time
Commits on Oct 11, 2024
Error handling & retrying
mauricedb committed last week
56f2165
The useOptimistic() hook
mauricedb committed last week
e008966
Manually calling a server action
mauricedb committed last week
ff81cba
The useActionState() hook
mauricedb committed last week
0cfa1fe
The useFormStatus() hook
mauricedb committed last week
d229684
Cleaning up the code
mauricedb committed last week
c57e07c
Form actions
mauricedb committed last week
88a936f
Guarding client components against server code
mauricedb committed last week
faed5c3
What does 'use client' really do
mauricedb committed last week
bb95139
Client components
mauricedb committed last week
2b9a79e
Suspense & RSC pages
mauricedb committed last week
5949b2c
Async server child components
mauricedb committed last week
b071c99
Fetching data in a RSC
4acba7c

Clone the GitHub Repository

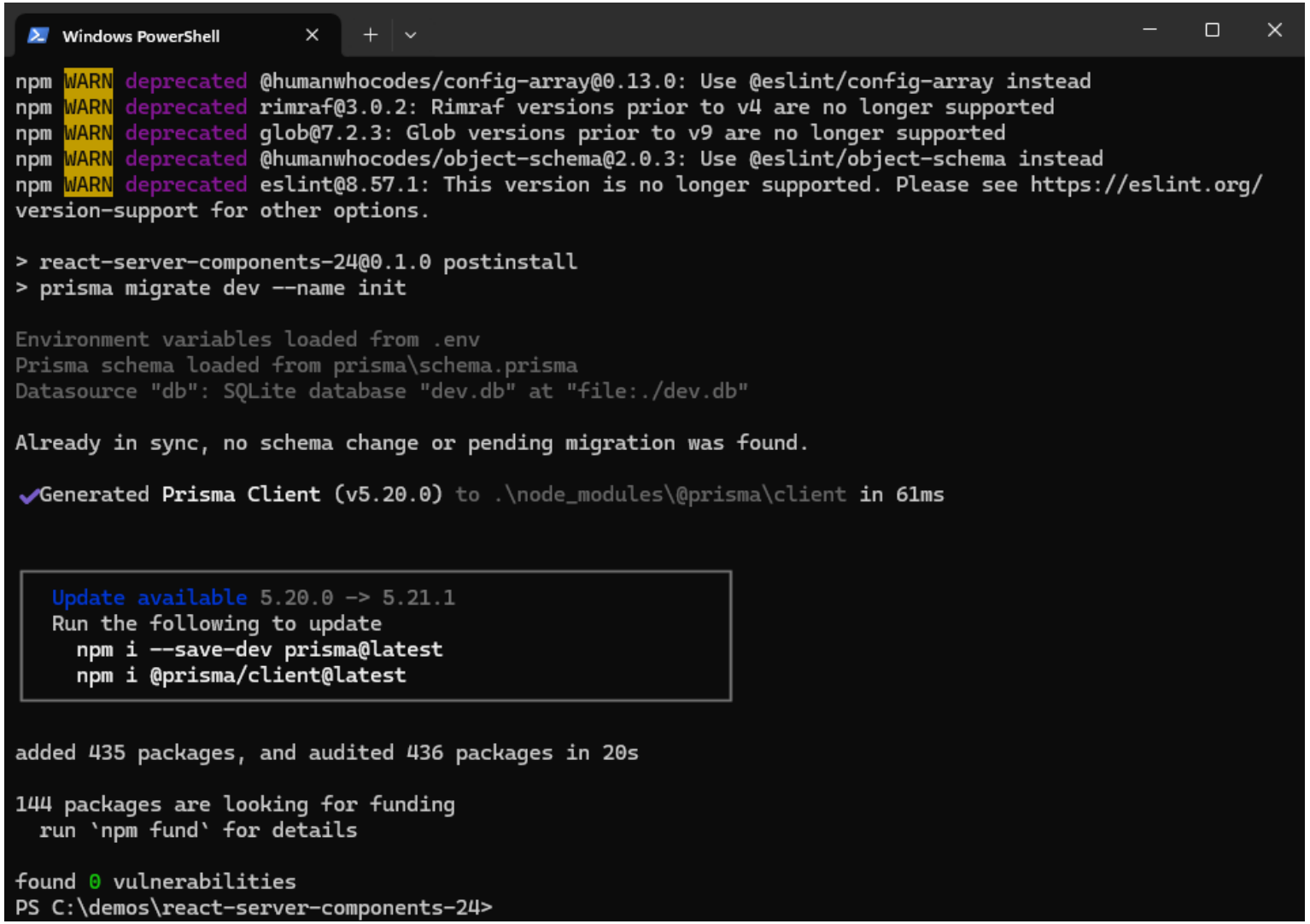


Install NPM Packages

- ☞ Use: npm install –force ☜



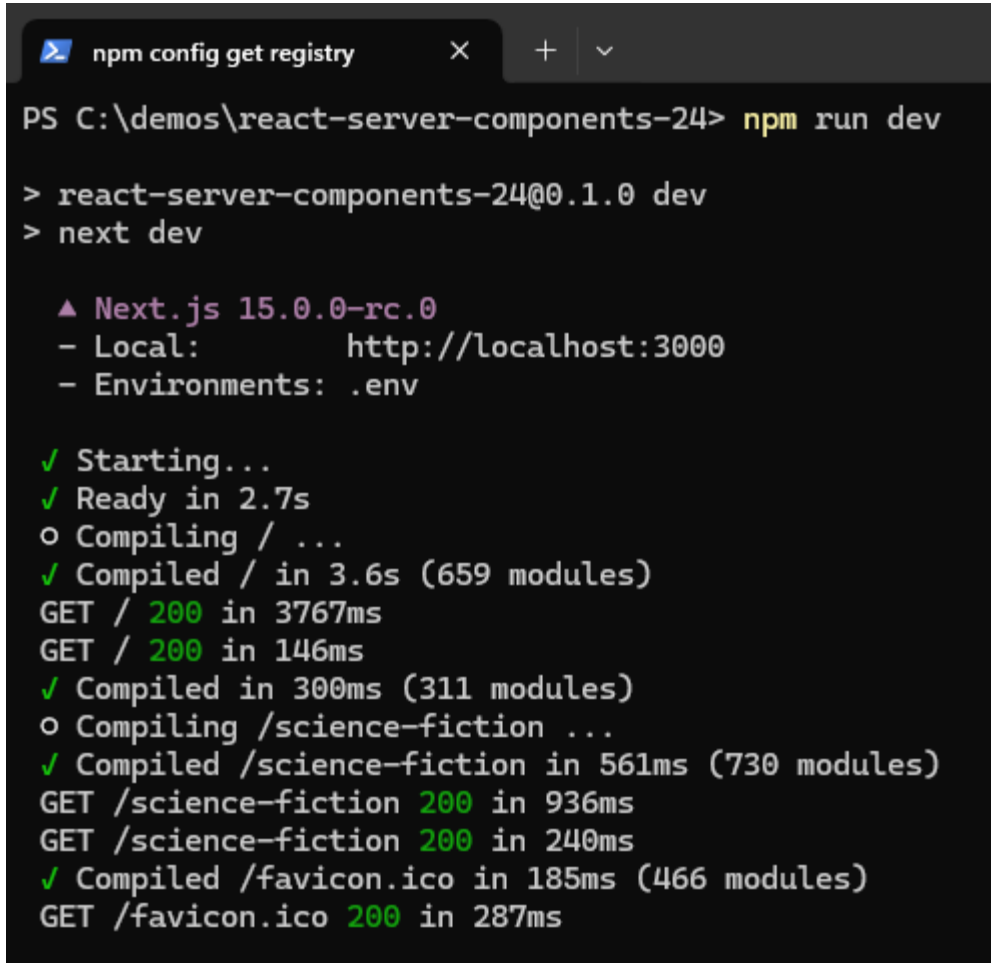
Install NPM Packages



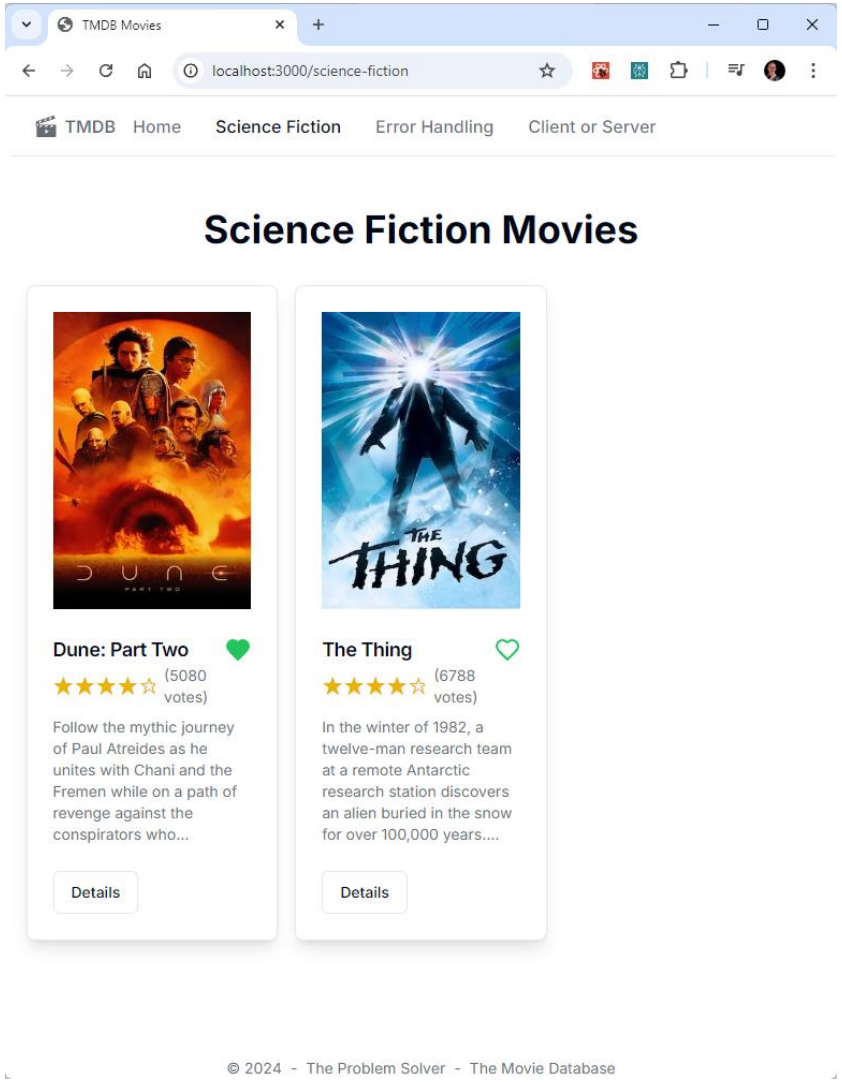
Start branch

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

Start the application



The application

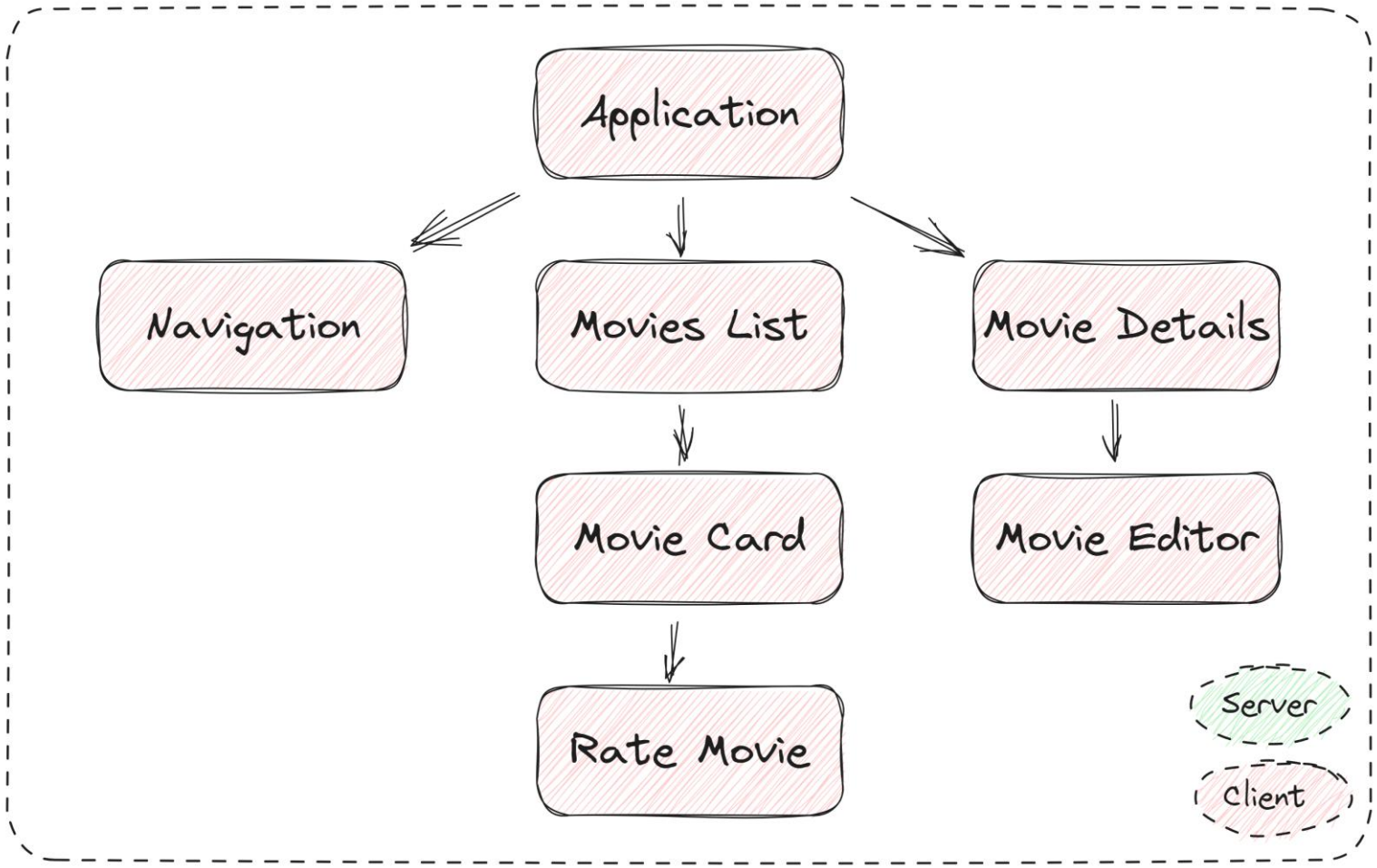


What are React Server Components?

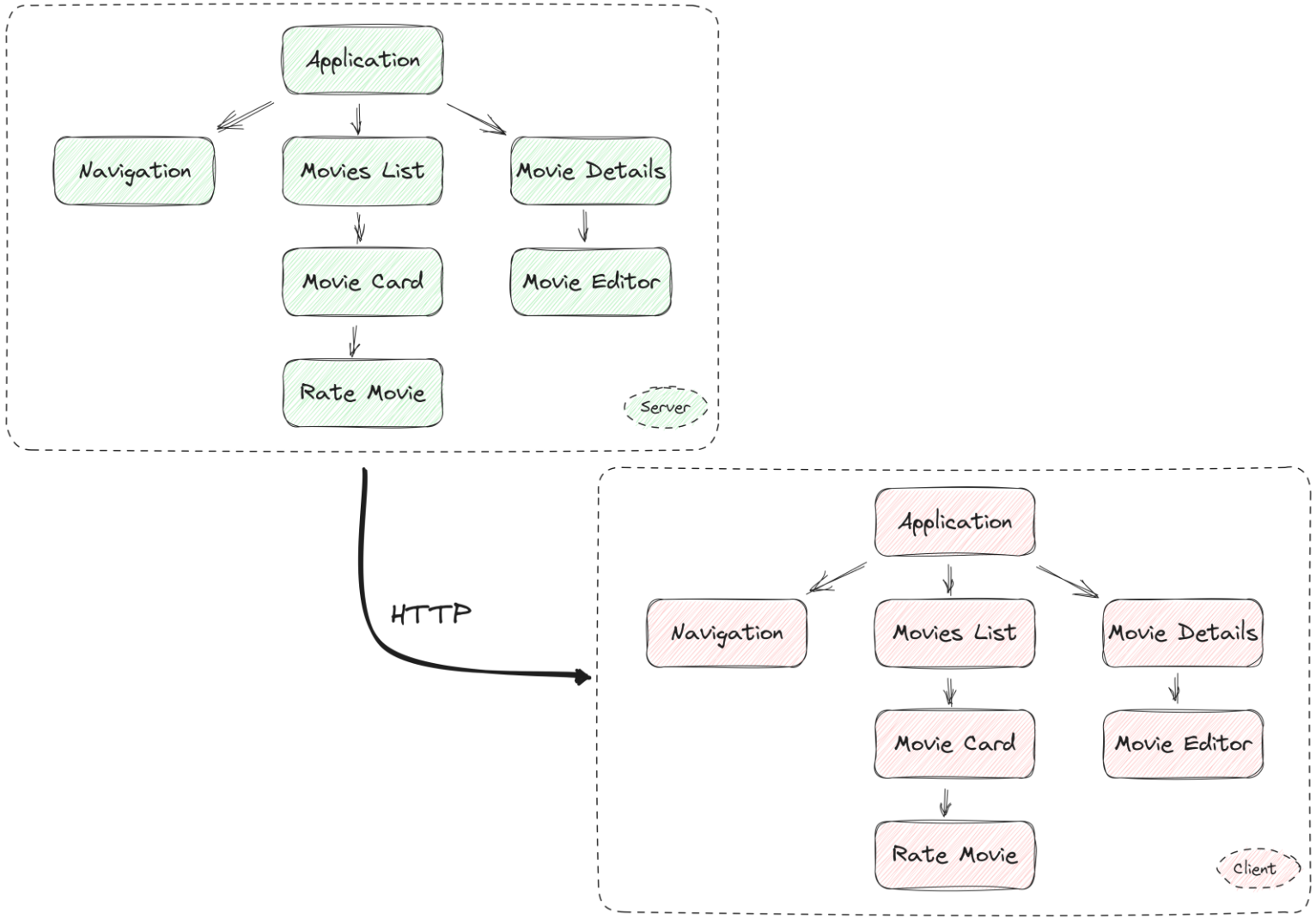
React Server Components

- 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** ☺

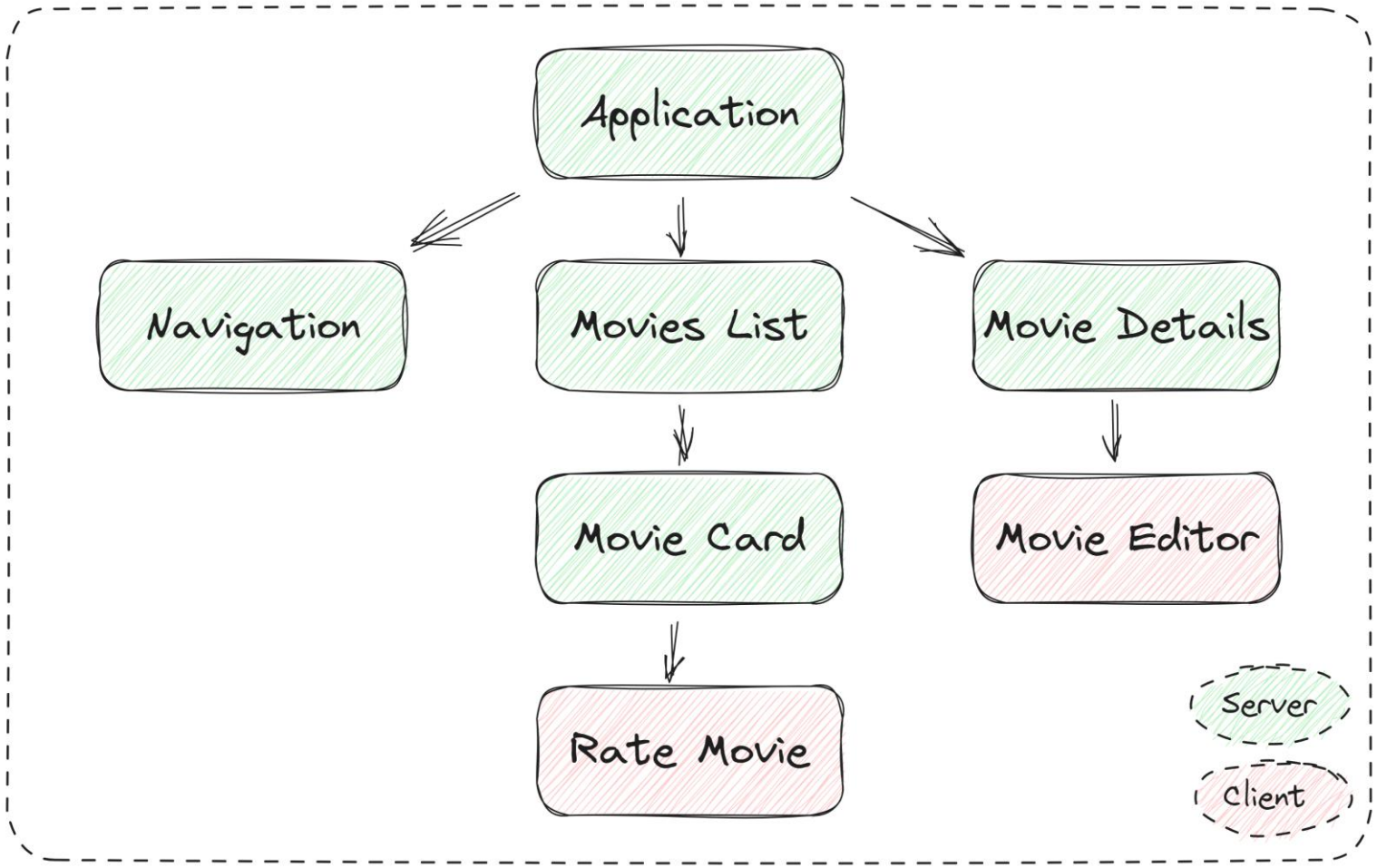
Before RSC (no SSR)



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 Component

page.tsx M

src > app > science-fiction > page.tsx > ...

```tsx
import { GridLayout } from '@/components/grid-layout'
import { MovieCard } from '@/components/movie-card'
import { prisma } from '@/lib/prisma'

export default async function AllMoviesPage() {
  const movies = await prisma.movie.findMany({
    orderBy: {
      title: 'asc',
    },
  })

  return (
    <div className="p-4">
      <h1 className="my-8 text-center text-4xl font-bold">
        Science Fiction Movies
      </h1>

      <GridLayout>
        {movies.map((movie) => (
          <MovieCard key={movie.id} movieId={movie.id} />
        ))}
      </GridLayout>
    </div>
  )
}
```

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**
  - ☞ Remix 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

Fetching data in a RSC

Fetching data in a RSC

- React Server Components an **execute normal Node.js code**
  - Read/write files on disk
  - Do fetch requests to other servers
  - Execute CRUD in a database
- RSC's **can be asynchronous** where needed
  - Just **await** whatever action needs to be done

src\app\science-fiction\page.tsx

page.tsx M

src > app > science-fiction > page.tsx > ...

```tsx
import { GridLayout } from '@/components/grid-layout'
import { MovieCard } from '@/components/movie-card'
import { prisma } from '@/lib/prisma'

export default async function ScienceFictionMovies() {
  const movies = await prisma.genre
    .findUniqueOrThrow({ where: { id: 878 } })
    .movies({
      orderBy: {
        title: 'asc',
      },
    })

  return (
    <div className="p-4">
```

Async server child components

Child RSC components

- A RSC component can render other RSC child components
  - They can execute the same server based code
  - Including async/await where needed

src\app\science-fiction\page.tsx

page.tsx M | movie-card.tsx M

src > app > science-fiction > page.tsx > ...

```tsx
export default async function ScienceFictionMovies() {
  const movies = await prisma.genre
    .findUniqueOrThrow({ where: { id: 878 } })
    .movies({
      select: {
        id: true,
      },
      orderBy: {
        title: 'asc',
      },
    })

  return (
    <div className="p-4">
      <h1 className="my-8 text-center text-4xl font-bold">
        Science Fiction Movies
      </h1>

      <GridLayout>
        {movies.map((movie) => (
          <MovieCard key={movie.id} movieId={movie.id} />
        ))}
      </GridLayout>
```

src\components\movie-card.tsx

page.tsx M | movie-card.tsx M

src > components > movie-card.tsx > ...

```tsx
import { prisma } from '@/lib/prisma'

type Props = {
  movieId: number
} & React.ComponentProps<typeof Card>

export async function MovieCard({ movieId, ...props }: Props) {
  // await sleep(Math.random() * 5_000);
  const movie = await prisma.movie.findUniqueOrThrow({
    where: { id: movieId },
  })

  return (
    <Card className="flex h-full flex-col shadow-lg" {...props}>
```

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** next to the page.tsx
  - They can be nested and the closest loading component will be used

src\app\science-fiction\page.tsx

page.tsx M | movie-card.tsx M

src > app > science-fiction > page.tsx > ...

```tsx
export default async function ScienceFictionMovies() {
  return (
    <div className="p-4">
      <h1 className="my-8 text-center text-4xl font-bold">
        Science Fiction Movies
      </h1>

      <GridLayout>
        <Suspense fallback={<div>Loading ... </div>}>
          {movies.map((movie) => (
            <MovieCard key={movie.id} movieId={movie.id} />
          ))}
        </Suspense>
      </GridLayout>
    </div>
```

src\app\science-fiction\loading.tsx

page.tsx M | loading.tsx U | movie-card.tsx M

src > app > science-fiction > loading.tsx > ...

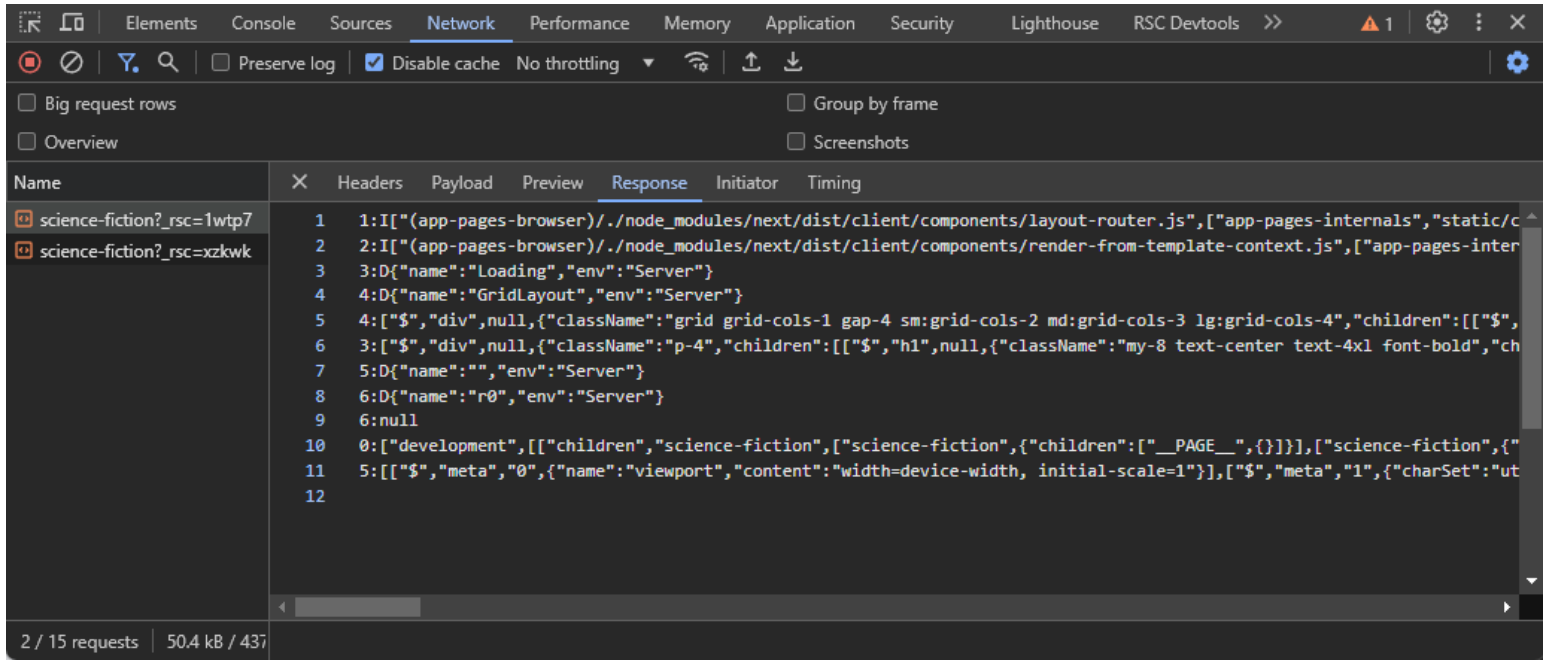
```tsx
export default function Loading() {
  return (
    <div className="p-4">
      <h1 className="my-8 text-center text-4xl font-bold">
        Science Fiction Movies
      </h1>

      <GridLayout>
        {Array.from({ length: 12 }).map((_, index) => (
          <div
            key={index}
            className="h-96 animate-pulse rounded-lg bg-gray-200 dark:bg-gray-700"
          />
        ))}
      </GridLayout>
    </div>
  )
}
```

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



Client components

Client components

- **Client components are required in a number of scenarios**
  - With interactive UI elements like elements with a click handler
  - When using browser API's like localStorage
  - When using React hooks like useState(), useEffect() etc.
- Add the **`use client`** directive
  - Makes a component a client component
- Client components **render in the browser**
  - Can't be asynchronous (for now)
- **Can't access files or databases** on the local machine
  - Other than using browser API's
- With **Server Side Rendering** they can also execute on the server
  - Next.js uses SSR by default

Client Component or 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

src\components\favourite-heart.tsx

favourite-heart.tsx M

src > components > favourite-heart.tsx > ...

```tsx
'use client'

import { Heart } from 'lucide-react'

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

type Props = {···
}

export function FavouriteHeart({ favourite, movieId }: Props) {
  return (
    <Heart
      aria-label={favourite ? 'Remove from favourites' : 'Add to favourites'}
      className={cn('cursor-pointer text-green-500', {
        'fill-green-500': favourite,
      })}
      onClick={() => {
        console.log(`We want to add this movie to favourites: ${movieId}`)
      }}
    />
  )
}
```

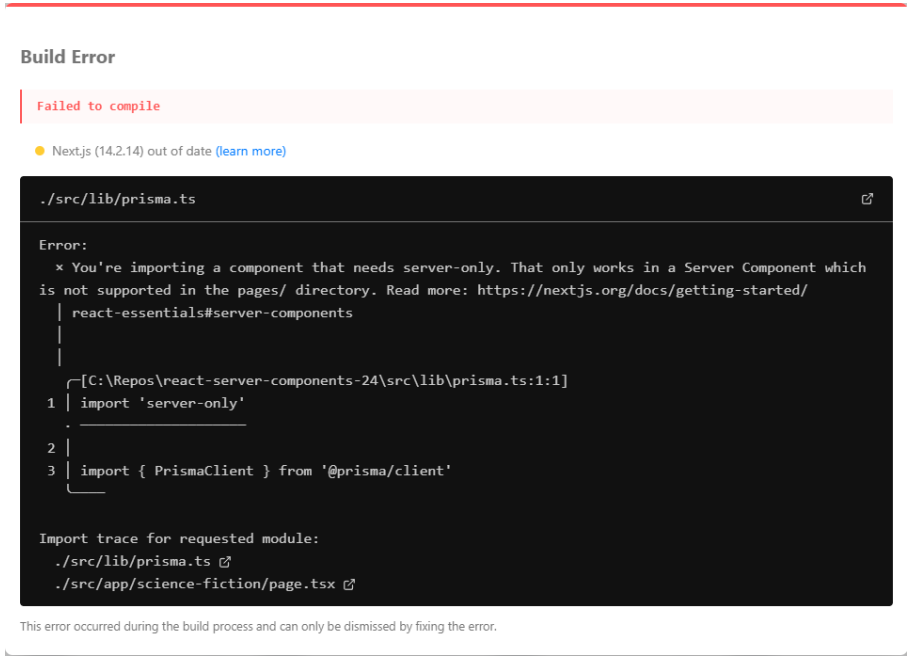
What does 'use client' really do

What is a server component?

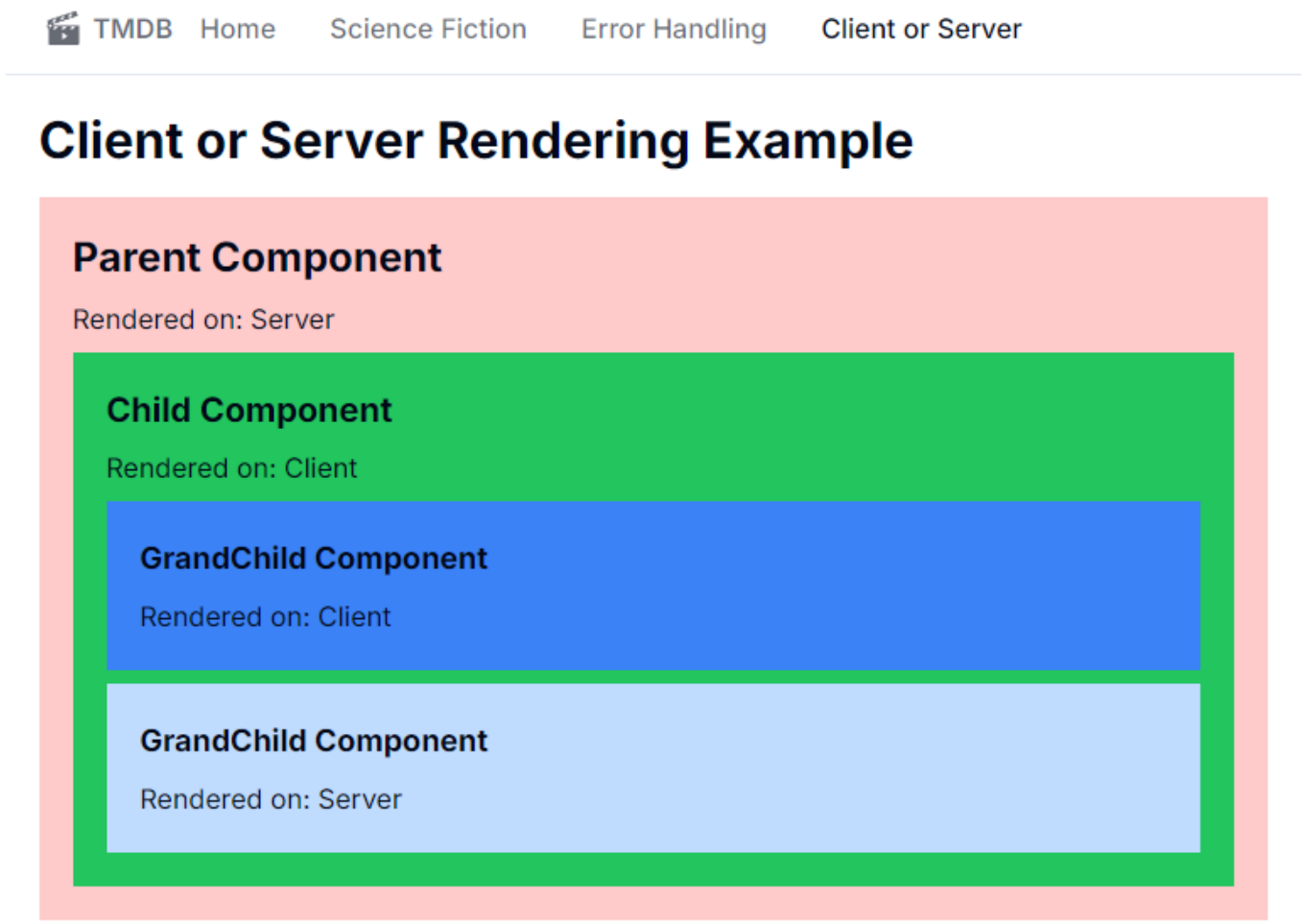
- What is a server component and what is not?
  - **Client components are marked with 'use client'**
- But **not all other components are server components**
  - With a component without 'use client' it depends on their parents
- If a component is a client component
  - Then **all components it renders 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



GrandChild is both a client and server component



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 💡

src\components\server-or-client\child.tsx

child.tsx M | parent.tsx M

src > components > server-or-client > child.tsx > ...

```tsx
'use client'

import { PropsWithChildren } from 'react'

import { cn } from '@/lib/utils'
import { GrandChild } from './grand-child'

export function Child({ children }: PropsWithChildren) {
  return (
    <div
      className={`flex flex-col gap-2 bg-green-500 p-5 ${cn({
        'bg-green-200': typeof window === 'undefined',
      })}`}
    >
      <h3 className="text-xl font-bold">Child Component</h3>
      <p>Rendered on: {typeof window === 'undefined' ? 'Server' : 'Client'}</p>
      <GrandChild />
      {children}
    </div>
  )
}
```

src\components\server-or-client\parent.tsx

child.tsx M | parent.tsx M

src > components > server-or-client > parent.tsx > ...

```tsx
import { cn } from '@/lib/utils'
import { Child } from './child'
import { GrandChild } from './grand-child'

export function Parent() {
  return (
    <div
      className={`flex flex-col gap-2 bg-red-500 p-5 ${cn({
        'bg-red-200': typeof window === 'undefined',
      })}`}
    >
      <h2 className="text-2xl font-bold">Parent Component</h2>
      <p>Rendered on: {typeof window === 'undefined' ? 'Server' : 'Client'}</p>
      <Child>
        <GrandChild />
      </Child>
    </div>
  )
}
```

Break time

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

Form actions

Form actions

- A <form> element can take a 'action' prop
  - Can point to an action function that executes on the client or server
  - More flexible that using the onSubmit
- All the <input> from the form is passed as a FormData parameter
  - Use hidden inputs to pass additional data
- 🔥 The server action function works even if JavaScript is disabled

src\app\movie\[id]\edit\page.tsx

page.tsx M | movie-editor.tsx M

src > app > movie > [id] > edit > page.tsx > ...

```tsx
export default async function MovieEditPage({ params }: Props) {
  const formAction = async (formData: FormData) => {
    'use server'
    const json = Object.fromEntries(formData.entries())
    console.log('Form submit', json)

    movie.title = formData.get('title') as string
    movie.overview = formData.get('overview') as string
    // Etc.

    await prisma.movie.update({
      where: { id: movie.id },
      data: movie,
    })
  }

  return (
    <div className="p-4">
      <MovieEditor movie={movie} formAction={formAction} />;
    </div>
  )
}
```

src\components\movie-editor.tsx

page.tsx M | movie-editor.tsx M

src > components > movie-editor.tsx > ...

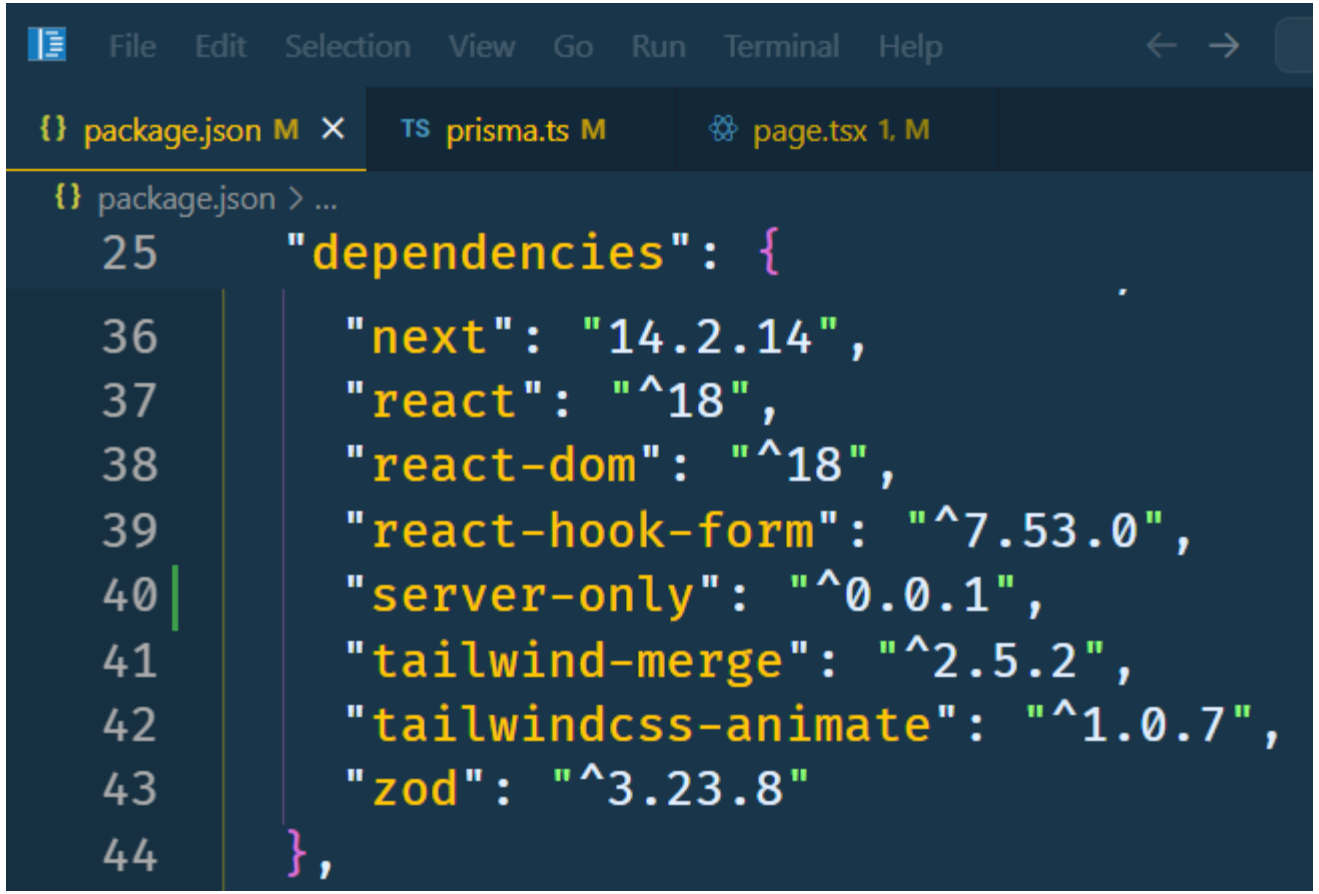
```tsx
type Props = {
  movie: Movie
  formAction: (formData: FormData) => void
}

export function MovieEditor({ movie, formAction }: Props) {
  const errorMessage = ''
  const form = useForm<Movie>({
    defaultValues: movie,
  })
  const { toast } = useToast()
  const posterPath = form.watch('posterPath')

  return (
    <div className="flex flex-col md:flex-row">
      <div className="md:w-1/3">···
      </div>
      <div className="max-w-xl md:w-2/3 md:pl-8">
        <Form {...form}>
          <form className="flex flex-col gap-4" action={formAction}>
            <input type="hidden" name="id" value={movie.id} />
```

Guarding client components against server code

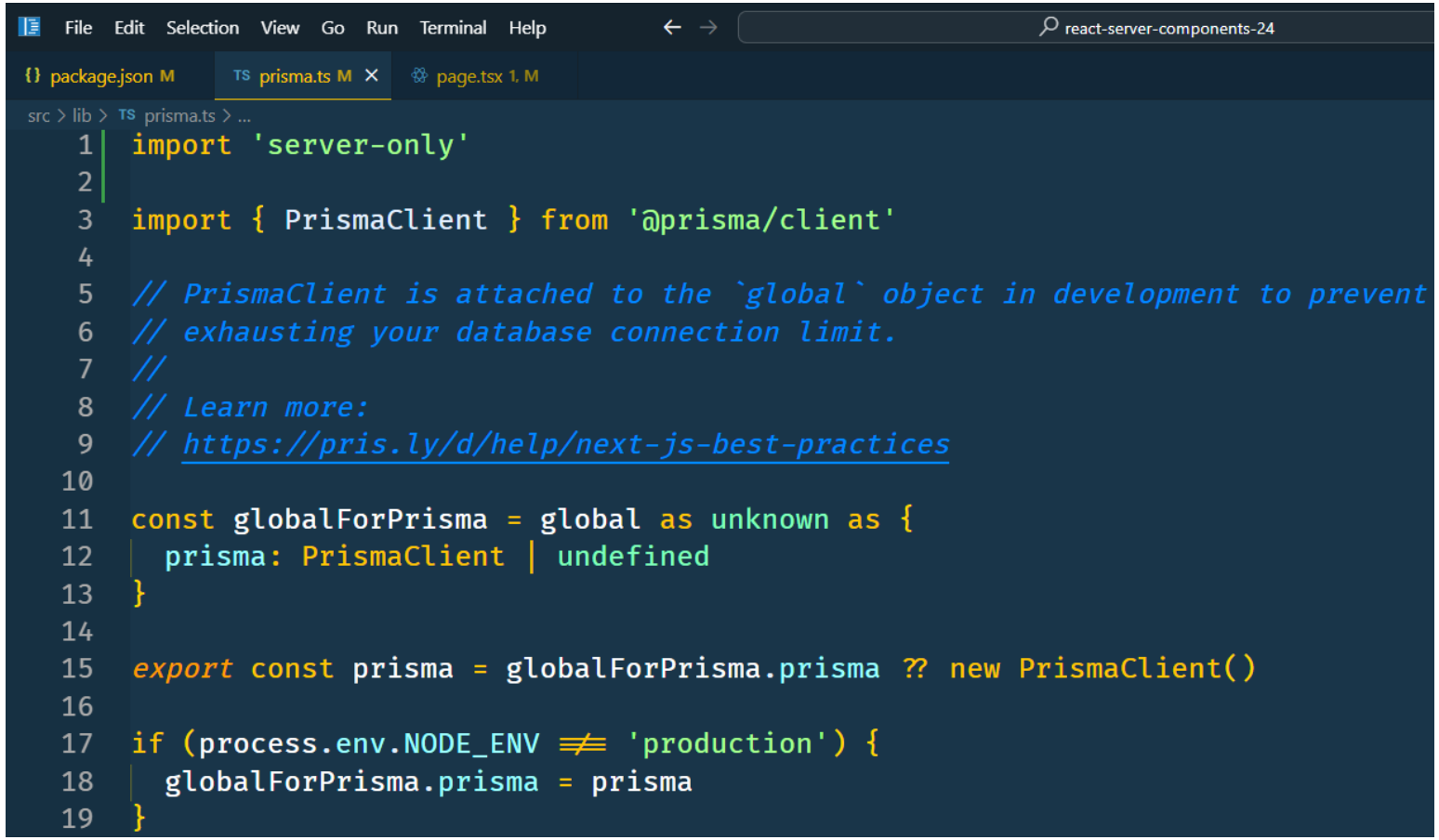
server-only

- **Components that render in the browser shouldn’t execute server code**
  - This would usually result in a runtime error
- An immediate **compile time error** is better
  - The server-only package does this
  - npm install server-only
- Add **import 'server-only’** to any code that should not be imported
  - Only needed in the modules that actually execute the Node code

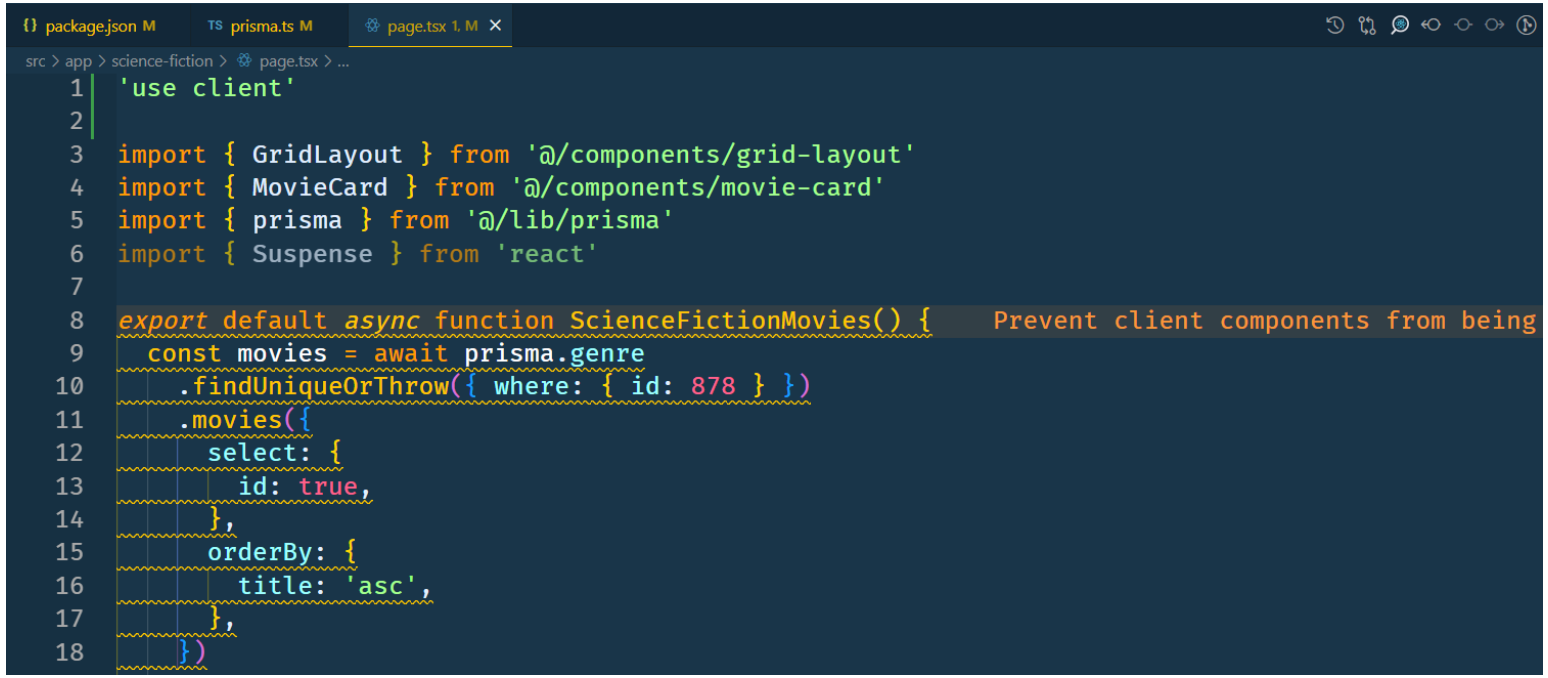
package.json



src\lib\ prisma.ts



src\app\science-fiction\page.tsx



The error


```
  × ./src/lib/prisma.ts
Error:
  × You're importing a component that needs server-only. That only works in a Server Component which is not supported in the pages/ direct
ory. Read more: https://nextjs.org/docs/getting-started/
    react-essentials#server-components


   ,-[C:\Repos\react-server-components-24\src\lib\prisma.ts:1:1]
 1 | import 'server-only'
   : ^^^^^^^^^^^^^^^^^^^^
 2 |
 3 | import { PrismaClient } from '@prisma/client'
   `----

Import trace for requested module:
./src/lib/prisma.ts
```

The useFormStatus() hook

useFormStatus hook

- The **useFormStatus()** hook gives **information about form submition**
  - The pending status let's you know if a submit is active
- ☞ Must be in a component that is **rendered as child from the <form>** ☜

src\components\submit-button.tsx

submit-button.tsx M

src > components > submit-button.tsx > ...

```tsx
import { useFormStatus } from 'react-dom'
import { Button } from './ui/button'

export function SubmitButton({ children }: React.PropsWithChildren) {
  const { pending } = useFormStatus()

  return (
    <Button type="submit" disabled={pending}>
      {children}
    </Button>
  )
}
```

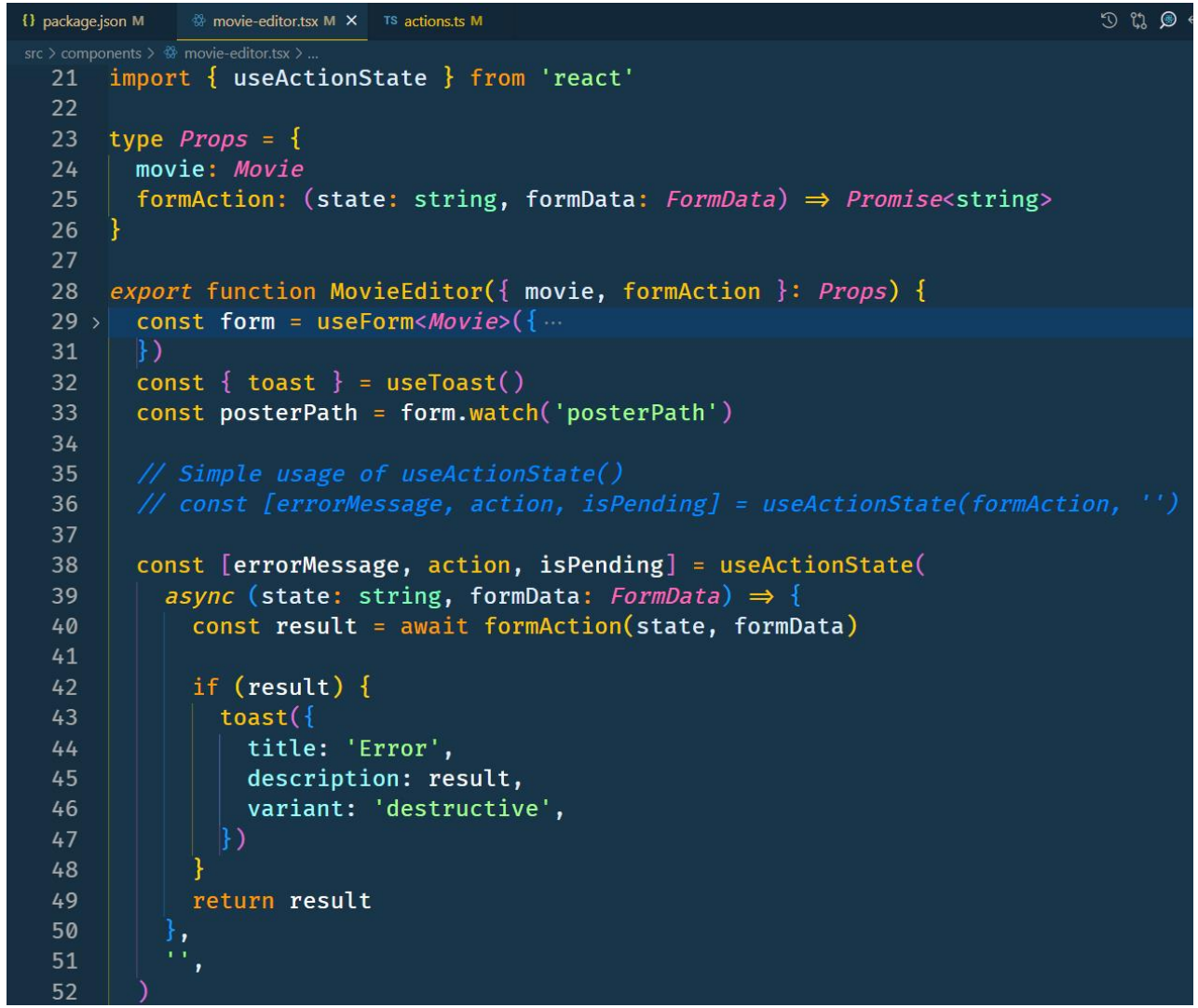
The useActionState() hook

useActionState hook

- Updates component **state based on the result of a form action**
  - The state round trips to the action function
  - Useful for form validation etc
- ☞ Note: **useFormState** for now with **production React/Next.js**! ☜
  - Doesn't expose an isPending status

package.json

src\components\movie-editor.tsx



src\server\ actions.ts

package.json M, M | movie-editor.tsx A, M | actions.ts A, M

src > server > actions.ts > ...

```ts
export async function handleSubmitMovieForm(
  state: string,
  formData: FormData,
): Promise<string> {
  //   await sleep(1_000);

  const idValue = formData.get('id')
  const id = Number(idValue)
  if (!id || isNaN(id)) {
    throw new Error('Invalid id')
  }

  const movie = await getMovieOrThrow(id)

  const movieRecord: Record<string, string | number | boolean> = movie
  formData.forEach((value, key) => {···
  })

  if (!movie.title) {
    return 'The movie title is required'
  }

  await updateMovie(id, movie)

  redirect(`/movie/${id}`)
}
```

Manually calling a server action

Manually calling a server action

- Server actions act as **normal asynchronous functions**
  - Makes the boundary between server and client almost transparent
- **Call like a normal async function** when needed
  - The network call is handled for you
- Return **any result you want**
  - As long as it can be serialized to JSON
- **Don't use throw new Error('Some message')**
  - ☞ Error messages are hidden in a production build ☜

src\components\favourite-heart.tsx

favourite-heart.tsx 1A, M

src > components > favourite-heart.tsx > ...

```tsx
import { toggleFavourite } from '@/server/actions'

type Props = {...
}

export function FavouriteHeart({ favourite, movieId }: Props) {
  return (
    <Heart
      aria-label={favourite ? 'Remove from favourites' : 'Add to favourites'}
      className={cn('cursor-pointer text-green-500', {
        'fill-green-500': favourite,
      })}
      onClick={async () => {
        try {
          await toggleFavourite(movieId)
        } catch (error) {
          console.error(error)
        }
      }}
    />
  )
}
```

The useOptimistic() hook

useOptimistic hook

- Create more **responsive user interfaces**
  - Immediately update the UI with an optimistic state before an asynchronous action
- Use whatever **optimistic state** you want
  - Automatically updated when the action completes

src\components\ favourite-heart.tsx

favourite-heart.tsx M

src > components > favourite-heart.tsx > ...

```tsx
import { useOptimistic, useTransition } from 'react'

type Props = {···
}

export function FavouriteHeart({ favourite, movieId }: Props) {
  const [isPending, startTransition] = useTransition()
  const [optimisticFavourite, setOptimisticFavourite] = useOptimistic(favourite)

  return (
    <Heart
      aria-label={favourite ? 'Remove from favourites' : 'Add to favourites'}
      className={cn('cursor-pointer text-green-500', {
        'fill-green-500': optimisticFavourite,
        'opacity-50': isPending,
      })}
      onClick={() => {
        startTransition(async () => {
          try {
            setOptimisticFavourite(!optimisticFavourite)
            await toggleFavourite(movieId)
          } catch (error) {
            console.error(error)
          }
        })
      }}
```

Error handling & retrying

Error handling & retrying

- An **ErrorBoundary will catch errors** in React Server Components
  - The normal expected React behavior
- Next.js makes it easy to **catch errors**
  - **Add a error.tsx** next to the page.tsx
  - They can be nested and the closest will be used

src\app\error-handling\error.tsx

error.tsx M

src > app > error-handling > error.tsx > ...

```tsx
type Props = {
  error: Error & { digest?: string }
  reset: () => void
}

export default function Error({ error, reset }: Props) {
  const router = useRouter()

  const tryAgainHandler = () => {
    startTransition(() => {
      router.refresh()
      reset()
    })
  }

  return (
    <div className="flex min-h-screen flex-col items-center justify-center gap-4">
      <h2 className="text-2xl font-bold">Something went wrong!</h2>
      <code className="text-center text-red-500">
        Message: {error.message}
        <br />
        (Digest: {error.digest})
      </code>
      <Button onClick={tryAgainHandler}>Try again</Button>
    </div>
  )
}
```

Cleaning up the code

Cleaning up the code

- It's considered a **best practice** not to put server logic in the UI
  - Server actions typically go into a separate actions.ts

src\app\movie\[id]\edit\page.tsx

page.tsx ...\edit M | actions.ts M | page.tsx ...\science-fiction M

src > app > movie > [id] > edit > page.tsx > ...

```tsx
import { MovieEditor } from '@/components/movie-editor'
import { handleSubmitMovieForm } from '@/server/actions'
import { getMovie } from '@/server/movie'

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

export default async function MovieEditPage({ params }: Props) {
  const movie = await getMovie(parseInt(params.id))

  if (!movie) {
    redirect('/404')
  }

  return (
    <div className="p-4">
      <MovieEditor movie={movie} formAction={handleSubmitMovieForm} />;
    </div>
  )
}
```

src\server\ actions.ts

page.tsx ...\edit M | actions.ts M | page.tsx ...\science-fiction M

src > server > actions.ts > ...

```
1  'use server'
2
3  import { revalidatePath } from 'next/cache'
4
5  import { sleep } from '@/lib/utils'
6  import { getMovieOrThrow, updateMovie } from './movie'
7  import { redirect } from 'next/navigation'
```

src\components\movie-card.tsx

page.tsx ...\edit M | actions.ts M | movie-card.tsx M | page.tsx ...\science-fiction M

src > components > movie-card.tsx > ...

```tsx
import { VoteStars } from './vote-stars'
import { FavouriteHeart } from './favourite-heart'
import { getMovieOrThrow } from '@/server/movie'

type Props = {
  movieId: number
} & React.ComponentProps<typeof Card>

export async function MovieCard({ movieId, ...props }: Props) {
  // await sleep(Math.random() * 5_000);
  const movie = await getMovieOrThrow(movieId)

  return (
    <Card className="flex h-full flex-col shadow-lg" {...props}>
```

src\app\science-fiction\page.tsx

page.tsx ...\edit M | actions.ts M | page.tsx ...\science-fiction M

src > app > science-fiction > page.tsx > ...

```tsx
import { GridLayout } from '@/components/grid-layout'
import { MovieCard } from '@/components/movie-card'
import { getScienceFictionMovies } from '@/server/movie'
import { Suspense } from 'react'

export default async function ScienceFictionMovies() {
  const movies = await getScienceFictionMovies()

  return (
    <div className="p-4">
      <h1 className="my-8 text-center text-4xl font-bold">
        Science Fiction Movies
      </h1>

      <GridLayout>
        {/* <Suspense fallback={<div>Loading ... </div>}> */}
        {movies.map((movie) => (
          <MovieCard key={movie.id} movieId={movie.id} />
        ))}
        {/* </Suspense> */}
      </GridLayout>
    </div>
  )
}
```

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

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'
- **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