# Data fetching patterns

In Next.js 15 ,
Explain Data Fetching Patterns:
1. Sequential data fetching pattern
2. Parallel data fetching pattern

Sure! In **Next.js 15**, data fetching patterns are important for optimizing performance and user experience. Let me explain the two common data fetching patterns you asked about:

---

### 1. Sequential Data Fetching Pattern

**Definition:**
In the sequential pattern, data fetching calls are made **one after another**. Each subsequent fetch starts only after the previous one has completed.

**How it works:**

- You await the result of the first fetch.
- Once it's done, you proceed to the second fetch, await it, and so on.

**Example:**

```js
async function fetchData() {
  const user = await fetch('/api/user').then(res => res.json());
  const posts = await fetch(`/api/posts?userId=${user.id}`).then(res => res.json());
  return { user, posts };
}
```

**When to use:**

- When the second fetch depends on the data from the first fetch (e.g., need user ID to fetch posts).
- When you want to keep requests ordered for some logic or business requirement.

**Downside:**

- Slower total loading time because requests are done one after another.

---

## 2. Parallel Data Fetching Pattern

**Definition:**

In parallel fetching, multiple data requests are made **at the same time**, and you wait for all of them to complete concurrently.

**How it works:**

- You initiate multiple fetches simultaneously without awaiting them immediately.
- Then use `Promise.all` to wait for all to resolve.

**Example:**

```js
async function fetchData() {
  const userPromise = fetch('/api/user').then(res => res.json());
  const postsPromise = fetch('/api/posts').then(res => res.json());

  const [user, posts] = await Promise.all([userPromise, postsPromise]);
  return { user, posts };
}
```

**When to use:**

- When multiple fetches are **independent** of each other.
- To improve performance by reducing total waiting time.

**Benefit:**

- Much faster overall fetching since all requests happen concurrently.

---

## Summary

| Pattern | Use Case | Performance | Example Scenario |
|---|---|---|---|
| Sequential | When fetches depend on each other | Slower (waiting in sequence) | Fetch user, then fetch posts by user ID |
| Parallel | Independent fetches | Faster (concurrent requests) | Fetch user data and posts independently |

---

If you'd like, I can also help you with how Next.js 15 uses these patterns in its new routing or server components. Want me to dive into that?

now lets provide a complete example on sequential data fetching

Absolutely! Here's a **complete example of sequential data fetching** in Next.js 15, using the new **App Router** and **Server Components** (which is the recommended pattern).

---

### Scenario:

You want to fetch user data first, then fetch posts by that user (so the second fetch depends on the first).

---

## 1. Folder Structure

```markdown
app/
  page.jsx
  components/
    UserPosts.jsx
```

---

## 2. Complete Code Example

`app/page.jsx`

```jsx
// app/page.jsx
import UserPosts from './components/UserPosts';

export default async function Page() {
  // Step 1: Fetch user data
  const userResponse = await fetch('https://jsonplaceholder.typicode.com/users/1', {
    // Use 'no-store' to ensure fresh data on every request
    cache: 'no-store',
  });
  const user = await userResponse.json();

  // Step 2: Fetch posts for the user (depends on user.id)
  const postsResponse = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${user.id}`, {
    cache: 'no-store',
  });
  const posts = await postsResponse.json();

  return (
    <main>
      <h1>User Info</h1>
      <p>Name: {user.name}</p>
      <p>Email: {user.email}</p>

      <UserPosts posts={posts} />
    </main>
  );
}
```

`app/components/UserPosts.jsx`

```jsx
// app/components/UserPosts.jsx
export default function UserPosts({ posts }) {
  return (
    <>
      <h2>User Posts</h2>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>
            <strong>{post.title}</strong>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </>
  );
}
```

---

### Explanation:

- The page fetches the user first.
- Only **after** the user data is loaded, it fetches the posts by that user.
- Because the posts fetch depends on the user ID, we **sequentially** wait for the user fetch to complete first.
- This is done inside an async Server Component ( `app/page.jsx` ).
- The posts are then passed down to a child component to render.

---

### Notes:

- Using `cache: 'no-store'` disables caching to always get fresh data on every request (optional).
- This approach is straightforward but slower compared to parallel fetching because the second fetch has to wait for the first one.

---

If you want, I can also provide an example for **parallel fetching** next! Want me to do that?

yes

Great! Here's the **complete example of parallel data fetching** in Next.js 15, again using the App Router and Server Components.

---

### Scenario:

You want to fetch **user data** and **posts** independently (no dependency between fetches), so you can fetch both **in parallel** for faster performance.

---

## 1. Folder Structure (same as before)

```markdown
app/
  page.jsx
  components/
    UserPosts.jsx
```

---

## 2. Complete Code Example

`app/page.jsx`

```jsx
// app/page.jsx
import UserPosts from './components/UserPosts';

export default async function Page() {
  // Start both fetches simultaneously (without awaiting yet)
  const userPromise = fetch('https://jsonplaceholder.typicode.com/users/1', {
    cache: 'no-store',
  }).then(res => res.json());

  const postsPromise = fetch('https://jsonplaceholder.typicode.com/posts?userId=1', {
    cache: 'no-store',
  }).then(res => res.json());

  // Await both promises in parallel
  const [user, posts] = await Promise.all([userPromise, postsPromise]);

  return (
    <main>
      <h1>User Info</h1>
      <p>Name: {user.name}</p>
      <p>Email: {user.email}</p>

      <UserPosts posts={posts} />
    </main>
  );
}
```

`app/components/UserPosts.jsx`

```jsx
// app/components/UserPosts.jsx
export default function UserPosts({ posts }) {
  return (
    <>
      <h2>User Posts</h2>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>
            <strong>{post.title}</strong>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </>
  );
}
```

---

## Explanation:

- Both `fetch` calls are initiated **at the same time** without waiting for each other.
- Then `Promise.all` is used to wait for **both** to resolve.
- This speeds up the total load time since data is fetched concurrently.
- The posts fetch uses a static userId ( `1` ) here, showing independence from the user fetch.

---

## Why Use This?

- When you have multiple independent data sources.
- To optimize page load speed and responsiveness.

---

If you want, I can help you convert these examples to **Client Components** or show how to handle errors and loading states next. Want to explore that?

yes

Awesome! Let's enhance the example by converting it to use **Client Components** with **loading** and **error handling**.

---

### Why Client Components?

- Server Components fetch data on the server (good for SEO, performance).
- Client Components fetch data on the client (good for interactivity, real-time updates).

---

### Example: Parallel Data Fetching in a Client Component with Loading and Error Handling

---

### 1. Folder Structure

```markdown
app/
  page.jsx
  components/
    UserWithPosts.jsx
```

---

### 2. Code

`app/page.jsx`

```jsx
// app/page.jsx
import UserWithPosts from './components/UserWithPosts';
```

```
export default function Page() {
  return (
    <main>
      <h1>User Info & Posts</h1>
      <UserWithPosts />
    </main>
  );
}
```

`app/components/UserWithPosts.jsx`

```jsx
'use client';  // Mark as a client component

import React, { useEffect, useState } from 'react';

export default function UserWithPosts() {
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchData() {
      try {
        setLoading(true);
        setError(null);

        // Fetch user and posts in parallel
        const userPromise = fetch('https://jsonplaceholder.typicode.com/users/1').then(res => {
          if (!res.ok) throw new Error('Failed to fetch user');
          return res.json();
        });

        const postsPromise = fetch('https://jsonplaceholder.typicode.com/posts?userId=1').then(res => {
          if (!res.ok) throw new Error('Failed to fetch posts');
          return res.json();
        });

        const [userData, postsData] = await Promise.all([userPromise, postsPromise]);
        setUser(userData);
        setPosts(postsData);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    }

    fetchData();
  }, []);

  if (loading) return <p>Loading user and posts...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <h2>{user.name} ({user.email})</h2>

      <h3>Posts:</h3>
      <ul>
        {posts.map(post => (
          <li key={post.id}>
            <strong>{post.title}</strong>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

---

## What’s happening here?

- This is a **Client Component** (`'use client'` directive).
- `useEffect` triggers the data fetch after the component mounts.
- Fetches user and posts **in parallel** with `Promise.all`.
- Manages loading and error states with React state.

- Shows loading text until data is ready, or error message if something goes wrong.

---

### When to use this?

- When you need client-side interactivity or dynamic updates.
- When you want to handle loading and error states in the UI.
- When you want more control over the data fetching lifecycle.

---

Would you like me to also show how to do **loading states and error handling** with **Server Components** (using `React Suspense`)?

yes

Perfect! Let's explore how to handle **loading states** and **error handling** with **Server Components** in Next.js 15 using **React Suspense** and **Error Boundaries**.

---

## Loading & Error Handling in Server Components (Next.js 15)

---

### How it works:

- Server Components fetch data on the server before sending HTML to the client.
- You can wrap components with React's `<Suspense>` to show fallback UI during async rendering.
- Use React's error boundaries to catch errors during rendering or data fetching.

---

### 1. Folder Structure

```markdown
app/
  page.jsx
  components/
    UserWithPosts.jsx
    Loading.jsx
    ErrorBoundary.jsx
```

---

## 2. Code

`app/page.jsx`

```jsx
import React, { Suspense } from 'react';
import UserWithPosts from './components/UserWithPosts';
import Loading from './components/Loading';
import ErrorBoundary from './components/ErrorBoundary';

export default function Page() {
  return (
    <main>
      <h1>User Info & Posts</h1>

      <ErrorBoundary>
        <Suspense fallback={<Loading />}>
          <UserWithPosts />
        </Suspense>
      </ErrorBoundary>
    </main>
  );
}
```

---

`app/components/UserWithPosts.jsx`

```jsx
// Server Component (no 'use client')
export default async function UserWithPosts() {
  // Fetch user and posts in parallel
  const userPromise = fetch('https://jsonplaceholder.typicode.com/users/1', {
    cache: 'no-store',
  }).then(res => {
    if (!res.ok) throw new Error('Failed to fetch user');
    return res.json();
  });

  const postsPromise = fetch('https://jsonplaceholder.typicode.com/posts?userId=1', {
    cache: 'no-store',
  }).then(res => {
    if (!res.ok) throw new Error('Failed to fetch posts');
    return res.json();
  });

  const [user, posts] = await Promise.all([userPromise, postsPromise]);

  return (
    <section>
      <h2>{user.name} ({user.email})</h2>

      <h3>Posts:</h3>
      <ul>
        {posts.map(post => (
          <li key={post.id}>
            <strong>{post.title}</strong>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </section>
  );
}
```

---

`app/components/Loading.jsx`

```jsx
export default function Loading() {
  return <p>Loading user and posts...</p>;
}
```

---

`app/components/ErrorBoundary.jsx`

```jsx
'use client'; // Error boundaries must be Client Components

import React from 'react';

export default class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true, error };
  }

  componentDidCatch(error, errorInfo) {
    // You can log error info here if you want
    console.error('ErrorBoundary caught an error:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <p>Error loading data: {this.state.error.message}</p>;
    }

    return this.props.children;
  }
}
```

---

## Explanation:

- `UserWithPosts` is a **Server Component** fetching data in parallel.
- React Suspense's `fallback` prop shows `<Loading />` while waiting for async data.
- If fetching fails, the error is thrown and caught by `ErrorBoundary`, displaying an error message.
- `ErrorBoundary` must be a **Client Component** (uses class component to catch errors).

---

**Benefits:**

- Clean separation of loading UI and error handling.
- Server renders content only when data is ready, improving SEO and performance.
- Suspense and error boundaries handle async rendering states gracefully.

---

If you want, I can also show you how to do **incremental loading** with Suspense or how to combine client & server components efficiently. Would you like that?

yes

Awesome! Let’s talk about **incremental loading** with Suspense and how to combine **Client & Server Components** efficiently in Next.js 15.

---

## Incremental Loading with Suspense

### What is incremental loading?

Instead of waiting for **all** data to load before rendering anything, you can progressively load parts of the UI as their data becomes ready, improving perceived performance and user experience.

---

### How to do it?

- Break your UI into smaller **Server Components**, each fetching their own data.
- Wrap each part in its own `<Suspense>` with individual fallback UI.
- This way, sections of your page can show independently as their data arrives.

---

### Example Structure:

```markdown
app/
  page.jsx
  components/
```

```
    UserInfo.jsx
    UserPosts.jsx
    LoadingUser.jsx
    LoadingPosts.jsx
```

---

## Code Example

`app/page.jsx`

```jsx
import React, { Suspense } from 'react';
import UserInfo from './components/UserInfo';
import UserPosts from './components/UserPosts';
import LoadingUser from './components/LoadingUser';
import LoadingPosts from './components/LoadingPosts';

export default function Page() {
  return (
    <main>
      <h1>User Info & Posts</h1>

      <Suspense fallback={<LoadingUser />}>
        <UserInfo userId={1} />
      </Suspense>

      <Suspense fallback={<LoadingPosts />}>
        <UserPosts userId={1} />
      </Suspense>
    </main>
  );
}
```

---

`app/components/UserInfo.jsx`

```jsx
export default async function UserInfo({ userId }) {
  const res = await fetch(`https://jsonplaceholder.typicode.com/users/${userId}`, { cache: 'no-
store' });
  if (!res.ok) throw new Error('Failed to fetch user');
  const user = await res.json();

  return (
    <section>
      <h2>{user.name}</h2>
      <p>{user.email}</p>
    </section>
  );
}
```

---

`app/components/UserPosts.jsx`

```jsx
export default async function UserPosts({ userId }) {
  const res = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${userId}`, {
cache: 'no-store' });
  if (!res.ok) throw new Error('Failed to fetch posts');
  const posts = await res.json();

  return (
    <section>
      <h3>Posts:</h3>
      <ul>
        {posts.map(post => (
          <li key={post.id}>
            <strong>{post.title}</strong>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </section>
  );
}
```

---

## Loading Components

```jsx
// LoadingUser.jsx
export default function LoadingUser() {
  return <p>Loading user info...</p>;
}

// LoadingPosts.jsx
export default function LoadingPosts() {
  return <p>Loading posts...</p>;
}
```

---

## Combining Client & Server Components Efficiently

### Pattern

- Use **Server Components** for fetching data and rendering static UI on the server.
- Use **Client Components** only where interactivity or lifecycle hooks (`useState`, `useEffect`) are needed.
- Pass fetched data from Server Components down to Client Components as props.

---

### Example:

```jsx
// Server Component (app/page.jsx)
import InteractivePostList from './components/InteractivePostList';

export default async function Page() {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts?userId=1', { cache: 'no-store' });
  const posts = await res.json();

  return (
    <main>
      <h1>Posts</h1>
      <InteractivePostList initialPosts={posts} />
    </main>
  );
}
```

```jsx
// Client Component (app/components/InteractivePostList.jsx)
'use client';

import React, { useState } from 'react';

export default function InteractivePostList({ initialPosts }) {
  const [posts, setPosts] = useState(initialPosts);

  // Add interactivity here (like filtering, liking posts, etc.)

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>
          <strong>{post.title}</strong>
          <p>{post.body}</p>
        </li>
      ))}
    </ul>
  );
}
```

---

### Benefits:

- Server Components reduce client bundle size and improve performance.
- Client Components enable rich interactivity only where needed.
- Data is fetched once on the server and passed down to client safely.

---

If you want, I can help you build a specific interactive feature combining client/server components or show real-world use cases! Just let me know.