BUILDING LARGE SCALE WEB APPS

A REACT FIELD GUIDE

ADDY OSMANI

HASSAN DJIRDEH

Building large scale web apps

A React field guide

Bonus H2/2024 Chapters

By Addy Osmani and Hassan Djirdeh

Routing

Routing (i.e, URL Routing) is a crucial aspect of building large-scale React applications. It allows users to navigate through different views and components, providing a seamless and intuitive user experience. As applications grow in size, effective routing becomes increasingly important to ensure maintainability, scalability, and performance.

In this chapter, we'll explore the fundamentals of routing in React, dive into various approaches and best practices, and discuss specific solutions such as React Router and Next.js App Router. We'll also cover topics like nested routing, data loading, and more. While this chapter is not intended to be a very detailed guide, it will equip you with an understanding of the importance of routing and how to implement it in your large-scale React applications.

Why does routing matter to users?

Routing is crucial for users because it plays a significant role in creating a seamless and intuitive user experience within a web application. Routing helps facilitate:

1. **Navigation and discoverability**. Routing provides a clear and logical structure to the application, making it easier for users to navigate through different sections and discover available content and functionality. Well-defined routes create an intuitive map of the application, enhancing the overall user experience.

2. **Bookmarking and sharing**. With routing, each view or page in the application has a unique URL. This allows users to bookmark specific routes for quick access later and easily share content or functionality with others by simply copying and pasting the relevant URL.

3. **Back and forward navigation**. Proper routing enables users to utilize the browser's back and forward buttons effectively. This preserves the application's state when navigating through browsing history, providing a smooth and intuitive experience that aligns with users' expectations of web navigation.

4. **SEO and indexing**. Distinct URLs for each view or page improve the application's visibility to search engines. This allows search engines to crawl and index the content, making it discoverable through search results and potentially increasing organic traffic to the application.

5. **Deep linking and external access**. Routing facilitates deep linking, allowing users to access specific views or pages directly through URLs. This is particularly useful when users follow external links or receive URLs through other channels, as they can be directed straight to the relevant part of an application.

6. **State management and data persistence**. With proper routing, applications can manage and persist state across different views. This ensures that relevant data and user progress are maintained as users navigate through various sections, providing a consistent and uninterrupted experience.

7. **Permission-based access and authentication**. Routing allows for the implementation of permission-based access control and authentication mechanisms. Different routes can be protected or restricted based on user roles or authentication status, ensuring that users only access appropriate sections of the application, thus enhancing security and user trust.

By implementing effective routing, developers can create web applications that are not only more organized and maintainable but also provide a significantly improved user experience. Routing transforms a collection of components and views into a cohesive, navigable, and user-friendly application that meets modern web standards and user expectations.

Understanding Routing in React

Before diving into the specifics of routing in React, let's further establish a solid foundation by understanding what routing entails and its significance in modern web applications.

Definition of Routing

Routing (i.e., URL Routing) is the process of mapping URLs (Uniform Resource Locators) to specific components or pages within an application. It enables users to navigate between different views by entering a URL in the browser's address bar or by clicking on hyperlinks.

For instance, clicking a link might change the URL from `https://website.com` to `https://website.com/about/`. This change in URL represents routing. When we navigate to the root path (`/`) of a website, we're typically accessing the home page. Similarly, visiting `/about` would display the "about page," and so forth.

While it is possible to create applications without routing, it becomes increasingly complicated as the application grows. Implementing routes provides a way to associate a particular URL with a corresponding component or set of components, allowing for a structured and organized application architecture.

Traditional Web Applications vs. Single-Page Applications (SPAs)

In traditional web applications, each URL typically corresponds to a separate HTML page served from the server. When a user navigates to a different URL, the browser sends a request to the server, which responds with a new HTML page. This approach requires a potentially full page reload for each navigation, resulting in a slower and less fluid user experience.

On the other hand, single-page applications (SPAs) built with React follow a different approach. In an SPA, the entire application is loaded on a single page, and the content is dynamically updated based on the current URL. When a user navigates to a different URL, React updates the UI by rendering the appropriate components without requiring a full page reload. This approach provides a more seamless and responsive user experience, as the application feels more like a native desktop or mobile app.

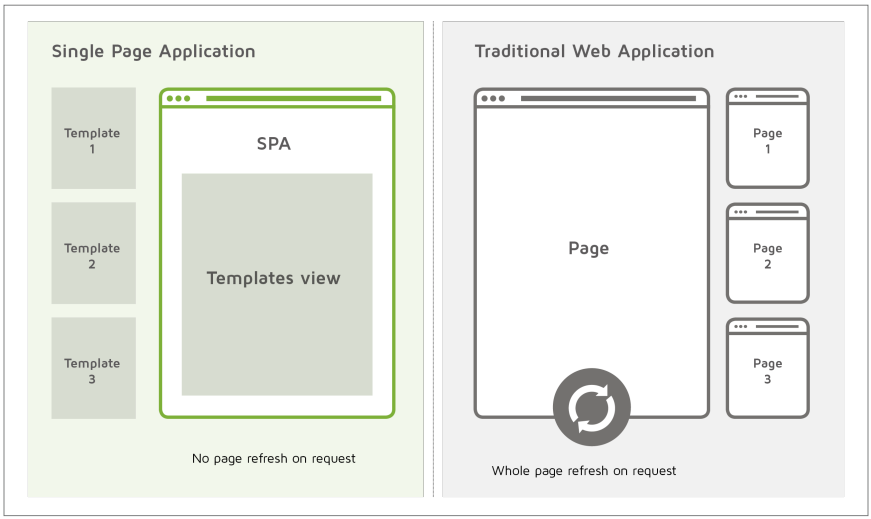


Figure 16-1. Comparison between SPA and non-SPA. From the article Single Page Applications – Why they make sense by Digital Clarity Group.

Each approach has its strengths and use cases. Traditional web applications excel in scenarios where SEO is crucial and initial page load time is a priority. They're often simpler to develop and can be more suitable for content-heavy sites. On the other hand, SPAs shine in creating interactive, app-like experiences. They offer smoother navigation, better performance after initial load, and a more responsive feel. This makes them ideal for complex web applications where user engagement and interactivity are key.

In the context of React development, the SPA approach is more commonly adopted. However, modern frameworks and techniques are blurring the lines between these two paradigms, allowing developers to leverage the benefits of both approaches.

Approaches to Routing in React

React doesn't provide a built-in routing solution, but the ecosystem offers several powerful libraries and frameworks for routing. The two main approaches in React are client-side routing and server-side routing.

Client-side routing

Client-side routing is the most common approach in React applications. The routing logic is handled entirely in the browser using JavaScript. When a user navigates to a different URL or clicks a link, the application intercepts the navigation event and updates the UI without making a server request. This approach is crucial for single-page applications (SPAs) as it allows for fast transitions between pages, giving the feel of a seamless, continuous experience without the traditional page reloads associated with server-side routing.

Benefits:

- **Fast and seamless navigation**. No additional server requests are needed, resulting in smoother navigation.
- **Decoupled frontend and backend**. Allows for clear separation, enabling independent development and maintenance.

Limitations:

- **Slower initial load time**. The entire application needs to be loaded up front.
- **Search engine optimization (SEO) challenges**. Search engine crawlers may struggle to index dynamically rendered content.

Server-side routing

Server-side routing handles the routing logic on the server. When a user enters a URL or clicks a link, the server processes the request and sends back the appropriate HTML content. This approach is foundational for traditional web applications as it allows each page request to be fully processed on the server.

Benefits:

- **Better SEO**. Search engine crawlers can easily index fully rendered HTML pages.
- **Faster initial load time**. The server can quickly render and send HTML content.

Limitations:

- **Slower navigation**. Each navigation requires a server round trip.
- **Tighter frontend-backend couplings**. Often requires closer integration, complicating development and maintenance.

Both approaches have their place in modern web development, and the choice depends on the specific requirements of an application. Some frameworks, like Next.js, even offer hybrid solutions that combine the benefits of both client-side and server-side routing.

React Routing Solutions

Now that we understand the different approaches to routing in React, let's explore some popular routing solutions used in the React ecosystem, particularly React Router and Next.js App Router.

React Router

React Router is a standalone library that enables client-side routing in React applications. It provides a declarative way to define routes and map them to corresponding components, allowing us to create a navigation structure for our application. React Router keeps the UI in sync with the URL, making it easy to handle client-side routing and creating a seamless navigation experience.

React Router offers a comprehensive set of features, including declarative routing, dynamic route matching, nested routes, programmatic navigation, route guards, and support for lazy-loading. We'll explore some of these in the following sections.

Configuring Routes

When setting up routing in a React application using React Router v6, the approach is slightly different from earlier versions. One way of setting up routing involves utilizing the createBrowserRouter function combined with RouterProvider to manage our routes. This method leverages the HTML5 history API to keep the user interface in sync with the URL but shifts towards a more modular setup.

Here's an example of how to configure routes using `createBrowserRouter` along with `RouterProvider`:

Configuring routes with JSX

```
import {
  createBrowserRouter,
  RouterProvider,
  Route,
  createRoutesFromElements,
} from "react-router-dom";

const router = createBrowserRouter(
  createRoutesFromElements(
    <Route>
      <Route path="/" element={<Home />} />
      <Route
        path="/about"
        element={<About />}
      />
      <Route
```

```
        path="/contact"
        element={<Contact />}
      />
    </Route>,
  ),
);

function App() {
  return <RouterProvider router={router} />;
}
```

In this example, we use `createBrowserRouter` to create a router instance and define routes using `createRoutesFromElements()`, which allows for a JSX-based route configuration. Each `Route` component specifies a path and an element to render when that path matches.

Navigating between Routes

React Router provides the Link component for declarative navigation between routes. We can use the `Link` component to create clickable links that navigate to different routes without triggering a full page refresh.

Facilitating navigation with the `Link` component

```
import { Link } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
    </nav>
```

```
  );
}
```

In the above example, we create a navigation menu using the `Link` component. The `to` prop specifies the route to navigate to when the link is clicked.

Route Parameters and Data Loading

React Router allows us to define dynamic routes that accept parameters. This is useful when we want to pass data through the URL and render components based on those parameters.

Route parameters and the `loader` prop

```
import {
  useLoaderData,
  useParams,
} from "react-router-dom";

function UserProfile() {
  const { userId } = useParams();
  const userData = useLoaderData();

  return (
    <div>
      <h1>User Profile</h1>
      <p>User ID: {userId}</p>
      <p>User Name: {userData.name}</p>
    </div>
  );
}

const router = createBrowserRouter(
  createRoutesFromElements(
    <Route
      path="/users/:userId"
      element={<UserProfile />}
      loader={async ({ params }) => {
        return fetch(
          `/api/users/${params.userId}`,
        );
      }}
```

```
    />,
  ),
);
```

In the above example, React Router is used to define dynamic routes that accept parameters and load data specific to those parameters. The useParams Hook retrieves the `userId` from the URL, which is then used to fetch user data via a loader function defined on the route. This function asynchronously fetches user data from an API endpoint using the `userId`, and the fetched data is accessed in the `UserProfile` component through the useLoaderData Hook.

Nested Routes

React Router supports nested routes, allowing us to create a hierarchical structure of routes and components. This is particularly useful when we have complex application structures or when we want to render nested components based on the URL.

Nested Routes with the `Outlet` component

```
const router = createBrowserRouter(
  createRoutesFromElements(
    <Route
      path="/dashboard"
      element={<Dashboard />}
    >
      <Route index element={<Overview />} />
      <Route
        path="profile"
        element={<Profile />}
      />
      <Route
        path="settings"
        element={<Settings />}
      />
    </Route>,
  ),
);

function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <nav>
```

```
          <Link to="/dashboard">Overview</Link>
          <Link to="/dashboard/profile">
            Profile
          </Link>
          <Link to="/dashboard/settings">
            Settings
          </Link>
        </nav>

        {/*
          This component renders the active
          child route component.
        */}
        <Outlet />
      </div>
    );
}
```

In the above example, `Dashboard` serves as a parent route component with several child routes defined within it. The `Outlet` component is used to render the appropriate child component based on the current URL path. This setup facilitates a clear hierarchical structure within the application and aligns with React Router's philosophy of linking URL segments to layouts.

Lazy-loading and code-splitting

React Router supports lazy-loading and code-splitting, allowing us to load components, loaders, actions, and other route-specific code on demand. This improves the performance of an application by reducing the initial bundle size and loading route-specific code only when needed.

One way of conducting lazy-loading is to use the lazy prop. The `lazy` prop accepts an async function that typically returns the result of a dynamic import. This function is executed after route matching occurs, allowing for the lazy-loading of non-route-matching portions of a route definition.

Using the `lazy` prop to lazily resolve route definitions

```
import {
  createBrowserRouter,
  RouterProvider,
  createRoutesFromElements,
```

```
  Route,
} from "react-router-dom";

/*
  Define routes with the 'lazy' prop for
  dynamic loading
*/
let routes = createRoutesFromElements(
  <Route path="/" element={<Layout />}>
    <Route
      path="a"
      lazy={() => import("./a")}
    />
  </Route>,
);

function App() {
  return (
    <RouterProvider
      router={createBrowserRouter(routes)}
    />
  );
}

export default App;
```

In this example, the route definitions for path "a" are lazily loaded. The `lazy` function returns a dynamic import of the respective module. In the lazy-loaded route modules (e.g., `a.js`), we can export the properties we want to be defined for the route:

Module where route properties are exported

```
export async function loader({ request }) {
  // ...
}

export function Component() {
  // ...
}

export function ErrorBoundary() {
  // ...
}
```

```
// ...
```

The `lazy` function can resolve and return various route properties, which are then incorporated into the route definition when the route is accessed.

The documentation notes that `lazy` cannot be used to define route-matching properties (like `path`, `index`, `children`, `caseSensitive`, etc.) as these are needed for initial route matching before the lazy function is executed.

By using the `lazy` prop, we can significantly reduce our initial bundle size and improve the performance of our React Router application, especially for large applications with many routes.

Conclusion

React Router is a library that simplifies client-side routing in React applications by offering a declarative approach to defining routes and mapping them to components, thereby creating a structured and intuitive navigation framework.

While this guide has introduced some core features and their implementations, React Router boasts a vast array of additional functionalities. For a comprehensive understanding and to explore more advanced topics, delving into the official documentation is highly recommended.

Next.js and the App Router

While React Router provides a flexible and powerful solution for client-side routing in React applications, Next.js takes a different approach by providing an integrated, file-system based routing solution called the App Router. This routing system is built into the Next.js framework itself, offering a more opinionated and potentially simpler way to handle routing in React applications. Server-side rendering (SSR) is integral to this approach, enhancing SEO, improving initial page load times, and delivering better performance on slower devices or connections.

When implementing SSR in a React application, the routing logic needs to be handled on both the server and the client. Frameworks like Next.js provide built-in support for SSR and make it easier to implement. Next.js handles the server-side rendering and routing out of the box, allowing developers to focus on building components and defining routes.

Furthermore, with its new App Router architecture, Next.js has fully embraced React Server Components. We'll briefly discuss Server and Client Components in this section, but we'll go into a lot more detail in the upcoming chapter—**The Future of React**.

Key differences from React Router

1. **File-system based routing**. Instead of defining routes programmatically, Next.js uses an app's file and folder structure to automatically create routes.

2. **Server-side rendering (SSR) and static site generation (SSG)**. Next.js provides built-in support for these rendering methods, which can improve performance and SEO.

3. **Automatic code splitting.** Next.js automatically splits code by routes, potentially improving load times.

4. **API Routes**. Next.js allows us to create API endpoints as part of your application structure.

Let's explore some key features of the Next.js App Router.

File-based routing

In the Next.js App Router setting, we create routes by adding files to the `app` directory, and the file path becomes the URL path.

`app/` directory

```
app/
├── page.js
├── about/
│   └── page.js
└── blog/
    └── [slug]/
```

```
        └── page.js
```

In this structure:

- `/` routes to `app/page.js`.
- `/about` routes to `app/about/page.js`.
- `/blog/[slug]` routes to `app/blog/[slug]/page.js`.

Server and Client Components

React Server Components are a new capability in React that allows us to create stateless React components that run on the server. These components bring the power of server-side processing to the React architecture, enabling us to offload certain computations and data-fetching tasks from the client to the server. Client Components, on the other hand, are executed on the client side and handle interactive aspects of the app, such as user inputs and dynamic updates.

With Next.js App Router, Server and Client Components can both be utilized in a powerful and efficient manner. Server components are the default but to create a Client Component, we need to opt-in by using the "use client" directive at the top of the file.

A Client Component

```
"use client"

import { useState } from "react";

export default function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button
        onClick={() => setCount(count + 1)}
      >
        Increment
      </button>
    </div>
  );
```

```
}
```

Data Fetching

With Server Components, we can fetch data directly within our component, and Next.js will automatically handle the data fetching on the server.

Fetching data on the server with Next.js's extended fetch function

```
async function getProduct(id) {
  const res = await fetch(
    `https://api.example.com/products/${id}`,
  );

  if (!res.ok) {
    throw new Error("Failed to fetch data");
  }

  const productData = await res.json();

  return productData;
}

async function ProductPage({ params }) {
  const product = await getProduct(params.id);

  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
    </div>
  );
}

export default ProductPage;
```

Loading States and Error Handling

The App Router setup provides a convention for handling loading states and errors. We can create loading.js and error.js files to handle these states.

`loading.js` and `error.js` files

```
app/
├── products/
│   └── [id]/
│       ├── page.js # Main page for /products/:id
│       ├── loading.js # Loading UI for /products/:id
│       └── error.js # Error UI for /products/:id
```

Next.js will automatically show the loading UI while data fetching is in progress, and the error UI if an unhandled error occurs.

Best Practices and Gotchas

When working with Next.js App Router and React Server Components, keep these best practices and potential gotchas in mind:

1. Use Server Components by default and only "opt-in" to Client Components when needed for interactivity or client-side state management. This keeps your client-side JavaScript bundle lean.

2. Be mindful of the boundary between Server and Client Components. Remember, Server Components cannot use client-side APIs or state.

3. Leverage the `loading.js` and `error.js` files for better user experience during data fetching and error handling.

4. Use the Link component for client-side navigation between routes. This ensures the navigation is handled by the App Router optimally.

5. Remember that Server Components always render on the server, even for client-side navigation. This means you should aim to keep them lightweight and avoid expensive computations.

6. Be cautious when passing data between Server and Client Components. Large data transfers can impact performance. Consider fetching data in the Client Component if it's not needed for the initial render.

7. Leverage caching mechanisms for data fetching in Server Components to avoid unnecessary round trips to the server.

8. If you need to share state between Server Components, consider lifting the state up to a shared Client Component.

9. Regularly profile and measure your application's performance to identify and optimize bottlenecks.

Conclusion

Next.js App Router represents a significant evolution in the way React applications handle routing, merging server-side efficiencies with client-side interactivity through its innovative use of Server and Client Components. By leveraging file-system-based routing along with the robust capabilities of Server Components to offload data fetching and computations to the server, Next.js optimizes performance and improves user experience across diverse web applications.

While this guide has introduced some basic features and their implementation, Next.js's App Router contains a deep array of functionalities that cater to more advanced development needs. Do refer to the detailed documentation for comprehensive guidance and to explore additional features in depth.

Wrap Up

Routing is a crucial aspect of building large-scale React applications. It enables users to navigate through different views and components, providing a seamless and intuitive user experience. As you embark on building large-scale React applications, keep in mind the following key points:

1. Choose the appropriate routing approach based on your application's requirements, considering factors like performance, SEO, and server-side rendering.

2. Leverage the power of nested routing to create modular and maintainable application structures.

3. Implement lazy loading, code splitting, and other performance optimization techniques to improve the loading times and user experience of your application.

4. Integrate routing with data fetching, state management, and other essential aspects of your application to create a cohesive and efficient development experience.

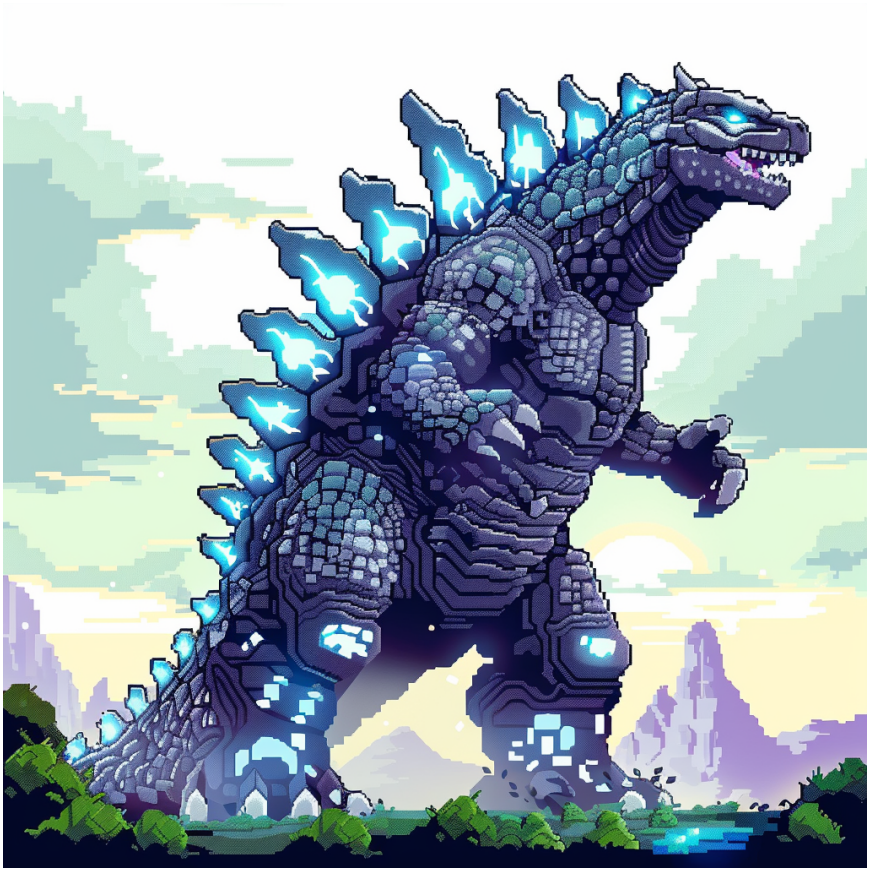
5. Test your routes and components thoroughly to catch bugs, ensure reliability, and maintain a high-quality codebase.

Remember, the key to successful routing in large-scale React applications lies in finding the right balance between simplicity, performance, and maintainability. By following best practices, leveraging powerful tools and libraries, and continuously refining your routing implementation, you can create robust and scalable applications that provide an exceptional user experience.

User-centric API Design

APIs (Application Programming Interfaces) enable communication between different software systems. They specify the methods and data structures that developers can use to interact with a service or platform.

API design is a crucial cornerstone of modern software development, enabling disparate systems to communicate and share functionality. You've possibly used a number of REST, GraphQL, or GRPC APIs in your time. The quality of an API's design can significantly impact the development process, user experience, and long-term maintainability. In this chapter, we'll explore some details of API design, providing an introductory guide to creating effective and user-friendly APIs.



APIs are not only the building blocks for internal applications and tools, allowing different systems within an organization to work seamlessly together, *but they are often products themselves*. In many cases, the API is the primary way users interact with a company's services. This is especially true for companies that provide data services, payment processing, or other cloud-based functionalities.

For example, companies like Stripe and PayPal provide payment processing APIs that developers can integrate into their applications to handle transactions. Google Maps offers APIs for embedding maps and location-based services into web and mobile apps. Twilio provides communication APIs that enable developers to integrate SMS, voice, and video capabilities into their applications. In these instances, the API itself is the product that developers are purchasing and relying on to add critical functionality to their own applications. While good API design is crucial for internal services within an organization, it becomes especially critical when the API is the product or service itself, directly impacting customer satisfaction and business success.

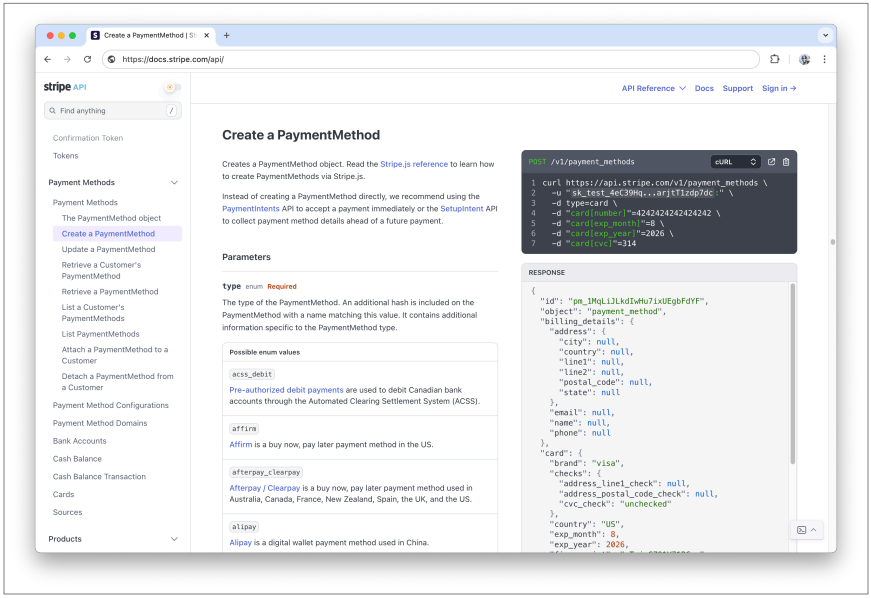


Figure 17-1. Stripe API Documentation

A well-designed API can lead to efficient development and robust applications, while a poorly designed one can become a maintenance

nightmare. The design of an API influences how easily it can be understood and used by developers, which directly affects productivity and the quality of the resulting applications.

Furthermore, time and effort spent on thoughtful design pay dividends in long-term stability and user satisfaction. Investing in the quality of your API from the start can prevent numerous issues down the line. This includes considering aspects such as clear and consistent naming conventions, comprehensive documentation, and thoughtful error handling.

In the next few sections, we'll discuss some strategies for creating user-centric APIs—APIs that are intuitive, easy to use, and designed with the developer experience in mind. By focusing on user-centric API design, we can create APIs that are not only functional but also enjoyable to use. This, in turn, fosters a positive developer experience, promotes adoption, and ensures long-term success for your API software projects.

The code examples we use in this chapter will be REST-based APIs, but the concepts we discuss transfer to any type of API you might be creating. Though building an API is done with server-side technologies (like Node.js) and not React, which is the focus of the book, understanding these principles is crucial when working with client-side React, as interacting with APIs is a common and necessary task.

Consistency

Consistency in API design is crucial for creating a seamless and intuitive experience for developers. When an API maintains consistency across its various components, it reduces the cognitive load on developers, making it easier for them to understand and work with the API. In this section, we'll delve into three key areas where consistency is paramount: naming conventions, resource structure, and response formats.

Naming Conventions

A clear and consistent naming convention helps developers understand and use an API more effectively. Consistent and intuitive naming makes an API self-documenting to a large extent, reducing the learning curve for developers and minimizing the chances of misunderstandings or

errors. When establishing naming conventions for an API, consider the following principles:

Use clear, descriptive names

Choose names that accurately describe the resource or action. Avoid abbreviations or cryptic shorthand that might confuse users.

- **Bad:** `/u`, `/p`
- **Good:** `/users`, `/products`

Stick to lowercase letters and hyphens

For multi-word resource names, use lowercase letters and hyphens. This improves readability and follows common URL conventions.

- **Bad:** `/orderItems`, `/ShippingAddresses`
- **Good:** `/order-items`, `/shipping-addresses`

Use nouns for resource names

Resources typically represent entities in a system, so use nouns to name them. This helps clarify that the resource is a thing, not an action.

- **Bad:** `/getCustomers`, `/createOrder`
- **Good:** `/customers`, `/orders`

Use verbs for actions that don't fit into CRUD operations

For operations that don't naturally map to HTTP methods (`GET`, `POST`, `PUT`, etc.), use verbs to describe the action. This helps distinguish these special actions from standard CRUD (Create, Read, Update, and Delete) operations.

- **Bad:** `/order-cancellation`, `/password-reset`
- **Good:** `/orders/{orderId}/cancel`, `/users/{userId}/reset-password`

Be consistent with pluralization

Choose either singular or plural for resource names and stick to it throughout the API. This consistency helps developers predict how to interact with different resources.

- **Bad:** (Mixed) `/customer`, `/orders`, `/product`
- **Good:** (All plural) `/customers`, `/orders`, `/products` OR
- **Good:** (All singular) `/customer`, `/order`, `/product`

Here's an example of how these naming conventions might be applied in a real-world API:

An example API that has customer, order, and product domains

```
GET /customers
POST /customers
GET /customers/{customerId}
PUT /customers/{customerId}
DELETE /customers/{customerId}
GET /customers/{customerId}/orders
POST /customers/{customerId}/orders
GET /orders/{orderId}
PUT /orders/{orderId}
DELETE /orders/{orderId}
POST /orders/{orderId}/cancel
GET /products
GET /products/{productId}
GET /products/{productId}/reviews
POST /products/{productId}/reviews
```

Resource Structure

The structure of API resources is another area where consistency can greatly enhance usability. A well-structured API organizes resources in a logical and intuitive manner, making it easier for developers to understand the relationships between different entities and how to navigate an API. When designing an API's resource structure, consider the following:

Use hierarchy to represent relationships

Nested resources can represent belongingness or composition. For example:

An example of nested resources

```
/customers/{customerId}/addresses
/orders/{orderId}/items
```

This structure clearly shows that addresses belong to customers and items belong to orders.

Keep URLs as short as possible while maintaining clarity

While nesting can be useful, avoid deep nesting that results in very long URLs. For instance:

A good and bad example of nested URLs

```
// Good
/orders/{orderId}/items

// Bad
/customers/{customerId}/orders/{orderId}/items/
{itemId}/variants
```

Use query parameters for filtering, sorting, and pagination

This keeps the base URL clean and allows for flexible querying. For example:

Using query parameters

```
GET /products?
category=electronics&sort=price&page=2&limit=20
```

Use consistent patterns for similar resources

If two resources have similar structures, maintain consistency in how they are represented. For instance:

Two resources with similar structures

```
GET /users/{userId}/profile
GET /companies/{companyId}/profile
```

Here's an example of how these principles might be applied to structure an e-commerce API:

An example e-commerce API with a consistent resource structure

```
/users
/users/{userId}
/users/{userId}/orders
/users/{userId}/addresses

/products
/products/{productId}
/products/{productId}/reviews

/orders
/orders/{orderId}
/orders/{orderId}/items
/orders/{orderId}/shipments

/categories
/categories/{categoryId}
/categories/{categoryId}/products
```

Response Formats

Consistent response formats ensure that developers know what to expect from the API. When responses follow a consistent structure, it becomes much easier for developers to parse and work with the data returned by an API. Here are some key considerations for maintaining consistency in response formats:

Use a consistent structure for all responses

Maintain a similar format for success and error responses. For example:

Example success and error responses

```
// Success response
{
  "status": "success",
  "data": {
    "id": 123,
    "name": "John Doe",
```

```
    "email": "john@example.com"
  }
}

// Error response
{
  "status": "error",
  "data": {
    "message": "User not found",
    "code": "NOT_FOUND"
  }
}
```

Use consistent field names across resources

If multiple resources have similar attributes, use the same names for them. For instance:

Using `createdAt` and `updatedAt` across different resources

```
// User
{
  "id": 123,
  "createdAt": "2023-07-01T12:00:00Z",
  "updatedAt": "2023-07-02T14:30:00Z",
  "name": "John Doe"
}

// Product
{
  "id": 456,
  "createdAt": "2023-06-15T09:00:00Z",
  "updatedAt": "2023-06-16T11:45:00Z",
  "name": "Smartphone X"
}
```

Use consistent date and time formats

Stick to a standard format like ISO 8601 for all date and time fields. For example:

ISO 8601 format used for different date fields

```
{
  "createdAt": "2023-07-01T12:00:00Z",
  "updatedAt": "2023-07-02T14:30:00Z",
  "scheduledFor": "2023-07-10T09:00:00+02:00"
}
```

Provide consistent metadata

Include metadata like pagination info or request identifiers consistently across all list responses. For example:

Metadata that includes pagination information

```
{
  "status": "success",
  "data": [
    // array of items
  ],
  "metadata": {
    "page": 2,
    "perPage": 20,
    "totalItems": 157,
    "totalPages": 8
  },
  "requestId": "req_abc123"
}
```

By adhering to these principles of consistency—naming conventions, resource structure, and response formats—we can create APIs that are easier to understand, use, and maintain. This consistency not only improves the developer experience but also enhances the overall reliability and usability of an API.

Error Handling

Error handling is a critical component of API design, as it directly affects the developer experience and usability of an API. Well-designed error responses help developers quickly identify and resolve issues, improving the overall developer experience. In this section, we'll cover some best practices for designing error handling in your API, including consistent error structure, meaningful error messages, and appropriate use of HTTP status codes.

Use appropriate HTTP status codes

HTTP status codes provide a standard way to communicate the outcome of a request. They should be used consistently and appropriately:

- 2xx for successful requests (e.g., 200 OK, 201 Created)
- 4xx for client errors (e.g., 400 Bad Request, 404 Not Found)
- 5xx for server errors (e.g., 500 Internal Server Error)

Provide detailed error messages

Error messages need to be concise and informative. They should provide enough context for developers to understand what went wrong and how to fix it. Be sure to avoid exposing sensitive information in error messages.

Clear error payload with details

```
{
  "status": "error",
  "message": "Invalid email format",
  "code": "INVALID_EMAIL",
  "details": "Provided email 'johndoe@example' is
missing a domain"
}
```

Use a consistent error response structure

Ensure all error responses follow a consistent format. This makes it easier for developers to parse and handle errors programmatically.

An error response structure

```
{
  "status": "error",
  "message": "string",
  "code": "string",
  "details": "string",
  "requestId": "string"
}
```

Include unique error codes

Where applicable, assign unique error codes to different types of errors. This allows developers to easily identify and handle specific error scenarios in their code.

A unique error code for insufficient funds

```
{
  "status": "error",
  "message": "Insufficient funds",
  "code": "INSUFFICIENT_FUNDS",
  "details": "Your account balance is $50, but the
transaction requires $100"
}
```

Handle validation errors

For requests with multiple fields, return all validation errors at once instead of one at a time. This saves developers time and reduces unnecessary API calls.

Validation errors for two separate form fields

```
{
  "status": "error",
  "message": "Validation failed",
  "code": "VALIDATION_ERROR",
  "errors": [
    {
      "field": "email",
      "message": "Invalid email format"
    },
    {
      "field": "password",
      "message": "Password must be at least 8
characters long"
    }
  ]
}
```

Log errors for debugging

While providing clear error messages to API consumers is important, it's important to ensure we're logging detailed error information server-side

for debugging purposes. This allows us to investigate and resolve issues more effectively without exposing sensitive information to API users. We can include details such as stack traces, request parameters, and system state in internal logs, but we'll need to be careful not to log sensitive data like passwords or API keys.

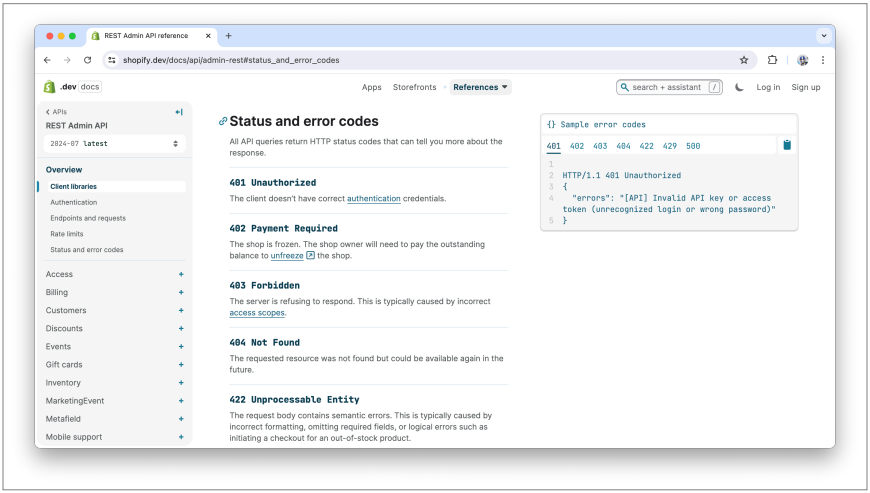


Figure 17-2. Shopify's REST Admin API — Status & Error Codes

Documentation

Comprehensive and well-organized documentation is essential for creating user-centric APIs. It serves as the primary resource for developers to understand how to use your API effectively. Good documentation can significantly reduce the learning curve, minimize support requests, and increase adoption rates. Let's explore key aspects to consider when creating documentation for an API.

Provide a Clear Overview

Start with a high-level overview of your API. This should include the purpose and main features of the API, authentication methods, base URL, versioning information, and common use cases or example scenarios. This overview gives developers a quick understanding of what your API offers and how it can be integrated into their projects.

For example, you might begin your documentation with something like:

"The Example E-commerce API allows developers to integrate our product catalog, order management, and customer data into their applications. The base URL for all API requests is https://api.example.com/v1, and we use OAuth 2.0 for authentication."

Detailed Endpoint Documentation

For each endpoint, provide comprehensive information. This should include the HTTP method and full URL, a description of the endpoint's purpose, request parameters (path, query, headers, body), response format, and possible status codes. Include example requests and responses to illustrate how the endpoint works in practice.

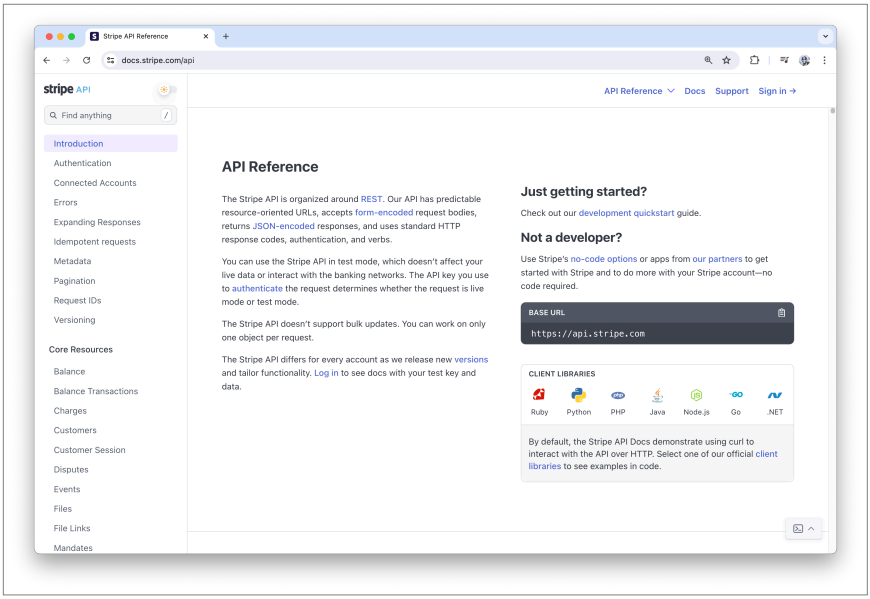


Figure 17-3. Stripe API reference/overview section

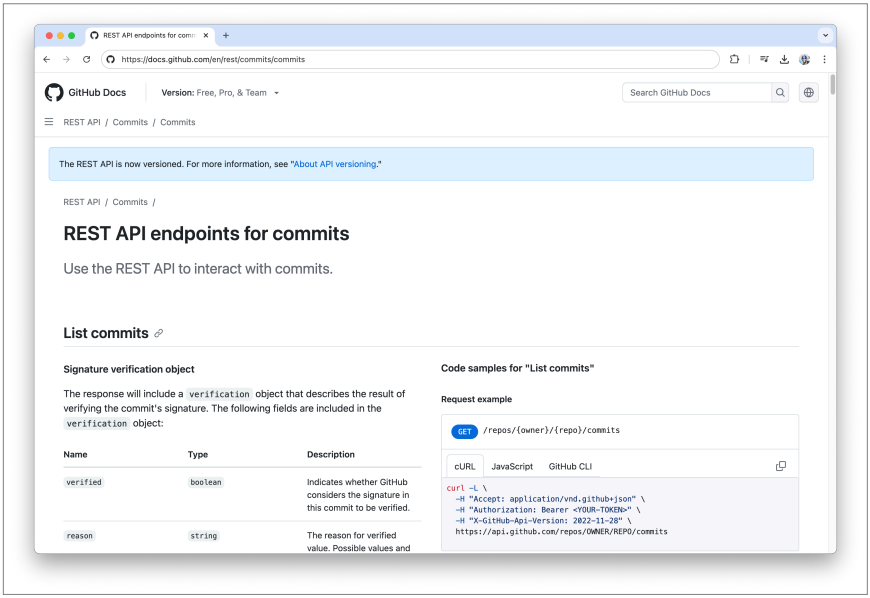
For instance, when documenting a "Get Product Details" endpoint, we'd include details such as the HTTP method (GET), the URL (`/products/{productId}`), a description of what the endpoint does, any parameters it accepts, and what the response looks like.

Include Code Samples

Provide code samples in popular programming languages to demonstrate how to use your API. This helps developers quickly understand how to integrate your API into their preferred language or framework. For example, you might include a JavaScript snippet showing how to make a request to your API using Node.js or a JavaScript client library like React.

Explain Authentication

Clearly explain how to authenticate with your API. This should include steps to obtain API keys or tokens, how to include authentication in requests, and security best practices. Proper authentication documentation ensures that developers can securely use your API from



the start.

Figure 17-4. GitHub's REST API endpoints for commits

Versioning

Versioning can be a critical aspect of API design, ensuring that changes and updates to your API can be managed without disrupting existing users. This is particularly important for APIs that serve as the core Strategiesproduct or service of a business, where maintaining stability and reliability for customers is crucial to the company's success and reputation.

A well-thought-out versioning strategy allows us to introduce new features and improvements while maintaining backward compatibility. In this section, we'll explore some best practices for API versioning, including different versioning strategies, how to implement them, and how to communicate changes to developers.

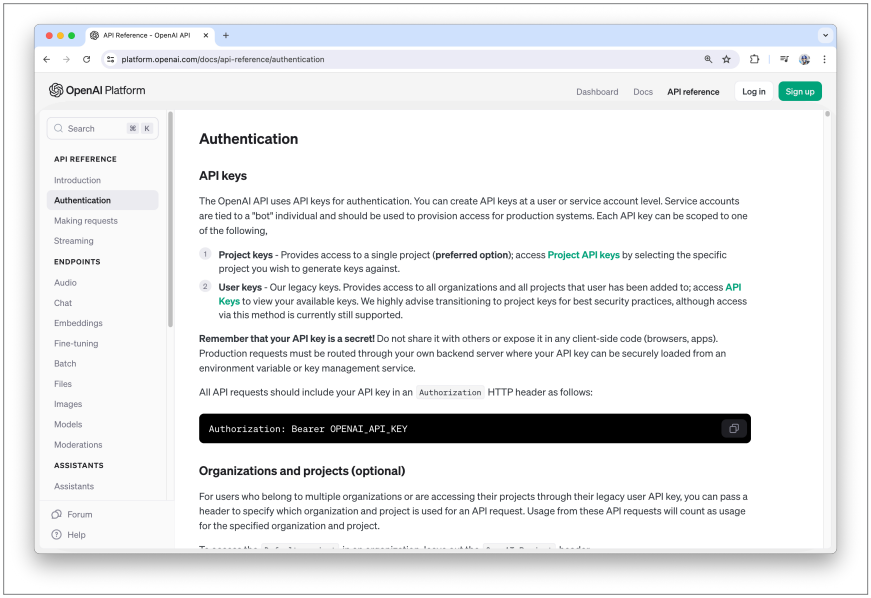


Figure 17-5. OpenAI API documentation on authentication

Versioning Strategies

There are several strategies for versioning an API, each with its pros and cons. The most common strategies include URL versioning, query parameter versioning, and header versioning.

URL Versioning

URL versioning involves including the version number in the API's base URL. This is one of the most straightforward and widely used versioning strategies.

Example of URL versioning

```
GET /v1/users/123
GET /v2/users/123
```

Pros:

- Easy to understand and implement
- Clear separation between different versions
- Allows for significant changes between versions

Cons:

- Can result in a large number of URLs
- Can be less flexible for minor changes

Query Parameter Versioning

Query parameter versioning involves specifying the API version as a query parameter in the request URL.

Example of query parameter versioning

```
GET /users/123?version=1
GET /users/123?version=2
```

Pros:

- Keeps the base URL clean
- Easy to implement and use
- Allows for fine-grained version control

Cons:

- Can be overlooked by developers
- May complicate caching strategies
- Less clear separation between major versions

Header Versioning

Header versioning involves specifying the API version in a custom HTTP header.

Example of header versioning

```
GET /users/123
Header: API-Version: 1
```

Pros:

- Keeps URLs clean and simple
- Separates versioning concerns from the resource identifier

Cons:

- Can be less discoverable
- May be more difficult to test in a browser
- Requires custom header handling on the server

Choosing the appropriate versioning strategy depends on your specific needs and the nature of your API. URL versioning is often preferred for its simplicity and clarity, while query parameter and header versioning can offer more flexibility in certain scenarios.

As a simple example of URL versioning, let's consider an example where we're evolving a products API:

GET /v1/products

```
// GET /v1/products
// Response:
{
  "data": [
    { "id": 1, "name": "Product A" },
    { "id": 2, "name": "Product B" }
  ]
}
```

Version 2 (changes the 'id' and 'name' fields):

GET /v2/products

```
// GET /v2/products
// Response:
{
  "products": [
    { "productId": 1, "productName": "Product A" },
    { "productId": 2, "productName": "Product B" }
  ]
}
```

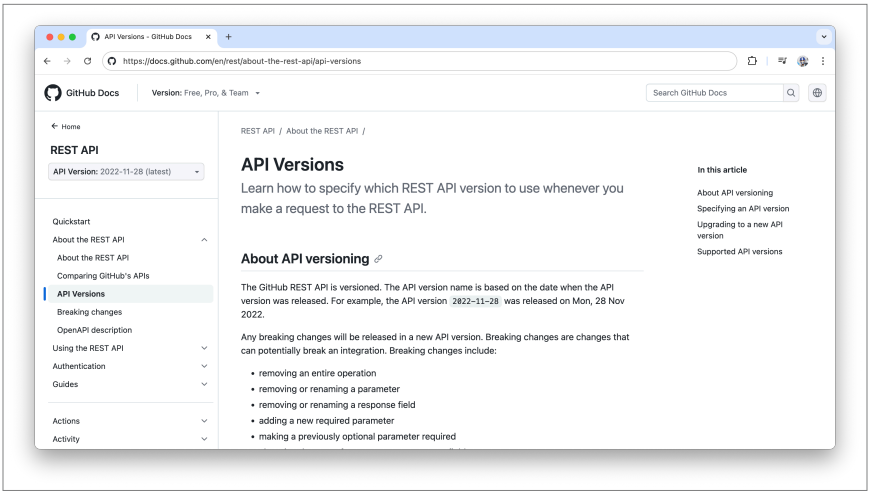


Figure 17-6. Github REST API documentation on API Versions

Deprecation and Sunsetting

When introducing new versions, it's important to have a clear policy for deprecating and eventually sunsetting old versions. This can include but is not limited to:

6. **Announcing the deprecation**: Clearly communicate when a version will be deprecated and eventually removed.

7. **Providing a timeline**: Give users ample time to migrate to newer versions.

8. **Sending deprecation warnings**: If possible, include deprecation warnings in API responses for outdated versions.

9. **Offering migration support**: Provide documentation, tools, and support to help users migrate to the new version.

By implementing a thoughtful versioning strategy, we can evolve an API over time while maintaining a positive developer experience for existing users. This approach allows us to innovate and improve an API without fear of breaking existing integrations.

Security

Security is an important aspect of API design. A well-designed API must not only be functional and user-friendly but also secure. Implementing robust security measures protects both the API provider and its users from potential threats and vulnerabilities. Given the breadth of topics under API security, in this section, we'll only highlight some key points.

Authentication and Authorization

Authentication and authorization are foundational elements of API security.

Authentication verifies the identity of a user or service, confirming that the entity requesting access is who they claim to be. This process is crucial for maintaining the security of your API by ensuring that only legitimate users can access your data or functionalities. Authentication can be implemented in several ways depending on the complexity and requirements of your application:

- API keys for simple applications.
- OAuth 2.0 for more complex scenarios requiring delegated access.
- JSON Web Tokens (JWT) for stateless authentication.

Here's an example of a simple authentication endpoint using JWT in Node.js. The endpoint handles user login requests by verifying credentials and generating a JWT (i.e., JSON Web Token) if the credentials are valid:

A simple authentication endpoint using JWT in Node.js

```
// Authentication endpoint
app.post("/login", (req, res) => {
  const { username, password } = req.body;

  // Check if user exists and password is correct
  const user = findUserByUsername(username);
  if (
    user &&
    verifyPassword(
      password,
      user.hashedPassword,
    )
  ) {
    // Generate and send JWT token
    const token = generateJWT(user);
    res.json({ token });
  } else {
    res
      .status(401)
      .json({ error: "Invalid credentials" });
  }
});
```

Once authenticated, **authorization** can be implemented to ensure that a user or service has the correct permissions to perform actions or access data. Here's a follow-up pseudo-code example of an API endpoint that authorizes a user:

Protected route with authorization in Node.js

```
// Protected route that requires authorization
app.get(
  "/api/protected-resource",
  authenticateToken,
  (req, res) => {
    // Check if user has necessary permissions
    if (req.user.role === "admin") {
      // Provide access to protected resource
      res.json({
        data: "This is sensitive information",
      });
    } else {
      res
        .status(403)
```

```
        .json({
          error: "Insufficient permissions",
        });
    }
  },
);

// Middleware to authenticate JWT token
function authenticateToken(req, res, next) {
  const token = req.headers["authorization"];

  if (!token)
    return res
      .status(401)
      .json({ error: "No token provided" });

  verifyJWT(token, (err, user) => {
    if (err)
      return res
        .status(403)
        .json({ error: "Invalid token" });
    req.user = user;
    next();
  });
}
```

This code example demonstrates a protected route that requires both authentication and authorization. The `authenticateToken()` middleware first verifies the JWT token from the request headers. If the token is valid, it adds the user information to the request object. The route handler then checks the user's role to determine if they have the necessary permissions to access the protected resource. If the user is an admin, they are granted access; otherwise, they receive an error message indicating insufficient permissions.

Security Headers

Implementing security headers helps protect your API from certain types of attacks, such as clickjacking, cross-site scripting, and other cross-site injections. These headers include but are not limited to headers like Content-Security-Policy, Content-Type, and Strict-Transport-Security.

Setting security headers in Express.js

```
app.use((req, res, next) => {
  res.setHeader(
    "Content-Security-Policy",
    "default-src 'self';",
  );

  res.setHeader(
    "Content-Type",
    "application/json",
  );

  res.setHeader(
    "Strict-Transport-Security",
    "max-age=63072000; includeSubDomains; preload",
  );

  // ...
  // ...
});
```

Input Validation and Sanitization

Beyond implementing security headers, it's important to validate and sanitize user input to guard against injection attacks and other malicious activities. Effective validation ensures that incoming data adheres to expected types, formats, and value ranges. Sanitization involves stripping or transforming special characters to prevent them from triggering harmful behavior in your application.

Below is a simple example demonstrating input validation for a user registration endpoint:

Input validation example in Node.js

```
app.post("/register", (req, res) => {
  const { username, email, password } =
    req.body;

  // Validate input
  if (
    !isValidUsername(username) ||
    !isValidEmail(email) ||
    !isStrongPassword(password)
```

```
  ) {
    return res
      .status(400)
      .json({ error: "Invalid input" });
  }

  // Sanitize input
  const sanitizedUsername =
    sanitizeString(username);
  const sanitizedEmail = sanitizeEmail(email);

  // Proceed with user registration
  // ...
});

function isValidUsername(username) {
  return /^[a-zA-Z0-9_]{3,20}$/.test(username);
}

function isValidEmail(email) {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(
    email,
  );
}

function isStrongPassword(password) {
  return (
    password.length >= 8 &&
    /[A-Z]/.test(password) &&
    /[a-z]/.test(password) &&
    /[0-9]/.test(password)
  );
}
```

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a security feature that allows or restricts resources on a web page to be requested from another domain outside the domain from which the first resource was served. This is crucial for modern web applications that interact with APIs hosted on different domains.

A simple example of setting up CORS in a Node/Express.js app

```
const cors = require("cors");

// Configure CORS options
const corsOptions = {
  origin: "https://example.com", // Allow this domain
  optionsSuccessStatus: 200, // For legacy support
};

app.use(cors(corsOptions));

app.get("/api/data", (req, res) => {
  res.json({
    message:
      "This is data accessible from example.com.",
  });
});
```

Rate Limiting

Rate limiting can be implemented to protect an API from abuse and potential denial-of-service attacks. This involves setting limits on the number of requests a client can make within a specified time frame.

Rate limiting example in Node.js

```
const rateLimit = require("express-rate-limit");

const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per
windowMs
  message:
    "Too many requests from this IP, try again later",
});

// Apply rate limiting to all requests
app.use(apiLimiter);
```

Security is a critical aspect of API design that should never be overlooked. In addition to some of the foundational security practices we've discussed, it's essential to always use HTTPS to secure data in transit and apply these security measures to protect your API and its users from potential threats and vulnerabilities.

The topics covered here are just a starting point—API security is a broad field with many more aspects to consider. A valuable resource for further exploration is the OWASP Cheat Sheet Series, particularly the REST Security Cheat Sheet and GraphQL Cheat Sheet, which provides in-depth best practices for securing your REST and GraphQL APIs.

Stakeholder Engagement

Developing a user-centric API isn't just about technical considerations; it's also about understanding and meeting the needs of stakeholders. Effective stakeholder engagement is crucial for creating an API that truly serves its purpose and satisfies its users.

Start by identifying all relevant stakeholders. This typically includes internal teams (such as product managers, developers, and customer support), external developers who will be using the API, and potentially end-users of the applications that will integrate your API. Each of these groups will have different perspectives and requirements that need to be considered.

Gather feedback early and often. Before finalizing your API design, share drafts of your API specification with key stakeholders. This could involve creating mock endpoints or interactive API documentation that allows stakeholders to explore and test the proposed API structure. Tools like Swagger or Postman can be valuable for this purpose.

Pay close attention to the needs of external developers. Consider setting up a developer relations program to maintain ongoing communication with your API users. This could include developer forums, regular feedback sessions, or even a beta program for testing new API features before they're widely released.

Internal alignment is equally important. Ensure that your API design aligns with your company's overall product strategy and technical roadmap. Regular check-ins with product managers and other internal teams can help ensure that the API is evolving in a direction that supports broader business goals.

Remember that stakeholder engagement is an ongoing process. As your API evolves, continue to seek feedback and be prepared to make adjustments based on real-world usage and changing requirements. This

iterative approach helps ensure that your API remains valuable and relevant over time.

Final Considerations

While we've covered some key aspects of user-centric API design, there are several other important considerations to keep in mind when building and working with APIs. These include optimizing performance, setting up analytics and monitoring, and more.

It's important to remember that creating a user-centric API is an ongoing process that requires continuous adaptation to emerging trends, user feedback, and evolving business needs. By staying informed and being responsive to these changes, you can ensure that your API remains relevant, effective, and enjoyable for developers to use.

Additional reading

- API design — Postman
- Designing for humans: better practices for creating user-centric API experiences — Postman
- Designing Good API Experiences (Video) — Postman
- RESTful web API design — Azure Architecture Center
- Best Practices in API Design — Swagger

The Future of React

As we've journeyed through the book, we've explored an array of tools and techniques integral to managing large-scale React JavaScript applications. We've come to see how these methodologies aren't just theoretical; they're rigorously used and highly valued in the industry today.

React, which emerged in 2013, revolutionized the way developers build user interfaces. Its declarative component-based model provided an efficient and intuitive way to construct complex, dynamic web applications. Over the years, React has grown exponentially, not just in popularity but in its capabilities, evolving into a comprehensive ecosystem that supports a vast network of tools, libraries, systems, and applications across the globe.

Today, we find ourselves at a pivotal moment in React's evolution since significant updates and changes are on the horizon for the React ecosystem.

While this book focuses on covering industry standards and best practices, it would be incomplete without addressing these future shifts in React's environment. Therefore, in this chapter, we'll aim to unpack some of these upcoming changes. We'll delve into what these developments mean for you as a developer, for the industry, and how they might influence the way we think about React application design and development moving forward.

What's changing?

At the start of 2024, Andrew Clark, a core team member of React, posted a succinct post on X (formerly Twitter), indicating that by year's end, several React APIs and patterns familiar to developers were slated to become obsolete or undergo significant changes.

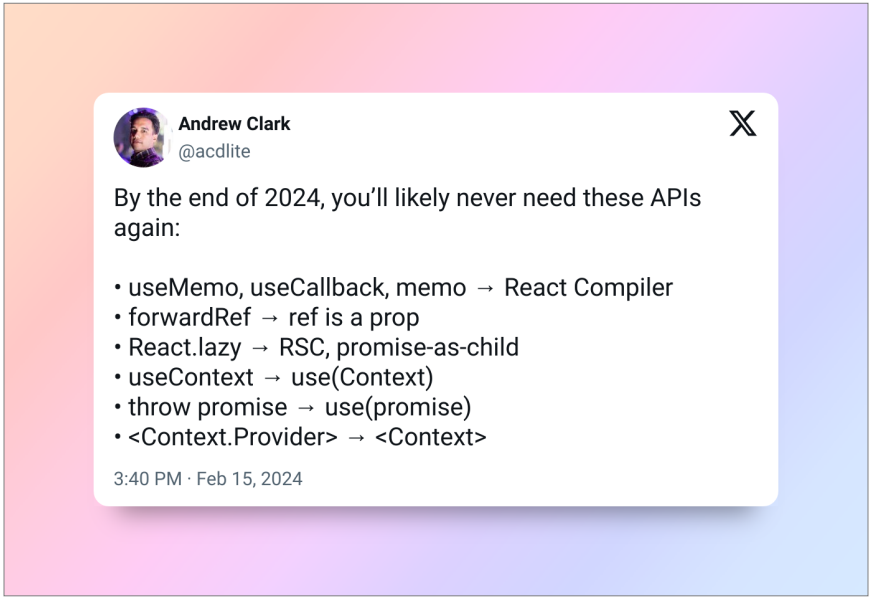


Figure 18-1. Post by @acdlite on X/Twitter on changes to React

We think it's helpful to group these changes into three different categories:

- New Hooks & APIs
- React Compiler
- React Server Components

For the rest of the chapter, we'll delve deeper into each of these categories.

New Hooks & APIs

Hooks and APIs within React have provided us with a robust framework for managing state and effects in functional components. Today, React has introduced new functionalities that change and improve how we build our React components in certain capacities. Instead of listing each new Hook and API one by one, we'll go through this section by discussing how certain features are traditionally built and how these new Hooks and APIs change the way we build, allowing for more efficient, readable, and maintainable code.

Async form submissions

A large part of how we use web applications today involves interacting with forms for various purposes, from user authentication to data submission to e-commerce transactions, feedback collection, search queries, and much more. As a result, a very common behavior in React component development involves making form submissions of a sort and handling the asynchronous updates of what the form should do when submitted. This involves managing local state changes, validating user input, and implementing logic to handle various states of submission, including loading, success, and error states.

In a previous work setting, an engineering manager used to share with my team and me how 90% of the work we did revolved around capturing user data and submitting forms. This routine yet important aspect of our projects underscored the importance of efficient form management and the powerful role React played in optimizing this process. **[Hassan Djirdeh]**

Let's walk through a traditional example of a simple component that submits a form and handles an asynchronous update when the form is submitted. We'll assume that there exists a `submitForm()` function that submits form information to a server and returns a response. The component will manage two state properties, `formState` and `isPending` that is used to convey to the user the current status of the form submission, providing feedback on whether the data is being processed or if the submission has been completed successfully or encountered an error.

Simple component for asynchronous form submission handling

```
import React, { useState, useCallback } from 'react'

const submitForm = async () => {/* form submit */}

export function Component() {
  // create form state
  const [formState, setFormState] = useState(null)
  const [isPending, setIsPending] = useState(false)

  // handle form submission
  const formAction = useCallback(async (event) => {
   // ...
  }, [])

  // display form template
  return (
    <form onSubmit={formAction}>
      {/* Form Template */}
    </form>
  )
}
```

When the form is submitted, the component `formAction()` method is triggered and this is where we can set the pending state, submit the form, and finally update the form state based on the response or if error occurs.

Handling state changes when form is submitted

```
import React, { useState, useCallback } from "react";

const submitForm = async () => {
  /* form submit */
```

```
};

export function Component() {
  // create form state
  const [formState, setFormState] = useState(null);
  const [isPending, setIsPending] = useState(false);

  // handle form submission
  const formAction = useCallback(async (event) => {
    event.preventDefault();
    setIsPending(true);
    try {
      const result = await submitForm();
      setFormState(result);
    } catch (error) {
      setFormState({
        message: "Failed to complete action",
      });
    }
    setIsPending(false);
  }, []);

  // display form template
  return (
    <form onSubmit={formAction}>
     {/* Form Template */}
    </form>
  );
}
```

When looking at this code and the expected UI changes that take place, we can understand this UI behavior to be a *transition*—an update that transition the UI from one view to another in a non-urgent manner. With some of the newest changes in React, React now supports using async functions in transitions where the useTransition Hook can be leveraged to manage the rendering of loading indicators or placeholders during asynchronous data fetching.

When leveraging the `useTransition` Hook, we can mark state updates that might take time to complete. In this setting, we don't have to create and manage our own pending/loading state property since the `useTransition` Hook provides a 'pending' flag to indicate whether the transition is still in progress.

Handling async updates with the useTransition Hook

```
import { useState, useTransition } from "react";

const submitForm = async () => {
  /* form submit */
};

export function Component() {
  const [formState, setFormState] = useState(null);
  const [isPending, startTransition] =
    useTransition();

  const formAction = (event) => {
    event.preventDefault();

    startTransition(async () => {
      try {
        const result = await submitForm();
        setFormState(result);
      } catch (error) {
        setFormState({
          message: "Failed to complete action",
        });
      }
    });
  };

  return (
    <form onSubmit={formAction}>
      {/* Form Inputs */}

      {/* ... */}

      {isPending ? <h4>Pending...</h4> : null}
      {formState?.message && (
        <h4>{formState.message}</h4>
      )}
    </form>
  );
}
```

With the recent improvements in React, the concept of transitions is taken a step further as functions that use async transitions are now

referred to as **Actions**. There now exist a few specialized Hooks to manage Actions and the first we'll take a look at is the **useActionState** Hook.

useActionState Hook

The `useActionState` Hook is a new Hook in React that allows us to update state based on the result of a form action. The Hook accepts three parameters:

1. An "action" function, which is executed when the form action is triggered.

2. An initial state object, that sets the starting state of the form before any user interaction.

3. [Optional] A permalink that refers to the unique page URL that this form modifies.

And returns three values in a tuple:

1. The current state of the form.

2. A function to trigger the form action.

3. A boolean indicating whether the action is pending.

useActionState Hook signature

```
import { useActionState } from "react";

export function Component() {
  const [state, dispatch, isPending] = useActionState(
    action,
    initialState,
    permalink,
  );

  // ...
}
```

The `action` function, provided as the first argument to the `useActionState` Hook, is activated upon form submission. It

determines the anticipated form state transition and whether the submission succeeds or fails due to errors. This function takes two parameters: the current state of the form and the form data at the time the action is initiated.

Let's revisit the form example we discussed earlier. We can instead create an action that triggers the `submitForm()` function that subsequently triggers an API call to submit the form data to a server. When the action is successful, it returns a form state object representing the next state of the form. When the action fails, it returns a form state object reflecting the error state, possibly including error messages or indicators to guide the user in correcting the issue.

Creating the action function that handles form submit

```
import { useActionState } from "react";

const submitForm = async () => {
  /* form submit */
};

const action = async (currentState, formData) => {
  try {
    const result = await submitForm(formData);
    return { message: result };
  } catch {
    return { message: "Failed to complete action" };
  }
};

export function Component() {
  const [state, dispatch, isPending] = useActionState(
    action,
    null
  );

  // ...
}
```

With our `useActionState()` Hook set-up like the above, we're then able to use the form `state`, `dispatch`, and `isPending` values in our form template.

<form> actions

`<form>` elements now have an `action` prop that can receive an action function that is triggered when a form is submitted. Here is where we'll pass down the `dispatch` function from our `useActionState()` Hook.

Passing dispatch to the <form> action prop

```
import { useActionState } from "react";

const submitForm = async () => {
  /* form submit */
};

const action = async (currentState, formData) => {
  try {
    const result = await submitForm(formData);

    return { message: result };
  } catch {
    return { message: "Failed to complete action" };
  }
};

export function Component() {
  const [state, dispatch, isPending] = useActionState(
    action,
    null
  );

  return (
    <form action={dispatch}>
      {/* ... */}
    </form>
  )
}
```

We can display the `state` somewhere in our form template and use the `isPending` value to convey to the user when the async action is in flight.

Displaying form state and isPending status in the template

```
import { useActionState } from "react";

const submitForm = async () => {
  /* form submit */
};
```

```
const action = async (currentState, formData) => {
  try {
    const result = await submitForm(formData);

    return { message: result };
  } catch {
    return { message: "Failed to complete action" };
  }
};

export function Component() {
  const [state, dispatch, isPending] = useActionState(
    action,
    null,
  );

  return (
    <form action={dispatch}>
      <input
        type="text"
        name="text"
        disabled={isPending}
      />

      <button type="submit" disabled={isPending}>
        Add Todo
      </button>

      {/*
        display form state message to convey when
        form submit is successful or fails.
      */}
      {state.message && <h4>{state.message}</h4>}
    </form>
  );
}
```

Voila! With these new changes to React, we no longer need to handle pending states, errors, and sequential requests manually when working with async transitions in forms. Instead, these values are accessed directly from the `useActionState()` Hook!

useFormStatus Hook

Traditionally, we've always relied on the Context API to propagate state or data from parent components to deeply nested child components. As we've seen in earlier chapters in the book, this is particularly useful in scenarios where multiple components need access to shared data without passing props manually at every level.

However, when it comes to forms, React has introduced a new Hook titled **useFormStatus** specifically designed to access status information from form submissions within nested components. This Hook functions similarly to how a context provider would, allowing child components within a form to directly access submission status data.

Accessing parent form submit status with useFormStatus Hook

```
import { useFormStatus } from "react";

export function NestedComponent() {
 /* access last form submission information */
 const { pending, data, method, action } =
  useFormStatus();

  return (
    /* ... */
  )
}
```

`useFormStatus` must be called from a component that is rendered inside a `<form>` and will only return status information from that parent `<form>`. Though the Context API can still be used to pass down form status and other data, `useFormStatus` simplifies form status handling by eliminating the need for manual context setup and streamlining state access directly related to forms.

Optimistic updates

Optimistic updates are a user experience enhancement technique that assumes success of an async operation and updates the UI in advance of the successful confirmation of the operation. **useOptimistic** is a new Hook in React that facilitates this pattern by enabling us to implement optimistic updates easily.

Assume we had a component designed to update a status message after an async call is made when a form is submitted:

Component that updates message prop after form submit

```
import React, { useState } from "react";

export function Component({
  message,
  updateMessage,
}) {
  const [inputMessage, setInputMessage] =
    useState("");

  const submitForm = async (event) => {
    event.preventDefault();
    const newMessage = inputMessage;

    // Update state when API submission resolves
    const updatedMessage =
      await submitToAPI(newMessage);

    updateMessage(updatedMessage);
  };

  return (
    <form onSubmit={submitForm}>
      {/* Show current message */}
      <p>{message}</p>

      <input
        type="text"
        name="text"
        value={inputMessage}
        onChange={(e) =>
          setInputMessage(e.target.value)
        }
      />
      <button type="submit">Add Message</button>
    </form>
  );
}
```

In the above simple form example, let's further assume we wanted to visually confirm to the user that their message update was accepted the moment they hit submit without waiting for the server to respond. This is

where the `useOptimistic()` Hook comes into play, providing a way for us to immediately reflect the presumed successful outcome in the UI.

Optimistically updating template while async form submit is in flight

```
import React, {
  useState,
  useOptimistic,
} from "react";

export function Component({
  message,
  updateMessage,
}) {
  const [inputMessage, setInputMessage] =
    useState("");

  const [
    optimisticMessage,
    setOptimisticMessage,
  ] = useOptimistic(message);

  const submitForm = async (event) => {
    event.preventDefault();
    const newMessage = inputMessage;

    /* before triggering API change,
       set value optimistically */
    setOptimisticMessage(newMessage);

    // Update state when API submission resolves
    const updatedMessage =
      await submitToAPI(newMessage);
    updateMessage(updatedMessage);
  };

  return (
    <form onSubmit={submitForm}>
      {/* show optimistic value */}
      <p>{optimisticMessage}</p>

      <input
        type="text"
        name="text"
```

```
          value={inputMessage}
          onChange={(e) =>
            setInputMessage(e.target.value)
          }
        />
        <button type="submit">Add Message</button>
      </form>
    );
}
```

In the above example, when the user submits the form by clicking the "Add Message" button, the `submitForm()` function is triggered. Before initiating the API request to update the message, the `setOptimisticMessage()` function is called with the new message value obtained from the form data. This immediately updates the UI to reflect the optimistic change, providing the user with instant feedback.

When the update finishes or errors, React will automatically switch the value of `optimisticMessage` used in the template back to the `message` prop value.

The use API

use is a new React API that provides a versatile way to read values from resources like Context or Promises.

In earlier sections of the book (chapter—**State Management**), we explored how context in React provides a way to share values between components without having to explicitly pass a prop through every level of the tree.

To begin working with the Context API, we often create a context object and then use the context Provider to wrap our components, making the data available to all nested components. A recent change in React now allows us to render `<Context>` directly as a provider instead of using `<Context.Provider>`.

Passing data with context in React (Parent Component)

```
import React, {
  useState,
  createContext,
```

```
} from "react";

// Create a context
const MessageContext = createContext();

function App() {
  const [message, setMessage] = useState(
    "Hello World!",
  );

  return (
    // Provide the state to nested components
    <MessageContext
      value={{ message, setMessage }}
    >
      <Child1>
        <Child2>
          <Child3>
            <DeeplyNestedChild />
          </Child3>
        </Child2>
      </Child1>
    </MessageContext>
  );
}
```

To read context values, we no longer need to use the `useContext()` Hook and can simply pass the context to `use()`. The `use()` function traverses the component tree to find the closest context provider.

Reading context with the use (Child Component)

```
import { use } from "react";
import { MessageContext } from "./context";

function DeeplyNestedChild() {
  // Access the data with the new use() API
  const { message, setMessage } = use(
    MessageContext
  );

  return (
    <div>
      <h1>{message}</h1>
```

```
      <button
        onClick={() =>
          setMessage(
            "Hello from nested component!",
          )
        }
      >
        Change Message
      </button>
    </div>
  );
}
```

Unlike the `useContext()` Hook to read context, the `use()` function can also be used within conditionals and loops in our components!

Reading context within a conditional (Child Component)

```
import { use } from "react";
import { MessageContext } from "./context";

function DeeplyNestedChild({ value }) {
  let message;
  let setMessage;

  // Access context data in a conditional
  if (value) {
    const context = use(MessageContext);
    message = context.message;
    setMessage = context.setMessage;
  }

  return (
    <div>
      {message && <h1>{message}</h1>}
      {setMessage && (
        <button
          onClick={() =>
            setMessage(
              "Hello from nested component!",
            )
          }
        >
          Change Message
```

```
        </button>
      )}
    </div>
  );
}
```

`use()` also integrates seamlessly with `Suspense` and error boundaries to read promises. We'll discuss this point in more detail in the upcoming section—**React Server Components**.

React Compiler

React Compiler is an experimental compiler introduced by the React team aimed at significantly improving the performance of React applications by **automating the optimization process**. The introduction of the React Compiler underscores a move towards a paradigm where the framework itself takes on the responsibility of (some) performance tuning, rather than leaving it solely to developers.

Memoization

React's design around a declarative model for building user interfaces marked a significant shift in UI development. By abstracting the direct manipulation of the DOM in favor of a component-based architecture, React allows developers to think about UIs as a reflection of state, not a sequence of imperative updates. This model greatly simplifies the mental model for developers, especially as applications grow in complexity.

Consider the following example, which shows a typical React component used to display a list of todos from an API:

TodoList React Component

```
import React, {
  useState,
  useEffect,
} from "react";

const TodoList = () => {
  const [todos, setTodos] = useState([]);
```

```
  useEffect(() => {
    const fetchTodos = async () => {
      try {
        const response = await fetch(
          "https://dummyjson.com/todos",
        );
        const data = await response.json();
        setTodos(data.todos);
      } catch (error) {
        console.error(
          "Error fetching todos:",
          error,
        );
      }
    };

    fetchTodos();
  }, []);

  return (
    <div>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>{todo.todo}</li>
        ))}
      </ul>
    </div>
  );
};

export default TodoList;
```

The `TodoList` component is a React function component designed to fetch and display a list of todo items from an API. It utilizes the `useState` Hook to manage the todos' state and the `useEffect` hook to perform the fetch operation when the component mounts initially. React's re-rendering capabilities ensure that the UI for the component above always reflects the current state.

The above component example is simple; however, as a React application scales and begins to manage large datasets or perform expensive computations, the default rendering behavior of components can sometimes lead to performance bottlenecks. React's rendering model

re-renders the entire component subtree whenever state changes, which is highly efficient for small components and datasets but can become costly with larger datasets or complex UIs.

Before delving deeper into optimizations, it's important to understand memoization. Memoization is an optimization technique used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

In the context of React, memoization helps avoid unnecessary calculations and re-rendering by caching complex functions or components when their inputs have not changed. In the `TodoList` component example we shared above, several pieces of the code can benefit from memoization.

For example, the data fetching function in `useEffect` doesn't need to run again once the todos have been initially fetched unless there's a specific reason to refetch the data (e.g., refreshing data). This effect is inherently "memoized" by the empty dependency array in `useEffect`, ensuring it only runs once after the component mounts.

Empty dependency array in useEffect

```
useEffect(() => {
  const fetchTodos = async () => {
    // ...
  };

  fetchTodos();
  /*
    empty dependency array ensures this effect
    only runs once after the component is
    mounted.
  */
}, []);
```

As the todo list of items grows, rendering each todo item can become a performance concern, especially if the list items include more complex rendering logic. Here, we could use the memo function to ensure that individual `TodoItem` components only re-render when their props change, not every time the parent `TodoList` component re-renders.

Using React.memo to memoize component functions

```
import React, {
  useState,
  useEffect,
  memo,
} from "react";

const TodoItem = memo(({ todo }) => {
  return <li>{todo}</li>;
});

const TodoList = () => {
  // ...
  // ...

  return (
    <div>
      <ul>
        {todos.map((todo) => (
          <TodoItem
            key={todo.id}
            todo={todo.todo}
          />
        ))}
      </ul>
    </div>
  );
};

export default TodoList;
```

While React's `memo` function is used to prevent unnecessary re-renders of components, there exists a useMemo Hook that helps in memoizing any computed values within a component. `useMemo` returns a memoized value and only recalculates it when one of its dependencies changes. As an example, assume we wanted to display the count of completed todos in our `TodosList` component. If computing the count of completed todos becomes computationally expensive as the list grows, we can use the `useMemo` Hook to ensure the computation of the number of completed todos is recalculated only when there is an actual change in the `todos` array.

Using the useMemo Hook to memoize a component value

```
import React, {
```

```
  useState,
  useEffect,
  useMemo,
} from "react";

const TodoList = () => {
  const [todos, setTodos] = useState([]);

  // ...
  // ...

  const completedCount = useMemo(() => {
    return todos.filter(
      (todo) => todo.completed,
    ).length;
  }, [todos]);

  return (
    <div>
      {/* ... */}
      <div>
        Completed Todos: {completedCount}
      </div>
    </div>
  );
};

export default TodoList;
```

Lastly, just like there's a `useMemo` Hook to memoize computed values within a component, there also exists a useCallback Hook to memoize function references within a component.

Automatic memoization with React Compiler

Historically, React developers have relied on `React.memo`, `useMemo`, and `useCallback` to prevent unnecessary re-renders and computations of components and values/functions within our components. These techniques, while effective, require us to manually determine what to memoize and when. This manual process is not only time-consuming but also prone to mistakes, leading to either over-optimization or insufficient memoization, both of which can degrade performance.

This is where React Compiler steps in. By understanding the "Rules of React" and leveraging advanced static analysis, the compiler intelligently applies memoization across the components and Hooks of a React application.

The best part is that React Compiler **automates** the decision-making process about what to memoize! This automation removes the burden from us, minimizing the risk of human error and the overhead associated with manual optimizations. This leads to optimized performance with minimal effort, ensuring that memoization is applied precisely and efficiently where it's most impactful.

For the `TodoList` component example, this means that the React Compiler can automatically determine when the `TodoItem` child components or computed values like `completedCount` need to be updated without explicit use of memoization functions.

Removing the use of the memo and useMemo functions in `TodoList`

```
import React, {
  useState,
  useEffect,
} from "react";

// we don't need memo()
const TodoItem = ({ todo }) => {
  return <li>{todo}</li>;
};

const TodoList = () => {
  const [todos, setTodos] = useState([]);

  useEffect(() => {
    // ...
  }, []);

  // we don't need useMemo()
  const completedCount = todos.filter(
    (todo) => todo.completed,
  ).length;

  return (
    <div>
      <ul>
```

```
          {todos.map((todo) => (
            <TodoItem
              key={todo.id}
              todo={todo.todo}
            />
          ))}
        </ul>

        <div>
          Completed Todos: {completedCount}
        </div>
      </div>
    );
};

export default TodoList;
```

By analyzing the component's usage of state and props, the React Compiler can optimize re-renders effectively. It identifies that changes to `todos` only affect specific parts of the DOM and can intelligently decide to re-render only those components that depend on the parts of `todos` that actually changed. In the React documentation, this is sometimes referred to as "fine-grained reactivity".

Memoization of external functions

React Compiler not only automates memoization for React components by skipping unnecessary re-renders of a component tree, but also addresses the memoization of expensive external functions that are used within components. For example, assume our `TodoList` component depends on an external `getPriorityTodos()` function that calculates and returns a list of priority todos based on some criteria, which is a computationally expensive task.

Using a hypothetical expensive function in `TodoList`

```
import React from "react";

// function itself is not memoized by React Compiler
const getPriorityTodos = (todos) => {
  // ...
};
```

```
const TodoList = () => {
  // ...

  // function call is memoized by React Compiler
  const priorityTodos = getPriorityTodos(items);

  return <div>{/* ... */}</div>;
};

export default TodoList;
```

React Compiler will memoize the call to `getPriorityTodos()` within the `TodoList` component, ensuring that it re-computes only when necessary (based on its input changes).

Keep in mind that this optimization is specifically tied to the usage of functions within the component. While React Compiler memoizes the call to `getPriorityTodos()`, it does not memoize the function itself. The React documentation notes that if an expensive function is expected to be used in many different components, it may be worth implementing its own memoization to ensure across-the-board efficiency regardless of the specific usage context.

Memoization of deps used in useEffect

Earlier in this section, we briefly discussed the role of memoization within the context of `useEffect` and the dependencies it uses. Traditionally, it's helpful to memoize the dependencies used in the `useEffect` Hook to ensure that the effect only reruns when absolutely necessary. However, the React Compiler does not handle the automatic memoization of dependencies in `useEffect` and this area remains a topic of ongoing research and development (see Introducing React Compiler).

Key assumptions for the Compiler

While the Compiler automatically optimizes our React app, it relies on several key assumptions about the code it processes. Understanding and adhering to these assumptions is essential to fully leverage the Compiler's capabilities and ensure optimal performance.

Valid, semantic JavaScript

The React Compiler assumes that the code it handles is valid and follows semantic JavaScript principles.

Safe Handling of Nullable Values

To ensure stability, the Compiler assumes that the code it processes includes safety checks for nullable and optional values before they are accessed.

For example, safely checking for a nullable/optional value could be achieved through conditional checks:

Conditionally checking the presence of a property

```
/*
  If nullableProperty is not null or undefined,
  access the 'foo' property
*/
if (object.nullableProperty) {
  return object.nullableProperty.foo
}
```

Or with optional chaining:

Accessing a property safely with optional chaining

```
/*
  Access the 'foo' property safely using
  optional chaining
*/
return object.nullableProperty?.foo
```

In a React/TypeScript setting, ensuring the safety of handling nullable and optional values can be bolstered by enabling the strictNullChecks compiler option.

Follows the Rules of React

The React Compiler not only requires that the code it processes is valid and semantically correct JavaScript, but it also assumes that this code

adheres to the "Rules of React." These rules are critical for ensuring that components and hooks behave predictably and maintainably, which is essential for the compiler to perform its optimizations effectively.

When the Compiler encounters code that violates these rules, it doesn't attempt to force the code to compile. Instead, it safely skips over the offending components or Hooks and continues compiling the rest of the code.

These rules include ensuring components and Hook are used in a manner that aligns with their intended design patterns within React:

- Components and Hooks must be pure
- Components must be idempotent
- Props and state should be treated as immutable
- Hooks should only be called at the top level and from React functions
- Etc.

For more details on how to implement these principles and the complete list of rules, be sure to check the official documentation on the "Rules of React".

Getting started

While the React Compiler is currently in use in production at Meta, it's important to emphasize that it is still in the experimental stage and not yet deemed stable for broad adoption.

For those keen to explore the React Compiler before its official release, the React team has prepared comprehensive guides to assist with the initial setup:

- React.dev — React Compiler Getting Started
- React Compiler Working Group — Successfully rolling out the compiler to your codebase

Teams interested in integrating the React Compiler into their production React applications can directly engage with the React team by applying to join the React Compiler Working Group. Finally, for more information

on using the Compiler, check out these talks given by the core React team members:

- Forget About Memo by Lauren Tan
- React Compiler Deep Dive by Sathya Gunasekaran and Mofei Zhang

React Server Components

React Server Components represent one of the most significant advancements in the React ecosystem, poised to reshape how some of us build React web applications. Unlike traditional React components, which run solely in the browser, React Server Components allow for a seamless integration of server-side rendering *into* the React architecture.

Before we delve into the specifics of React Server Components, it's helpful to establish a foundational understanding of one of the traditional paradigms of web applications: server-side rendering.

Server-side rendering

Traditionally, React applications have predominantly utilized client-side rendering, where the bulk of rendering logic and component handling occurs in the user's browser. This approach offers robust interactivity and user experience but often at the cost of initial load times and performance.

On the other hand, server-side rendering processes the application's components on the server, sending fully formed HTML to the browser, thus improving initial load times and SEO visibility but sometimes at the expense of interactivity and responsiveness.

To best illustrate how server-side rendering works, let's walk through a simple example of building a blog application that is a server-rendered React app. In particular, we'll focus our attention on a specific /blog/:id page—the page responsible for surfacing the details and full-text content of a single blog post.

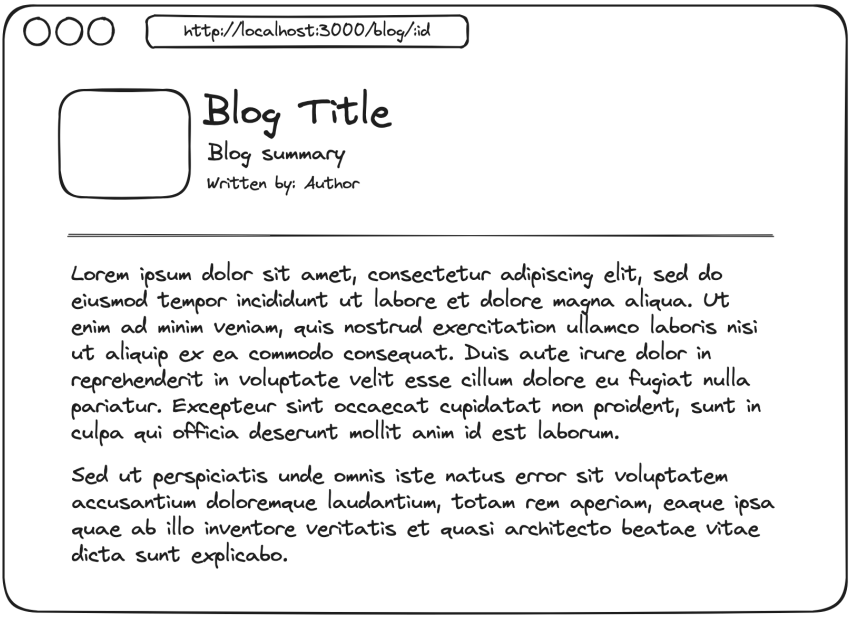


Figure 18-2. An example blog page

When a user visits the `/blog/:id` page of this hypothetical blog, requests to the URL follow a pattern:

1. **The client (i.e., browser) makes a request to the server for the particular page.** This directs the server on which initial content to retrieve.

2. **The server processes the initial request, retrieves the structural layout of the page, and sends the structural HTML to the client.** It retrieves the structural layout for the page, such as the HTML framework that will host the content, without yet fetching the detailed blog post data.

3. **The client receives the HTML, hydrates the page, and then makes an API request to gather data.** After processing the initial HTML, React hydrates the components on the client side. Hydration is the process where the browser takes the server-rendered HTML and attaches event handlers and other browser-only interactions. The client then makes an API request to the

server to fetch the actual data for the blog post using the ID from the URL. The server sends the requested data back to the client in the form of a JSON payload, which includes all the necessary information to populate the page fully.

4. **The client renders the content dynamically.** Using the data received from the API request, the client dynamically populates the content areas of the page with the blog post details. This step involves client-side scripting and rendering handled by React.

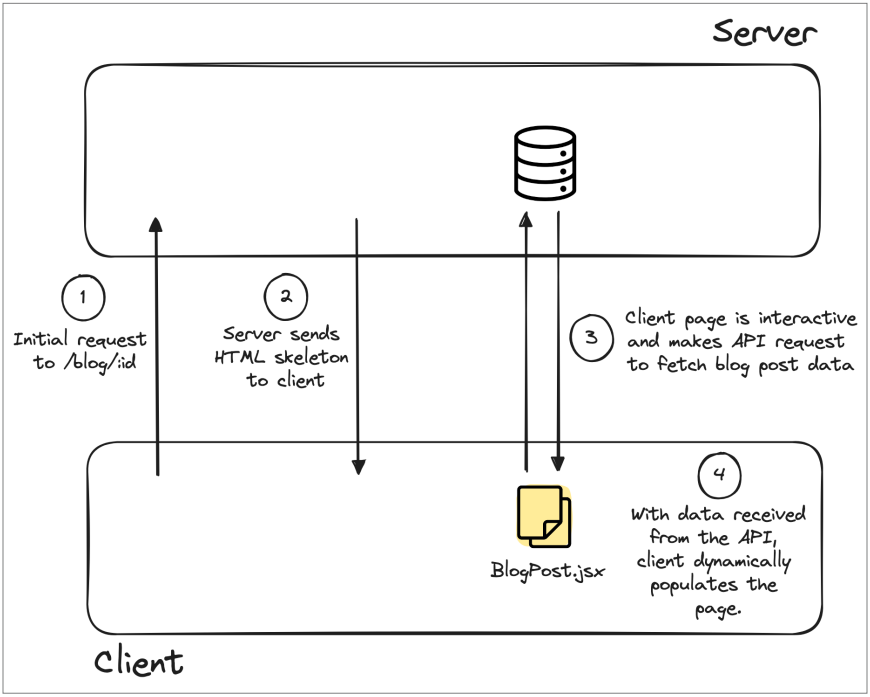


Figure 18-3. Diagram of the server-side rendering flow

There are some details we may have overlooked, but the above summary of the request flow showcases some of the main steps between the interactions of the client and the server in a server-rendered React application.

Notice an interesting behaviour about how the server returns HTML but *then* the client needs to make a subsequent API request after *back* to the server to fetch data? It would be more efficient if this data could be

retrieved in the *initial server response*, streamlining the process and reducing the need for multiple requests.

Next.js traditionally avoided this redundant round-trip by having a function that we can use called getServerSideProps. `getServerSideProps()` fetches the necessary data on the server side before the page is rendered, ensuring that all the required information is included in the initial server response, which helps streamline the rendering process and improve the overall performance of the application..

Outside of these outsized capabilities provided to us from third party frameworks like Next.js, the React client would have to handle data fetching and rendering separately. Wouldn't it be cool if there was a standard React way of integrating server-side data fetching directly into component rendering? This is exactly what **React Server Components** aims to solve.

React Server Components

React Server Components are a new capability in React that allows us to create stateless React components that *run on the server*. These components bring the power of server-side processing to the React architecture, enabling us to offload certain computations and data-fetching tasks from the client to the server. This shift can reduce the amount of code shipped to the client, decrease loading times, and even improve overall application performance.

In the rendering cycle we saw earlier, the server would not only send back the structural HTML but also process and render the actual content of the blog post server-side. As a result, things would look a little different particularly around how the data fetching and component rendering are managed.

If we had a server-rendered React application that utilizes React Server Components, requests to the `/blog/:id` URL would follow a pattern like the following:

1. The client (i.e., browser) makes a request to the server for the particular page. This directs the server on which initial content to retrieve. *[Same as before]*

2. **Server prepares and serves React Server Components.** Instead of sending only the structural HTML, the server uses React Server Components to render more complex elements, including data-fetching components directly. This allows the server to execute component-level logic, fetch necessary data, and integrate it directly into the rendered output.

3. **The client receives the HTML and relevant data from the server**. The server sends a response that includes both the HTML structure and the fully populated content of the blog post. This approach eliminates the need for the client to make a separate API call to fetch the blog content after loading the structural HTML. Upon receiving the server's response, the client hydrates the React components.

4. **The client renders the final content**. The client-side React application now handles user interactions and any dynamic changes that occur after the initial rendering. For instance, if a user posts a comment or interacts with the page, these actions can trigger client-side updates. *[Same as before]*

For the hypothetical blog application we're working with, the code for a React Server Component titled `BlogPost.js` could look something like the following:

`BlogPost` Server Component

```
import db from "./database";

// React Server Component
async function BlogPost({ postId }) {
  // Load blog post data from database
  const post = await db.posts.get(postId);

  return (
    <div>
      <h2>{post.title}</h2>
      <p>{post.summary}</p>
      <p>Written by {post.author}</p>
      <p>{post.content}</p>
    </div>
```

```
  );
}
```

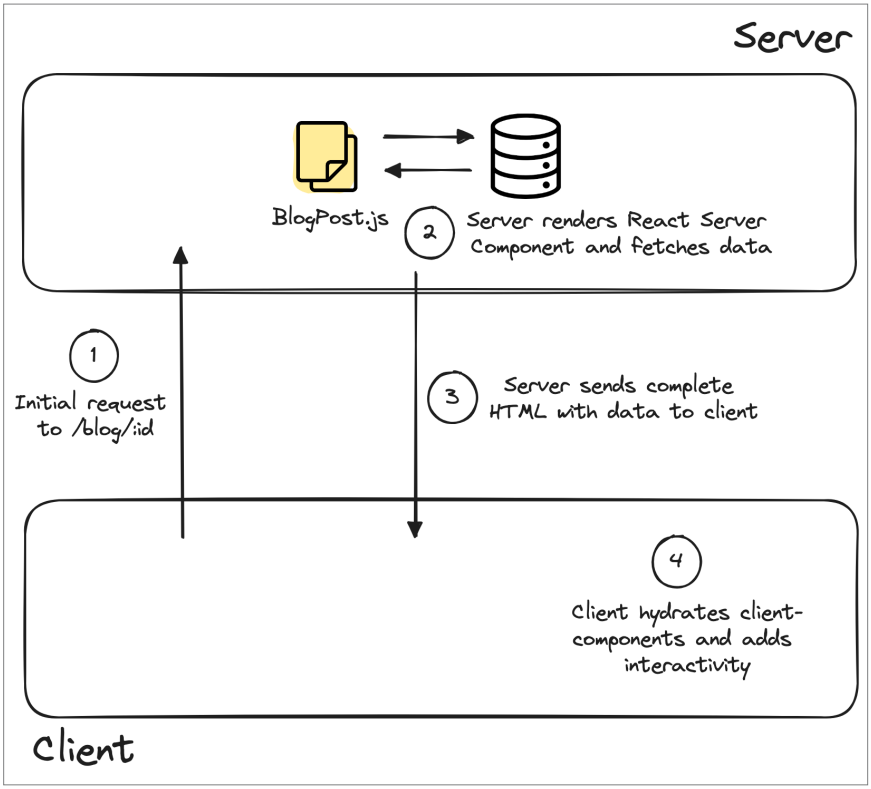


Figure 18-4. Diagram of the server-side rendering flow with RSCs

In the `BlogPost.js` code snippet, the `BlogPost` function is defined as a React Server Component that asynchronously fetches blog post data from a database using a provided `postId`. The data for the post, including its title, summary, author, and full content, is then rendered within a div element.

How cool is that! With this, we don't have to expose an API endpoint or use additional client-side fetching logic to load data directly into our components. All the data handling is done on the server.

Since React Server Components run exclusively on the server, there are some differences as to how they behave when compared with traditional Client Components.

No access to client-side React APIs

Since Server Components are run on the server and not the browser, they're unable to use traditional React component APIs like `useState`. To introduce interactivity to a React Server Component setting, we'll need to leverage Client Components that complement the Server Components for handling interactivity. To continue the above blog post example, this can look something like having a Client `Comment` component being rendered that includes some state and interactivity.

`BlogPost` rendering a list of Client `Comment` components

```
import db from "./database";

// React Server Component
async function BlogPost({ postId }) {
  const post = await db.posts.get(postId);

  // Load comments for the post from database
  const comments =
    await db.comments.getByPostId(postId);

  return (
    <div>
      <h2>{post.title}</h2>
      <p>{post.summary}</p>
      <p>Written by {post.author}</p>
      <p>{post.content}</p>

      // Render list of Comment Client components
      <ul>
        {comments.map((comment) => (
          <li key={comment.id}>
            <Comment {...comment} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Our `BlogPost` Server Component has now been updated to include commenting functionality, enabling users to see interactions related to

each post. The new `Comment` Client Component being rendered can include some client-side state and interactivity:

`Comment` Client Component

```
"use client";
import { useState } from 'react'

// React Client Component
export function Comment({ id, text }) {
  const [likes, setLikes] = useState(0);

  function handleLike() {
    setLikes(likes + 1);
  }

  return (
    <div>
      <p>{text}</p>
      <button onClick={handleLike}>
        Like ({likes})
      </button>
    </div>
  );
}
```

The above starts by rendering `BlogPost` as a Server Component, followed by configuring the bundler to assemble a bundle for the `Comment` Client Component. Within the browser, the `Comment` components receive the output from the `BlogPost` Server Component as props.

Notice the declaration of "use client" at the top of the `Comment` component file. When working with React Server Components, "use client" denotes that the component is a Client Component, which means it can manage state, handle user interactions, and use browser-specific APIs. This directive explicitly tells the React framework and bundler to treat this component differently from Server Components, which are stateless and run on the server.

On the flip-side, Server Components are the default so we don't state "use server" at the top of Server Component files. Instead, "use server" should only be used to mark server-side functions that can be called from

Client Components. These are called Server Actions (more on that shortly).

Async Server Components

Server Components have the capability to be *asynchronous*, enabling them to perform data fetching, computations, and other tasks directly within their execution on the server. This approach allows for the handling of complex operations before the content is ever sent to the client, optimizing the rendering process and enhancing the overall performance of web applications.

Async Server Components operate by defining asynchronous functions that handle data fetching, processing, or any other async tasks. Here's an example that builds a bit upon the previous discussion:

Awaiting for `post` content but not `comments`

```
import { Suspense } from "react";
import { Comments } from "./comment";
import db from "./database";

// async component
async function BlogPost({ postId }) {
  // await here
  const post = await db.posts.get(postId);

  // No await here
  const comments =
    db.comments.getByPostId(postId);

  return (
    <div>
      <h2>{post.title}</h2>
      <p>{post.summary}</p>
      <p>Written by {post.author}</p>
      <p>{post.content}</p>
      <Suspense
        fallback={<p>Loading comments...</p>}
      >
        <Comments commentsPromise={comments} />
      </Suspense>
    </div>
```

```
  );
}
```

In the above example, the `await` keyword is used to fetch the main `post` content. This will suspend the Server Component until the data from the database is retrieved, ensuring that the essential elements of the blog post, such as the title, summary, author information, and content, are fully loaded before the component is rendered. This approach is crucial for critical content that needs to be immediately available upon the initial render, enhancing the user's experience by displaying a complete and informative post without delay.

The comments section, on the other hand, is handled differently. By initiating the comments fetch without using `await`, the request becomes non-blocking. This means the Server Component does not wait for the comments to be loaded before continuing the rendering process. Instead, these comments are fetched asynchronously and managed by React's `Suspense` component, which shows a fallback loading message until the comments are ready to be displayed.

Since the `comments` content is initiated on the server but not awaited there, it can be passed to the client where it will be handled asynchronously. This is where React's new `use()` function comes in.

In the `Comment` Client Component, the `use()` function will play the important role in managing the asynchronous loading of comments. By using `use(comments)`, the component subscribes to the promise passed down from the Server Component. This allows the Client Component to render the comments as soon as they are available without blocking the initial rendering of the rest of the page.

Subscribing to promise with `use()`

```
"use client";
import { use } from "react";
import { Comment } from "./Comment";

// React Client Component
export function Comments({ commentsPromise }) {
  const comments = use(commentsPromise);

  return (
```

```
    <ul>
      {comments.map((comment) => (
        <li key={comment.id}>
          <Comment {...comment} />
        </li>
      ))}
    </ul>
  );
}
```

With this setup, the post content is loaded and shown immediately because it is awaited on the server, ensuring that critical information is available upon the initial render, while comments are loaded asynchronously and displayed as they become available, without delaying the rendering of the post content. This pattern embodies the principles of progressive enhancement, where basic content functionality is provided initially, and additional features are layered incrementally.

Server Actions

Server Actions allow Client Components to invoke server-side functions directly, which can sometimes help combine the benefits of server-side processing with the responsiveness of client-side dynamics.

Server Actions can be defined within Server Components using the `use server` directive. Building on our blog post example, let's add a Server Action to handle an upvote capability:

Server Action created in `BlogPost` Server Component

```
import { Comments } from "./Comments";
import db from "./database";

async function BlogPost({ postId }) {
  // ...

  // Server Action
  async function upvoteAction(commentId) {
    "use server";
    await db.comments.incrementVotes(
      commentId,
      1,
    );
```

```
  }

  return (
    <div>
      {/* ... */}

      {/* Server Action passed down as props */}
      <Comments
        commentsPromise={comments}
        upvoteAction={upvoteAction}
      />
    </div>
  );
}
```

The Server Action we defined is passed down as props and can now be used in our Client Component. We'll update our `Comments` component to utilize this action:

Server Action used in `Comments` Client Component

```
"use client";
import { use } from "react";

export function Comments({
  commentsPromise,
  upvoteAction,
}) {
  const comments = use(commentsPromise);

  return (
    <ul>
      {comments.map((comment) => (
        <li key={comment.id}>
          {comment.text} (Votes: {comment.votes}
          )
          <button onClick={upvoteAction}>
            Upvote
          </button>
        </li>
      ))}
    </ul>
  );
}
```

In the `Comments` Client Component, we directly use the `upvoteAction()` as an `onClick` handler for each comment's upvote button. The `commentId` is passed to the action when the button is clicked.

When the "Upvote" button is clicked, the `upvoteAction()` Server Action is invoked, triggering a server-side operation to increment the vote count for the specific comment. This process occurs without requiring a full page reload, providing a seamless user experience that combines the real-time responsiveness of client-side interactions with the data integrity and processing power of server-side operations.

In addition to having Server Actions being passed down from Server Components to Client Components, Client Components can also import Server Actions directly from files that declare the "use server" directive.

Server Actions defined in a separate file

```
"use server";

import db from "./database";

export async function upvoteAction(commentId) {
  await db.comments.incrementVotes(
    commentId,
    1,
  );
}

export async function downvoteAction(
  commentId,
) {
  await db.comments.incrementVotes(
    commentId,
    -1,
  );
}
```

Are RSCs the Future of Building with React?

As we've come to see, React Server Components (RSCs) are poised to reshape the landscape of React development by offering a blend of

server-side and client-side rendering that optimizes performance and user experience. React Server Components bring several benefits to the world of React development, which include:

1. **Improved Performance**: By offloading complex computations and data fetching to the server, RSCs reduce the amount of JavaScript that needs to be sent to the client. This leads to faster initial load times and improved performance, particularly for users on slower connections or devices.

2. **Enhanced SEO**: Server-side rendering ensures that content is readily available to search engines, enhancing SEO and making applications more discoverable.

3. **Simplified Data Fetching**: RSCs integrate data fetching directly into the component rendering process, eliminating the need for separate API calls from the client. This streamlined approach reduces the complexity of managing client-side state and data synchronization.

It's important to note that React Server Components represent a relatively new approach to running server code within components. This paradigm shift brings both opportunities and challenges. Developers accustomed to client-side React development may need to adjust their mental models and practices to fully leverage RSCs. The React ecosystem, including libraries and tools, will need to evolve to support and optimize for RSC-based development. As RSCs become more widely adopted, new best practices and patterns will emerge, shaping how we structure and build React applications.

Furthermore, using RSCs currently requires a compatible server and client environment, which often means relying on a framework. For a significant period of time, Next.js was the only framework supporting RSCs, paving the way for their adoption. However, as React 19 enters stability, other frameworks are beginning to incorporate RSC support which include RedwoodJS, Waku, and Gatsby. We can expect to see more frameworks adopt RSC support in the coming months and years, broadening the options for developers.

With all that we've covered, this raises the important question—**are RSCs the future of building with React**? As always, it depends.

React Server Components represent an exciting evolution in React development, but they are not necessarily the only way forward. React still remains a powerful utility for client applications and can be used as a standalone client library. Developers have the flexibility to continue building Single Page Applications (SPAs) with React or Server-Side Rendered (SSR) apps without having to adopt RSCs.

React's strength lies in its versatility and ability to accommodate different development approaches. As a library, React does a fantastic job of facilitating various ways of building applications. A prime example of this flexibility is the continued support for class-based components. Even though they were largely deprecated in favor of functional components and Hooks in 2018, class components are still supported in React!

In conclusion, while React Server Components offer compelling benefits and may become increasingly prevalent, they represent an additional tool in the React ecosystem rather than a mandatory replacement for existing patterns. The future of building with React is likely to be diverse, with developers choosing the approach that best suits their project requirements, whether that involves RSCs, traditional client-side rendering, or a hybrid approach.

Additional reading

- React Server Components — React.dev
- Server Actions — React.dev
- Understanding React Server Components — Vercel
- When to use Server and Client Components? — Vercel
- Making Sense of React Server Components — Josh W. Comeau
- React Server Components: A comprehensive guide — Chinwike Maduabuchi

About the Authors

Addy Osmani is an engineering leader working on Google Chrome. He leads up Chrome's Developer Experience organization, helping reduce the friction for developers to build great user experiences.

Hassan Djirdeh is a software engineer and has helped build large production applications at scale at organizations like Doordash, Instacart, and Shopify.