BUILDING LARGE SCALE WEB APPS

A REACT FIELD GUIDE

ADDY OSMANI HASSAN DJIRDEH

Learn tools and techniques to build and maintain large-scale React web applications.

www.largeapps.dev

Modularity

"The secret to building large apps is never to build large apps. Break your applications into small pieces. Then, assemble those testable, bite-sized pieces into your big application." *[Justin Meyer]*

One of the key principles of building large JavaScript applications is to modularize and componentize code. This can be described as **dividing an application into small, independent modules or components that can be developed and tested independently.**

Modularity (and componentization) makes our code more reusable by allowing us to share modules and components between different projects and teams. This can save time and reduce costs by avoiding the need to reinvent the wheel for every new project. In addition, by making it easier to manage and maintain our application, we also reduce the likelihood of bugs and make it easier to add new features in the future.

The best way to build a modular application is to start with small, self-contained pieces of functionality that we can test and debug independently. We can write that code as a module and make sure it works as expected before adding more modules or components.

Modules in JavaScript

JavaScript modules allow us to break up code into separate files that can export and import functionality. The native modules syntax provides:

- The export declaration: for exporting anything - functions, objects, primitives, etc.
- The import declaration: for importing from other modules.

Modules encourage reusability and maintainability. We can write a module once and reuse it across different parts of our application and in different projects. As a result, updates to a module won't require changes across the entire codebase. Modules also enable encapsulation. By only exporting the functionality we need publicly, we can hide internal implementation details in private module scopes.

For example, we can have a UI module that exports reusable UI components:

ui.js

```
export function Button({text}) {
  // button component
}

export function Header({title}) {
  // header component
}
```

And import just the parts we need in other files:

page.js

```
import { Header } from './ui.js';

function Page() {
  return <Header title="My Page"/>;
}
```

As our JavaScript applications grow in size and complexity, the importance of modularity becomes even more pronounced. Modules allow for organized, reusable, and maintainable codebases.

With this brief understanding of JavaScript modules under our belt, let's now see how the concept of modules translates when working within a React application.

Componentization in React

In the context of React, modules are often applied in a pattern called "componentization." React, by its nature, encourages developers to think in terms of **components**. Each component represents a distinct piece of the UI and, when built correctly, can be reused across different parts of an application, much like how modules allow us to reuse code.

The beauty of componentization is that it aligns with the modular approach we discussed. For instance, we might have a `Button` component in one file and a `Header` component in another. These components, much like JavaScript modules, can be imported and used wherever needed in our React application. This ensures a consistent look and behavior throughout, while also centralizing the logic and state management for each component.

With a growing React application, the need arises to organize components in a scalable and maintainable manner. This is where the concepts of component libraries, atomic design, or even domain-driven design might come into play. We discuss some of these concepts throughout the book, but for now, we'll delve deeper into some common and important strategies for approaching componentization in a large-scale React application:

Identify reusable components

The first step in componentizing our React application is to identify reusable components.

Good opportunities for componentization include:

- Repeated elements like buttons, menus, and cards.
- Sections of a page like headers, content areas, and footers.
- Logical chunks of functionality.

By identifying reusable components, we can create a library of components that can be reused throughout our application and even in other projects. This can save us a lot of time and effort in development and testing.

Here's an example of a `Post` component that contains functionality for a post made by a certain author on a social network site. This includes information like the author's name and the post's title, text, date, and other post details.

A post element that contains all the functionality in one component

```
function Post({ post }) {
  return (
    <div>
      <img
        src={post.profileUrl}
        alt={`${post.author}'s profile`}
      />

      <h1>{post.title}</h1>

      <p>{post.text}</p>

      <div>Author: {post.author}</div>

      <div>Date: {post.date}</div>
      <p>{`${post.numLikes} likes`}</p>

      <p>{`${post.numComments} comments`}</p>

      <p>{`${post.numShares} shares`}</p>

      <button>Like</button>
      <button>Share</button>
      <button>Comment</button>
    </div>
  )
}

export default Post;
```

In this example, we're rendering all the content of the post element in a single component. The `post` prop is passed to the component as a parameter, and we're using it to display the title, text, author, date, and other information of the post. We're also including buttons to like, share, and comment on the post.

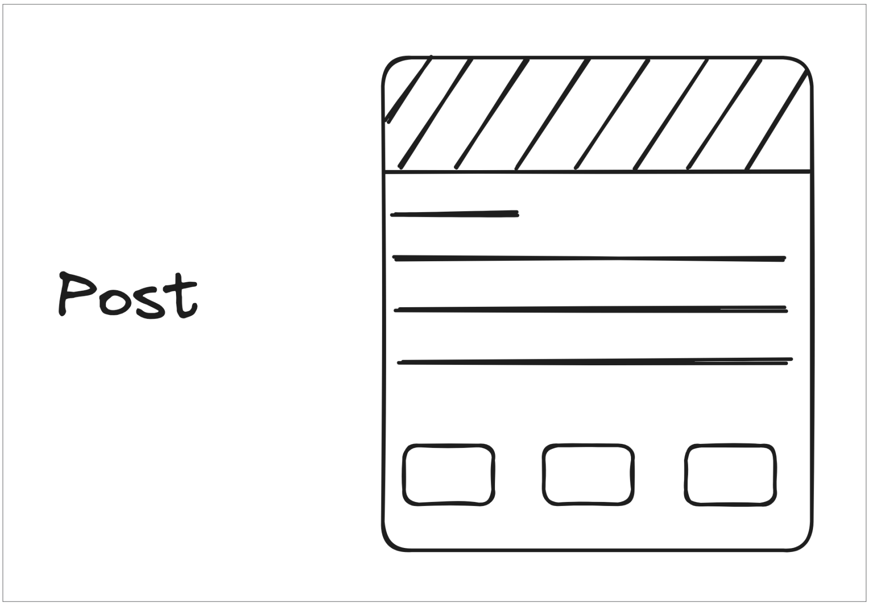

Figure 3-1. Post component

The approach of having all the UI of a certain element be kept within one component can be simple and quick to implement, but it can also make the component more difficult to maintain and test as the codebase grows. Modularity and componentization can help with maintaining and testing, as well as making the component more reusable and flexible.

Divide your application into smaller components

Breaking our application down into smaller, more manageable components is a key aspect of componentization. Instead of building large, monolithic components, we can instead divide our application into smaller components that are easier to develop, test, and maintain.

Smaller components are also more flexible and can be reused in different contexts, making our application more adaptable and scalable.

For example, we can break the `Post` component into smaller components as follows:

- PostHeader
- PostContent
- PostFooter

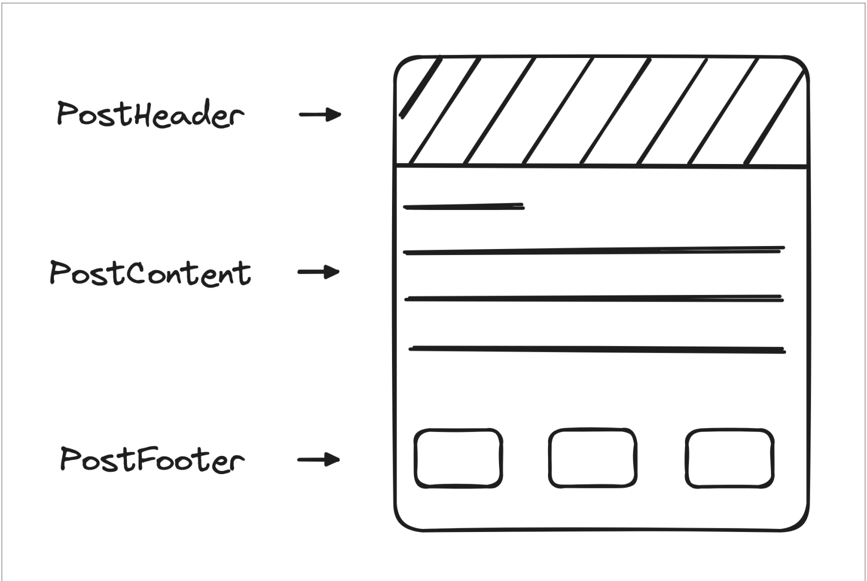

Figure 3-2. PostHeader, PostContent, and PostFooter components

PostHeader

The `PostHeader` component can hold the responsibility for displaying the header of the post. It can take props for the author's name, profile picture, and timestamp.

The `PostHeader` component

```
function PostHeader({
  authorName,
  profileUrl,
  timestamp
}) {
  return (
    <div>
      <img
        src={profileUrl}
        alt={`${authorName}'s profile`}
      />
      <p>{authorName}</p>
      <p>{timestamp}</p>
    </div>
  );
}

export default PostHeader;
```

PostContent

The `PostContent` component will be responsible for displaying the main content of the post. It can take a prop for the post's text and perhaps also an array of media elements (e.g., images, videos).

The `PostContent` component

```
function PostContent({ text, media }) {
  return (
    <div>
      <p>{text}</p>
      {media.map((element, index) => (
        <img
          key={index}
          src={element.url}
          alt={element.alt}
        />
      ))}
    </div>
  );
}

export default PostContent
```

PostFooter

The `PostFooter` component would be responsible for displaying the footer of the post and can take props for the number of likes, comments, and shares.

The `PostFooter` component

```
function PostFooter({
  numLikes,
  numComments,
  numShares,
}) {
  return (
    <div>
      <p>{`${numLikes} likes`}</p>
      <p>{`${numComments} comments`}</p>
      <p>{`${numShares} shares`}</p>

      <button>Like</button>
      <button>Share</button>
      <button>Comment</button>
    </div>

  );
}

export default PostFooter
```

Post

The `Post` component will finally be responsible for combining all of these smaller components into a cohesive post element. It can take props for the post's author, content, and footer and pass these props down to each of the child components that require them.

The parent `Post` component

```
function Post({
  author,
  content,
  footer,
}) {
  return (
    <div>
      <PostHeader
        authorName={author.name}
        profileUrl={author.profileUrl}
        timestamp={author.timestamp}
      />

      <PostContent
        text={content.text}
        media={content.media}
      />

      <PostFooter
        numLikes={footer.numLikes}
        numComments={footer.numComments}
        numShares={footer.numShares}
      />
    </div>
  );
}

export default Post
```

By dividing the post into smaller components like this, we make the code more modular and easier to maintain. We can also reuse these smaller components in other parts of the application, making it more scalable and adaptable.

While it's tempting to make everything a component, it's essential to strike a balance. Too granular, and you might end up with a codebase that's hard to navigate. Too broad, and you miss out on the benefits listed above. How we decide to break down components can be based on multiple factors:

- **Reusability**: Do we want a specific UI element or functionality to be repeated in various parts of the application? If so, it might be a good candidate to be turned into its own component.

- **Simplicity and readability**: How readable is the component's code? Would it be easier to read and understand the code if the component was broken down into smaller sub-components with their own focused responsibilities?

- **Improved testability**: Can the component be tested more effectively when it's smaller and has a focused responsibility? Smaller components often have less internal state and fewer side effects, making them easier to isolate in tests and ensuring that each piece of functionality works as intended.

- **Performance considerations**: Would breaking down the component optimize rendering or reduce unnecessary operations? In frameworks like React, smaller components can sometimes lead to fewer re-renders with the capability to memoize components and computations. This could sometimes improve the app's overall performance.

Implement a Design System

A design system is a collection of reusable components, guidelines, and assets that help teams build cohesive products. Many popular design systems exist today, such as Material by Google, Polaris by Shopify, Human Interface Guidelines by Apple, Fluent Design System by Microsoft, and more.

Building a design system can help us standardize the design and development of components in our React application. Furthermore, using an existing open-source design system, like Material by Google, can expedite the development process by providing a well-documented set of components and patterns. This allows us to focus on application-specific logic and functionality.

We go into more detail discussing how reusable components play a role in building and maintaining design systems in the upcoming chapter—**Design Systems**.

Lazy-loading

Lazy-loading is a technique for loading resources only when they are needed. This can be useful for improving the performance of an application by reducing the amount of resources that need to be loaded initially.

In React, we can optimize the performance and responsiveness of applications by judiciously loading components only when they are required (i.e., lazily). We're able to achieve this with the help of React's lazy function and Suspense component.

- **lazy**: a function that allows us to load components on demand.
- **Suspense**: a component that can be used to display a fallback component while the lazy component is being loaded.

Let's go through an example of how we can use lazy-loading in a React application to load the `Post` component we created earlier. We'll first start with a basic example of importing the `Post` component statically, and then show how to use dynamic imports with `React.lazy()` to load the component dynamically.

Static import

With a standard static import, the `Post` component is imported at the top of the file with the import declaration:

Static import of the `Post` component

```
import React from 'react';

// importing the Post component
import Post from './components/Post';
```

```
function App() {
  return (
    <div>
      <Post />
    </div>
  );
}

export default App;
```

If the `Post` component is large and takes a long time to load, this can impact the initial loading time of its parent `App` component, especially if the `Post` component is not immediately needed on the initial render. This is because static imports will ensure that the entire `Post` component and its dependencies are fetched and executed before the `App` component is initialized.

Dynamic import

To avoid this issue, we can attempt to load the Post component lazily (i.e., dynamically) with React's `lazy()` function.

Dynamic import of the `Post` component

```
import React, { lazy, Suspense } from 'react';

// dynamically importing the Post component
const Post = lazy(
  () => import("./components/Post"),
);

function App() {
  return (
    <div>
      {/* using Suspense to render fallback while
          Post is dynamically loading */}
      <Suspense fallback={<div>Loading...</div>}>
        <Post />
      </Suspense>
    </div>
  );
}
```

```
export default App;
```

In the example above, we leverage the dynamic import() syntax to asynchronously load the `Post` component. The `import()` function returns a promise that resolves to the imported module, allowing us to utilize it in conjunction with React's `lazy()` function to create a lazily-loaded component.

With this approach, the `Post` component is only loaded when it's actually needed, reducing the initial bundle size and improving the load times of the `App` component.

The `Suspense` component is used to display a loading message or placeholder while the `Post` component is being loaded.

Lazy-load on interaction

We can also use this lazy-loading pattern to dynamically load components on interaction/click.

Here's an example of how we can import the `Post` component with a click instead of dynamically loading the component as the parent is being rendered.

Lazy-loading the `Post` component on button click

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

function App() {
  const [Post, setPost] = useState(null);

  const handleClick = () => {
    import("./components/Post").then((module) => {
      setPost(() => module.default);
    });
  };

  return (
    <div>
      {Post ? (
        <Post />
      ) : (
        <button onClick={handleClick}>
          Load Post
        </button>
      )}
    </div>
  );
}

export default App;
```

In this example, we use the `useState` Hook to initialize a `Post` state variable as `null`. When the button is clicked, we use the dynamic `import()` function to load the `Post` component and then set it to the value of the `Post` state. Once `Post` is set, it will be rendered to the screen.

Note that using React's `lazy()` function with dynamic imports only works with default exports, so if the `Post` component has named exports, we'll need to adjust the import statement accordingly. Also, keep in mind that dynamic imports with React's `lazy()` function should only be used for large components that are not needed immediately, as it adds some complexity to the code and can cause issues with server-side rendering.

Lazy-load with the Intersection Observer API

Intersection Observer is a JavaScript API that allows us to detect when an element is visible in the viewport. This can be useful for implementing on-demand code splitting, where code is loaded when the user scrolls to a specific section of the page.

To use the Intersection Observer API in your React application, we can create our own custom functionality or import this functionality from a third-party library like react-intersection-observer.

Here's a rough example of how we could use a custom `useIntersectionObserver()` Hook to lazily load the `Post` component when it enters the viewport:

Lazy-loading with Intersection Observer

```
import React, {
  useState,
  useRef,
  lazy,
  Suspense,
} from "react";
import useIntersectionObserver from "./hooks";

const Post = lazy(
  () => import("./components/Post"),
);

function App() {
  const [shouldRenderPost, setShouldRenderPost] =
    useState(false);
  const postRef = useRef(null);

  const handleIntersect = ([entry]) => {
    if (entry.isIntersecting) {
      setShouldRenderPost(true);
    }
  };
  useIntersectionObserver(
    postRef,
    handleIntersect,
    { threshold: 0 },
  );

  return (
    <div>
      <div style={{ height: "1000px" }}>
        Some content before the post
      </div>

      <div ref={postRef}>
        {shouldRenderPost ? (
          <Suspense
            fallback={<div>Loading...</div>}
          >
            <Post />
          </Suspense>
        ) : (
          <div>Loading...</div>
        )}
      </div>

      <div style={{ height: "1000px" }}>
        Some content after the post
      </div>
    </div>
  );
}

export default App;
```

In the above example, we're using a `useIntersectionObserver()` Hook to watch for changes in the visibility of the `postRef` element, and trigger the `handleIntersect()` callback when it enters the viewport. The `shouldRenderPost` state property is set to a value of `true` when the element is intersecting, which triggers the rendering of the `Post` component inside a `Suspense` component.

Note that this example assumes that the `useIntersectionObserver()` Hook is defined somewhere in our codebase.

By loading resources only when they are needed, lazy-loading components can significantly minimize the initial load of an application, thereby improving user experience and resource utilization. While lazy-loading focuses on loading components only when necessary, the concept of **code-splitting** takes this a step further by breaking down an entire application into smaller chunks that can be loaded independently.

Code-splitting

Code splitting is a technique for optimizing the performance of large applications by splitting the application's code into smaller, more manageable chunks.

By having a modular or componentized application structure, we naturally pave the way for more efficient code-splitting. When components are designed to be self-contained and independent, it becomes easier to separate them into distinct chunks that can be loaded on demand. This modular approach aligns perfectly with the core idea behind code-splitting, where the aim is to load only the necessary code for the user at any given moment rather than loading the entire application upfront.

In React applications, code-splitting patterns commonly include:

- **Splitting by route**: Load page modules as user navigates.
- **Splitting by component**: Lazy-load large components like graphs and tables.
- **On-demand loading**: Load code when a user clicks buttons, dropdowns, etc.

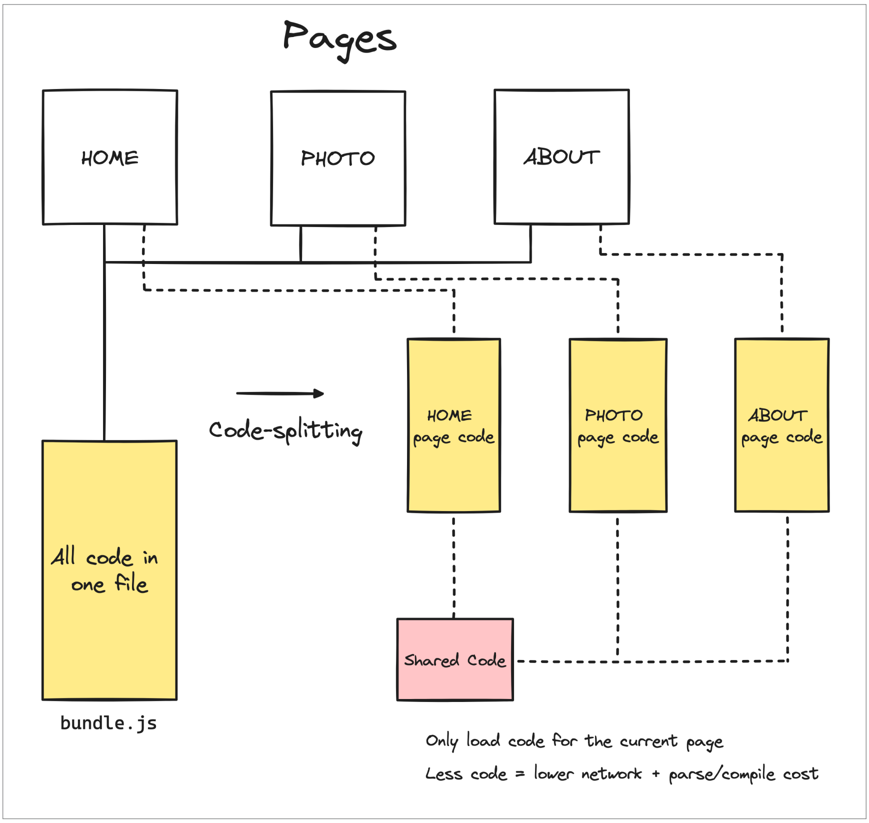

Figure 3-3. Code-splitting a monolithic bundle.js into granular chunks.

The first step in implementing code splitting is to identify the critical path of our application. The critical path is the sequence of resources that must be loaded before our application can be displayed to the user. By identifying this critical path, we can determine which resources should be loaded first and which resources can be loaded later using advanced code-splitting techniques.

In the next few sections, we'll discuss some strategies for approaching advanced code splitting in a large React application.

Entry point splitting

The entry point is the initial JavaScript file that is loaded when a user visits a website. **With entry point splitting, we break up the initial JavaScript file into smaller chunks that are only loaded when needed**, reducing the initial load time for the page.

For example, imagine we have a website with:

- A home page
- A product page
- And a contact page

Each of the pages has its own unique JavaScript code. If we load all of the code at once, the initial page load time could be slow. With entry point splitting, we can break up the code for each page into separate chunks. With this type of splitting, when a user visits:

- The home page: only the code for the home page is loaded.
- The product page: only the code for the product page is loaded.
- The contact page: only the code for the contact page is loaded.

This type of code-splitting results in faster load times and a better user experience since only the necessary code is loaded for each specific page, reducing the amount of redundant or unnecessary data being fetched.

Vendor splitting

Vendor splitting is a technique used to separate out third-party dependencies from your own code. When we use a third-party library or framework, the code for that library is included in our JavaScript bundle. This can make the bundle larger and slower to load and can also cause cache invalidation issues when the library is updated.

With vendor splitting, we can break out the code for these third-party dependencies into a separate chunk that can be cached independently. This means that when we update our own code, the end user won't need to re-download the entire library since it's already cached. This can result in faster load times and a better user experience by optimizing caching and reducing unnecessary data downloads.

Dynamic splitting

Dynamic splitting is a technique used to load JavaScript code on demand, as needed. This is useful for large-scale JavaScript applications where different parts of the code are only needed in certain situations. For example, if we have an application with a dashboard and a settings page, the code for the dashboard might not be needed when the user is on the settings page, and vice versa.

With dynamic splitting, we can load the code for each page or component only when it's needed. This can reduce the initial load time for the page and improve performance overall. It can also help to keep the size of the JavaScript bundle under control, which is important for large-scale applications.

Dynamic splitting differs from entry-point splitting in that it doesn't rely solely on predefined entry points. Instead, it leverages tools and patterns like React's `lazy` and `Suspense` or the dynamic `import()` function to split code at specific modules or components. This allows developers to granularly control when different parts of the codebase are loaded based on user interactions or other runtime conditions.

Component-level splitting

In component-level code-splitting, each component is lazy-loaded only when it's needed, which means that the application loads only the components that are required for the current page. This technique can lead to more efficient use of bandwidth, but it can sometimes increase latency due to the need to load components on demand.

Route-based splitting

In route-based code-splitting, the application is split into separate bundles based on routes. When a user navigates to a different route, the appropriate bundle is loaded on demand, reducing the amount of code that needs to be downloaded initially. This technique can help reduce the initial load time of an application, but it may not be as efficient as component-level code-splitting in terms of bandwidth usage.

Trade-offs with aggressive code-splitting

Aggressive code-splitting refers to the practice of extensively breaking down the application's JavaScript into numerous small chunks. While code-splitting has clear benefits in terms of loading only the necessary code for a given view or action, there are some difficulties associated with aggressive code-splitting.

1. **Granularity trade-off**: When we aggressively code-split, we end up with a large number of smaller chunks of code. This can be good for caching and de-duplication but bad for compression and browser performance. Smaller chunks compressed individually get lower compression rates, and loading performance can be impacted, even with as low as 25 chunks and very severely at 100+ chunks.

2. **Interoperability**: Different browsers, servers, and CDNs may implement code-splitting differently, which can sometimes lead to compatibility issues.

3. **Overhead**: While code-splitting can improve loading performance, it can also introduce additional overhead due to the need to process, fetch, and parse multiple files. This could sometimes slow down an application, especially on slower devices or networks.

4. **Debugging**: With a large number of smaller chunks, it can be difficult to debug the code and identify issues as the code is spread across multiple files.

5. **Build complexity**: Aggressive code-splitting can make the build process more complex and time-consuming, as the codebase is broken down into multiple smaller chunks that need to be managed and sometimes built separately.

Wrap up

Modularity, through componentization, not only makes our applications more maintainable and scalable but also enhances the developer experience by providing a clear structure and reusability of components.

As applications grow in complexity, there's an ever-increasing need to optimize for performance and user experience. Code-splitting allows us to break down applications into manageable chunks, ensuring users load only the necessary code at the right time. Having our application broken down into components makes implementing code-splitting efficient, as we can dynamically load individual components based on user interaction or the current view.

In the next chapter, we'll spend a bit more time discussing and sharing helpful resources on the topic of **Performance**.