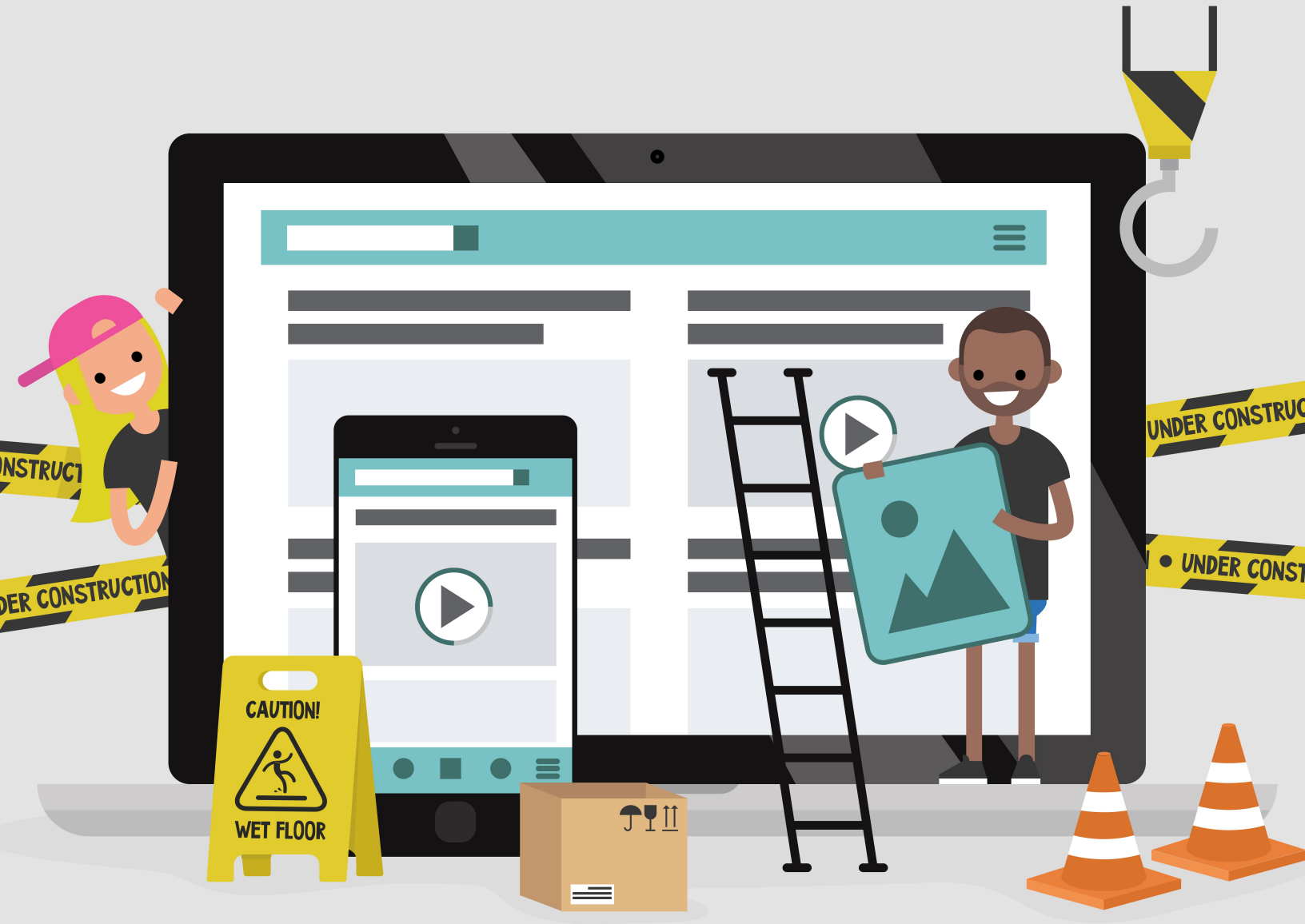


RENDERING

Rendering content on the web can be done in many ways today. The decision on how and where to fetch and render content is key to the performance of an application. The available frameworks and libraries can be used to implement different rendering patterns like Client-Side Rendering, Static Rendering, Hydration, Progressive Rendering and Server-Side Rendering. It is important to understand the implications of each of these patterns before we can decide which is best suited for our application.

The Chrome team has encouraged developers to consider static rendering or server-side rendering over a full rehydration approach. Over time, progressive loading and rendering techniques by default may help strike a good balance of performance and feature delivery when using a modern framework

The following sections will provide a guideline on measuring the performance requirements for an application with respect to web rendering and suggest patterns that best satisfy each of these requirements. Subsequently, we will explore each pattern in-depth and learn how it can be implemented. We will also talk a bit about Next.js which can be used to implement these patterns. However, before we go into the available patterns or Next.js, let's take a look at how we got here and what were the drivers that resulted in the creation of the React framework and Next.js.

A brief history of web rendering

Web technologies have been continuously evolving to support changing application requirements. The building blocks for all websites HTML, CSS and

JavaScript have also evolved over time to support changing requirements and utilize browser advancements.

In the early 2000's we had sites where HTML content was rendered completely by the server. Developers relied on server-side scripting languages like PHP and ASP to render HTML. Page reloads were required for all key navigations and JavaScript was used by clients minimally to hide/show or enable/disable HTML elements.

In 2006, Ajax introduced the possibility of Single-Page Applications (SPA), Gmail being the most popular example. Ajax allowed developers to make dynamic requests to the server without loading a new page. Thus, SPAs could be built to resemble desktop applications. Soon developers started using JavaScript to fetch and render data. JavaScript libraries and frameworks were created that could be used to build the view layer functionality in the MVC framework. Client-side frameworks like JQuery, Backbone.js and AngularJS made it easier for developers to build core features using JavaScript.

React was introduced in 2013 as a flexible framework for building user interfaces and UI components and provided a base for developing both single-page web and mobile applications. From 2015 to 2020 the React ecosystem has evolved to include supporting data-flow architecture libraries (Redux), CSS frameworks (React-Bootstrap), routing libraries and mobile application framework (React Native). However, there are some drawbacks of a pure Client-Side rendering framework. As a result, developers have started exploring new ways to get the best of both the Client-side and Server-side rendering worlds.

Rendering - Key Performance Indicators

Before we talk about drawbacks, let us understand how we could measure the performance of a rendering mechanism. A basic understanding of the following terms will help us to compare the different patterns discussed here.

Acronym	Description
TTFB	Time to First Byte - the time between clicking a link and the first bit of content coming in.
FP	First Paint - First time any content becomes visible to the user or the time when the first few pixels are painted on the screen
FCP	First Contentful Paint - Time when all the requested content becomes visible
LCP	Largest Contentful Paint - Time when the main page content becomes visible. This refers to the largest image or text block visible within the viewport.
TTI	Time to Interactive - Time when the page becomes interactive e.g., events are wired up, etc.
TBT	Total Blocking Time - The total amount of time between FCP and TTI.

Some important notes about these performance parameters are as follows.

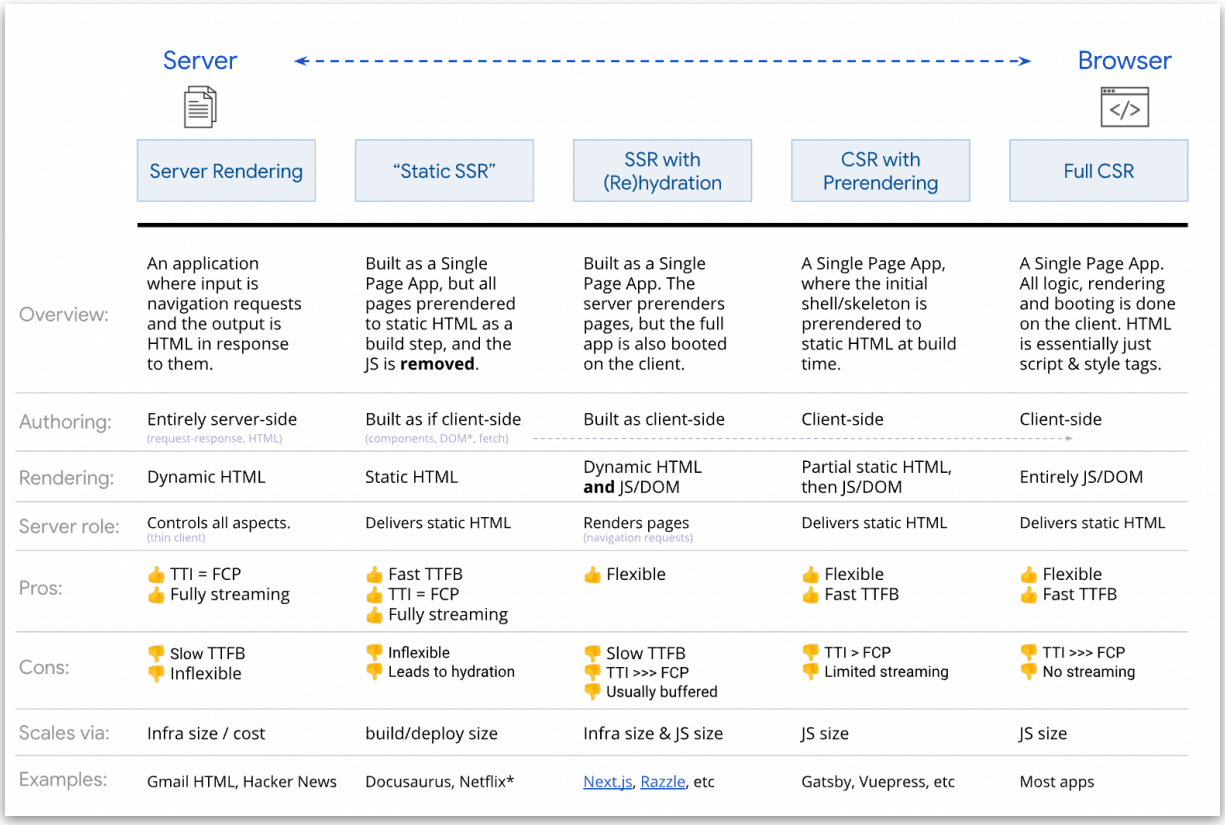
- A large JavaScript bundle could increase how long a page takes to reach FCP and LCP. The user will be required to wait for some time to go from a mostly blank page to a page with content loaded.
- Larger JavaScript bundles also affect TTI and TBT as the page can only become interactive once the minimal required JavaScript is loaded and events are wired.
- The time required for the first byte of content to reach the browser (TTFB) is dependent on the time taken by the server to process the request.

- Techniques such as preload, prefetch and script attributes can affect the above parameters as different browsers interpret them differently. It is helpful to understand the loading and execution priorities assigned by the browser for such attributes before using them.

We can now use these parameters to understand what exactly each pattern has to offer with respect to rendering requirements.

Patterns - A Quick Look

Client-Side Rendering (CSR) and Server-Side Rendering (SSR) form the two extremes of the spectrum of choices available for rendering. The other patterns listed in the following illustration use different approaches to provide some combination of features borrowed from both CSR and SSR.



We will explore each of these patterns in detail. Before that, however, let us talk about Next.js which is a React-based framework. Next.js is relevant to our discussion because it can be used to implement all of the following patterns.

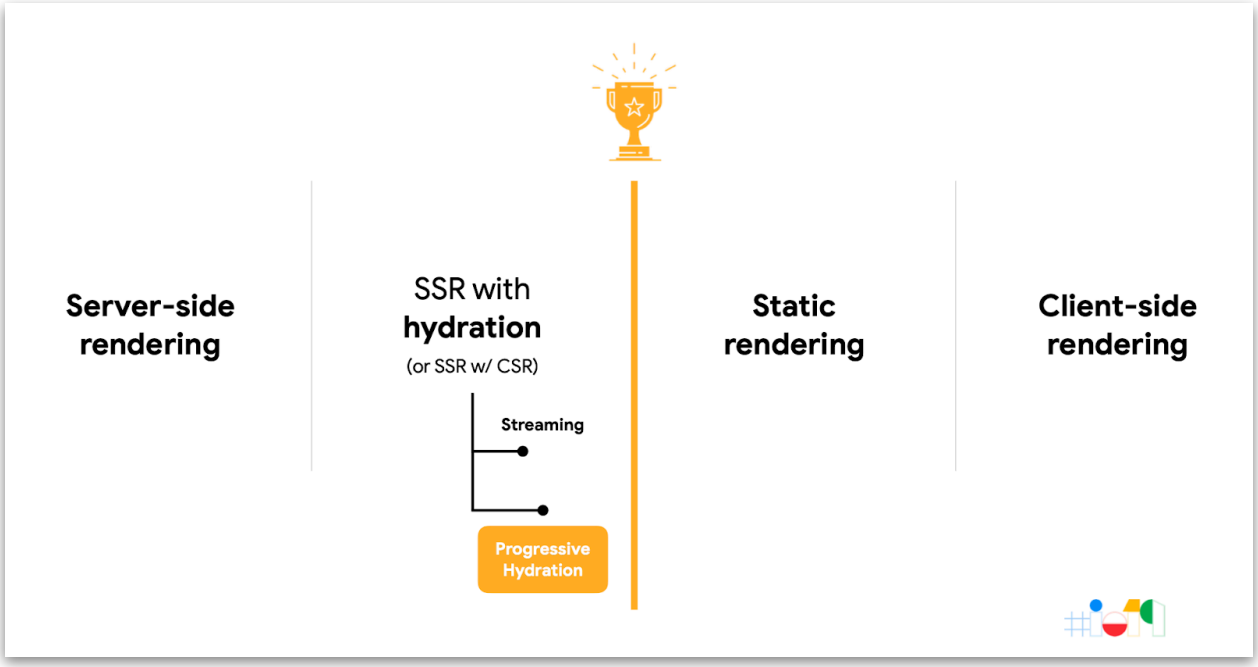
- SSR

- Static SSR (experimental flag)
- SSR with Rehydration
- CSR with Prerendering also known as Automatic Static Optimization
- Full CSR

based framework. Next.js is relevant to our discussion because it can be used to implement all of the following patterns.

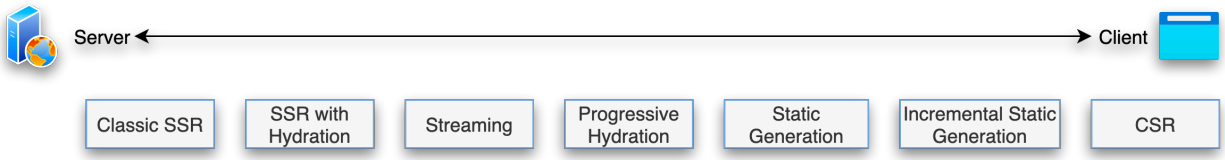
Conclusion

We have now covered four patterns that are essentially variations of SSR. These variations use a combination of techniques to lower one or more of the performance parameters like TTFB (Static and Incremental Static Generation), TTI (Progressive Hydration) and FCP/FP (Streaming). The patterns build upon existing client-side frameworks like React and allow for some sort of rehydration to achieve the interactivity levels of CSR. Thus, we have multiple options to achieve the ultimate goal of combining both SSR and CSR benefits.



Summary

Depending on the type of the application or the page type, some of the patterns may be more suitable than the others. The following chart revisits, summarizes and compares the highlights of each pattern that we discussed in the previous sections and provides use cases for each.



HTML generated on	Server	Server	Server	Server	Build Server	Build Server	Client
JavaScript for Hydration	No Hydration	JS for all components to be loaded for hydration	JS is streamed with HTML	JS is loaded progressively	Minimal JS	Minimal JS	No Hydration but JS for all components is required for rendering and interactivity
SPA Behaviour	Not Possible	Limited	Limited	Limited	Not Possible	Not Possible	Extensive
Crawler Readability	Full	Full	Full	Full	Full	Full	Limited
Caching	Minimum	Minimum	Minimum	Minimum	Extensive	Extensive	Minimum
TTFB	High	High	Low and consistent across page sizes	High	Low	Low	Low
TTI : FCP	TTI = FCP	TTI > FCP	TTI > FCP	TTI > FCP	TTI = FCP	TTI = FCP	TTI >> FCP
Implemented Using	Server side scripting languages like PHP	React for Server, Next.js	React for Server (React 16 onwards)	Full fledged React solution under development	Next.js	Next.js	CSR frameworks like React, Angular etc
Suitable For	Static content pages like news or encyclopedia pages	Mostly static pages with few interactive components. E.g., comments section of a blog	Mostly static pages that can be streamed in chunks. E.g., search results listing pages	Interactive pages where activation of some components may be delayed. E.g., Chatbot	Static content that does not change often. About Us or Contact us pages of websites	Huge quantities of static content that may change frequently. Blog listing or Product listing pages.	Highly Interactive apps where user experience is critical. E.g., Social media messaging and commenting

Overview of Next.js

Vercel's framework for hybrid React applications

Next.js, created by Vercel, is a framework for hybrid React applications. It is often difficult to understand all the different ways you might load content. Next.js abstracts this to make it as easy as possible. The framework allows you to build scalable, performant React code and comes with a zero-config approach. This allows developers to focus on building features.

Let us explore the Next.js features that are relevant to our discussion

Basic Features

Pre-rendering

By default, Next.js generates the HTML for each page in advance and not on the client-side. This process is called pre-rendering. Next.js ensures that JavaScript code required to make the page fully interactive gets associated with the generated HTML. This JavaScript code runs once the page loads. At this point, React JS works in a Shadow DOM to ensure that the rendered content matches with what the React application would render without actually manipulating it. This process is called hydration.

Each page is a React component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the pages directory. The route is determined based on the file name. E.g., `pages/about.js` corresponds to the route `/about`. Next.js supports pre-rendering through both Server-Side Rendering (SSR) and Static generation. You can also mix different rendering methods in the same app

where some pages are generated using SSR and others using Static generation. Client-side rendering may also be used to render certain sections of these pages.

Data Fetching

Next.js supports data fetching with both SSR and Static generation. Following functions in the Next.js framework make this possible.

- `getStaticProps`
  - Used with Static generation to render data
- `getStaticPaths`
  - Used with Static generation to render dynamic routes
- `getServerSideProps`
  - Applicable to SSR

Static File Serving

Static files like images can be served under a folder called `public` in the root directory. The same image may then be referenced in the `<img>` tag code on different pages using the root URL. E.g., `src=/logo.png`.

Automatic Image Optimization

Next.js implements Automatic Image Optimization which allows for resizing, optimizing, and serving images in modern formats when the browser supports it. Thus, large images are resized for smaller viewports when required. Image optimization is implemented by importing the Next.js Image component which

is an extension of the HTML `<img>` element. To use the Image component, it should be imported as follows.

```
import Image from 'next/image'
```

The image component can be served on the page using the following code.

```
<Image src="/logo.png" alt="Logo" width={500} height={500} />
```

Routing

Next.js supports routing through the `pages` directory. Every .js file in this directory or its nested subdirectories becomes a page with the corresponding route. Next.js also supports the creation of dynamic routes using named parameters where the actual document displayed is determined by the value of the parameter.

For example, a page `pages/products/[pid].js`, will get matched to routes like `/post/1` with `pid=1` as one of the query parameters. Linking to these dynamic routes on other pages is also supported in Next.js

Code Splitting

Code splitting ensures that only the required JavaScript is sent to the client which helps to improve performance. Next.js supports two types of code splitting.

- Route-based: This is implemented by default in Next.js. When a user visits a route, Next.js only sends the code needed for the initial route. The other chunks are downloaded as required when the user navigates around the application. This limits the amount of code that needs to be parsed and compiled at once thereby improving the page load times.

- Component-based: This type of code splitting allows splitting large components into separate chunks that can be lazy-loaded when required. Next.js supports component-based code splitting through dynamic import(). This allows you to import JavaScript modules (including React components) dynamically and load each import as a separate chunk.

Client-side Rendering

Render your application's UI on the client

In Client-Side Rendering (CSR) only the barebones HTML container for a page is rendered by the server. The logic, data fetching, templating and routing required to display content on the page is handled by JavaScript code that executes in the browser/client. CSR became popular as a method of building single-page applications. It helped to blur the difference between websites and installed applications.
To better appreciate the benefits provided by other patterns, let us first take a deeper look at Client-Side Rendering (CSR) and find out which are the situations where it works great and what are its drawbacks.

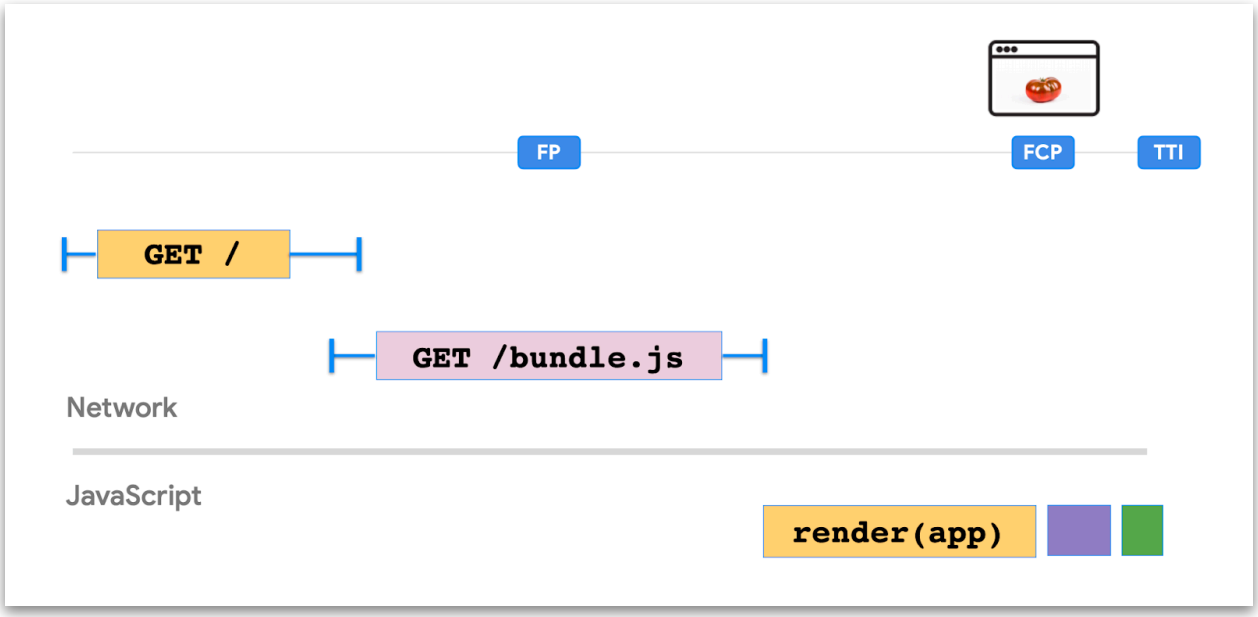
Consider this simple example for showing and updating the current time on a page using React.

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
 ReactDOM.render(element,document.getElementById('root'));
}
setInterval(tick, 1000);
```

The HTML consists of just a single root div tag. Content display and updates on the other hand are handled completely in JavaScript. There is no round trip to the server and rendered HTML is updated in-place. Here time could be replaced by any other real-time information like exchange rates or stock prices obtained from an API and displayed without refreshing the page or a round trip to the server.

JavaScript bundles and Performance

As the complexity of the page increases to show images, display data from a data store and include event handling, the complexity and size of the JavaScript code required to render the page will also increase. CSR resulted in large JavaScript bundles which increased the FCP and TTI of the page.



As shown in the above illustration, as the size of bundle.js increases, the FCP and TTI are pushed forward. This implies that the user will see a blank screen for the entire duration between FP and FCP.

Client-side React - Pros and Cons

With React most of the application logic is executed on the client and it interacts with the server through API calls to fetch or save data. Almost all of the UI is thus generated on the client. The entire web application is loaded on the first request. As the user navigates by clicking on links, no new request is generated to the server for rendering the pages. The code runs on the client to change the view/data.

CSR allows us to have a Single-Page Application that supports navigation without page refresh and provides a great user experience. As the data processed to change the view is limited, routing between pages is generally faster making the CSR application seem more responsive. CSR also allows developers to achieve a clear separation between client and server code. Despite the great interactive experience that it provides, there are a few pitfalls to this CSR.

1. **SEO considerations**: Most web crawlers can interpret server rendered websites in a straight-forward manner. Things get slightly complicated in the case of client-side rendering as large payloads and a waterfall of network requests (e.g for API responses) may result in meaningful content not being rendered fast enough for a crawler to index it. Crawlers may understand JavaScript but there are limitations. As such, some workarounds are required to make a client-rendered website SEO friendly.

2. **Performance**: With client-side rendering, the response time during interactions is greatly improved as there is no round trip to the server. However, for browsers to render content on client-side the first time, they have to wait for the JavaScript to load first and start processing. Thus users will experience some lag before the initial page loads. This may affect the user experience as the size of JS bundles get bigger and/or the client does not have sufficient processing power.

3. **Code Maintainability**: Some elements of code may get repeated across client and server (APIs) in different languages. In other cases, clean separation of business logic may not be possible. Examples of this could include validations and formatting logic for currency and date fields.

4. **Data Fetching**: With client-side rendering, data fetching is usually event-driven. The page could initially be loaded without any data. Data may be subsequently fetched on the occurrence of events like page-load or button-clicks using API calls. Depending on the size of data this could add to the load/interaction time of the application.

The importance of these considerations may be different across applications. Developers are often interested in finding SEO friendly solutions that can serve pages faster without compromising on the interaction time. Priorities assigned to the different performance criteria may be different based on application requirements. Sometimes it may be enough to use client- side rendering with some tweaks instead of going for a completely different pattern.

Improving CSR performance

Since performance for CSR is inversely proportional to the size of the JavaScript bundle, the best thing we can do is structure our JavaScript code for optimal performance. Following is a list of pointers that could help.

1. **Budgeting JavaScript**: Ensure that you have a reasonably tight JavaScript budget for your initial page loads. An initial bundle of < 100-170KB minified and gzipped is a good starting point. Code can then be loaded on-demand as features are needed

2. **Preloading**: This technique can be used to preload critical resources that would be required by the page, earlier in the page lifecycle. Critical resources may include JavaScript which can be preloaded by including the following directive in the <head> section of the HTML

```
<link rel="preload" as="script" href="critical.js">
```

This informs the browser to start loading the critical.js file before the page rendering mechanism starts. The script will thus be available earlier and will not block the page rendering mechanism thereby improving the performance.

1. **Lazy loading**: With lazy loading, you can identify resources that are non-critical and load these only when needed. Initial page load times can be improved using this approach as the size of resources loaded initially is

reduced. For example., a chat widget component would generally not be needed immediately on page load and can be lazy loaded.

2. **Code Splitting**: To avoid a large bundle of JavaScript code, you could start splitting your bundles. Code-Splitting is supported by bundlers like Webpack where it can be used to create multiple bundles that can be dynamically loaded at runtime. Code splitting also enables you to lazy load JavaScript resources.

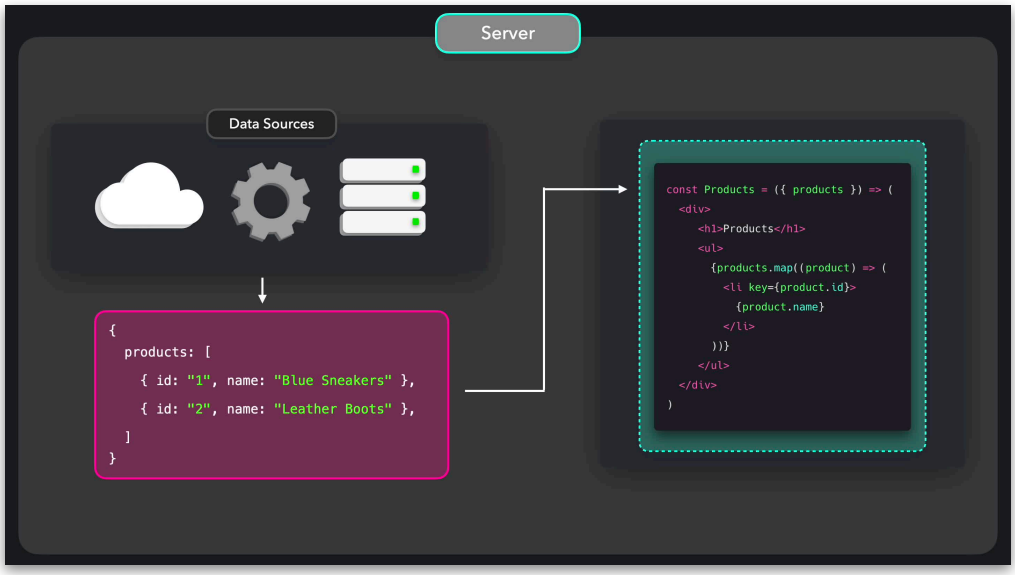
3. **Application shell caching with service workers**: This technique involves caching the application shell which is the minimal HTML, CSS, and JavaScript powering a user interface. Service workers can be used to cache the application shell offline. This can be useful in providing a native single-page app experience where the remaining content is loaded progressively as needed.

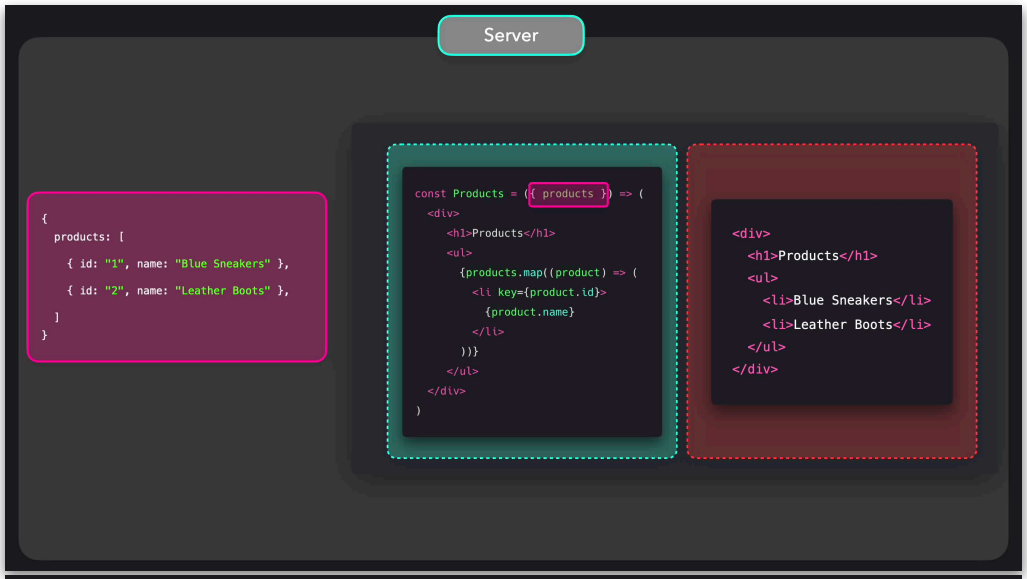
With these techniques, CSR can help to provide a faster Single-Page Application experience with a decent FCP and TTI. Next, we will see what is available at the other end of the spectrum with Server-Side Rendering.

Server-side Rendering

Generate HTML to be rendered on the server in response to a user request

Server-side rendering (SSR) is one of the oldest methods of rendering web content. SSR generates the full HTML for the page content to be rendered in response to a user request. The content may include data from a datastore or external API.




Server
{
products: [
{ id: "1", name: "Blue Sneakers" },
{ id: "2", name: "Leather Boots" },
]
}
const Products = ({ products }) => (

<h1>Products</h1>

{products.map((product) => (
<li key={product.id}>
{product.name}

))}

)

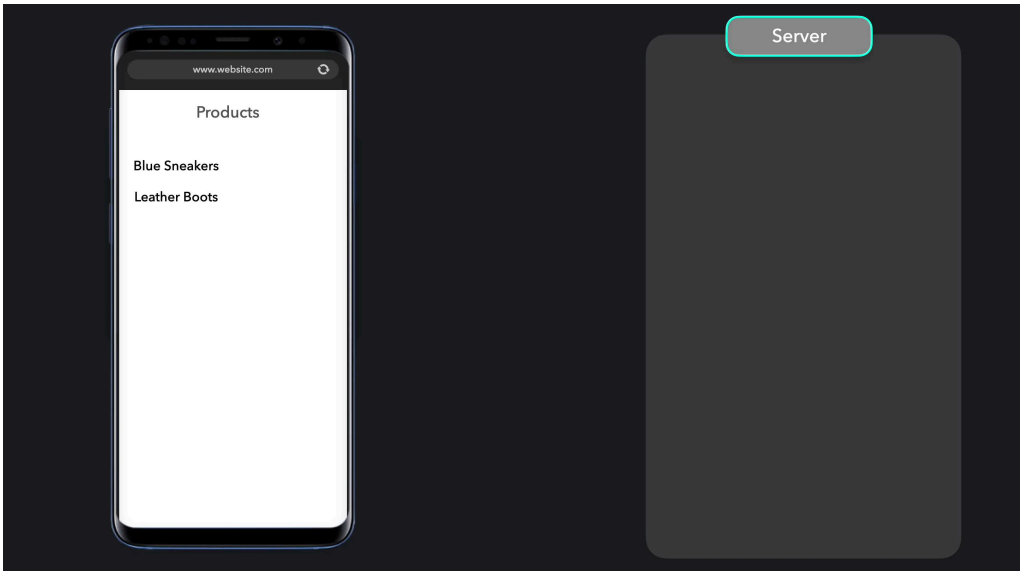
<h1>Products</h1>

Blue Sneakers
Leather Boots


www.website.com
GET /
Server

<h1>Products</h1>

Blue Sneakers
Leather Boots


www.website.com
Products
Blue Sneakers
Leather Boots
Server

The connect and fetch operations are handled on the server. HTML required to format the content is also generated on the server. Thus, with SSR we can avoid making additional round trips for data fetching and templating. As such, rendering code is not required on the client and the JavaScript corresponding to this need not be sent to the client.

With SSR every request is treated independently and will be processed as a new request by the server. Even if the output of two consecutive requests is not very different, the server will process and generate it from scratch. Since the server is common to multiple users, the processing capability is shared by all active users at a given time.

Classic SSR Implementation

Let us see how you would create a page for displaying the current time using classic SSR and JavaScript.

index.html

```
<!DOCTYPE html>
<html>
   <head>
       <title>Time</title>
   </head>
   <body>
       <div>
       <h1>Hello, world!</h1>
       <b>It is <div id=currentTime></div></b>
       </div>
   </body>
</html>
```

index.js

```
function tick() {
    var d = new Date();
    var n = d.toLocaleTimeString();
    document.getElementById("currentTime").innerHTML = n;
}
setInterval(tick, 1000);
```

Note how this is different from the CSR code that provides the same output. Also note that, while the HTML is rendered by the server, the time displayed here is the local time on the client as populated by the JavaScript function `tick()`. If you want to display any other data that is server specific, e.g., server time, you will need to embed it in the HTML before it is rendered. This means it will not get refreshed automatically without a round trip to the server.

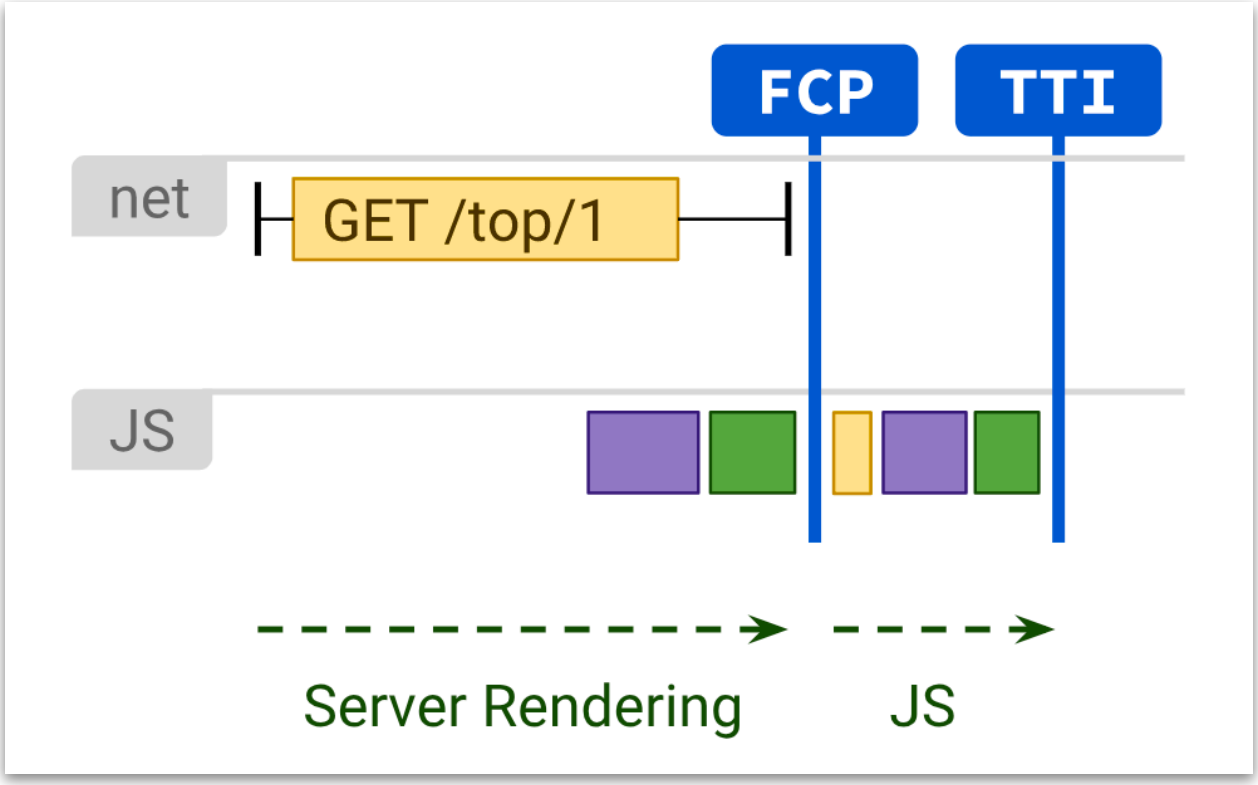
SSR - Pros and Cons

Executing the rendering code on the server and reducing JavaScript offers the following advantages.

Lesser JavaScript leads to quicker FCP and TTI

In cases where there are multiple UI elements and application logic on the page, SSR has considerably less JavaScript when compared to CSR. The

time required to load and process the script is thus lesser. FP, FCP and TTI are shorter and FCP = TTI. With SSR, users will not be left waiting for all the screen elements to appear and for it to become interactive.



Provides additional budget for client-side JavaScript

Development teams are required to work with a JS budget that limits the amount of JS on the page to achieve the desired performance. With SSR, since you are directly eliminating the JS required to render the page, it creates additional space for any third party JS that may be required by the application.

SEO enabled

Search engine crawlers are easily able to crawl the content of an SSR application thus ensuring higher search engine optimization on the page. SSR works great for static content due to the above advantages. However, it does have a few disadvantages because of which it is not perfect for all scenarios.

Slow TTFB

Since all processing takes place on the server, the response from the server may be delayed in case of one or more of the following scenarios

- Multiple simultaneous users causing excess load on the server.
- Slow network
- Server code not optimized.

Full page reloads required for some interactions

Since all code is not available on the client, frequent round trips to the server are required for all key operations causing full page reloads. This could increase the time between interactions as users are required to wait longer between operations. A single-page application is thus not possible with SSR.

To address these drawbacks, modern frameworks and libraries allow rendering on both server and client for the same application. We will go into details of these in the following sections. First, let's look at a simpler form of SSR with Next.js.

SSR with Next.js

The Next.js framework also supports SSR. This pre-renders a page on the server on every request. It can be accomplished by exporting an async function called `getServerSideProps()` from a page as follows.

```
export async function getServerSideProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

The context object contains keys for HTTP request and response objects, routing parameters, querystring, locale, etc.

React for the Server

React can be rendered isomorphically, which means that it can function both on the browser as well as other platforms like the server. Thus, UI elements may be rendered on the server using React.

React can also be used with universal code which will allow the same code to run in multiple environments. This is made possible by using Node.js on the

server or what is known as a Node server. Thus, universal JavaScript may be used to fetch data on the server and then render it using isomorphic React.

Let us take a look at the react functions that make this possible.

```
ReactDOMServer.renderToString(element)
```

This function returns an HTML string corresponding to the React element. The HTML can then be rendered to the client for a faster page load.

The `renderToString()` function may be used with `ReactDOM.hydrate()`. This will ensure that the rendered HTML is preserved as-is on the client and only the event handlers attached after load.

To implement this, we use a `.js` file on both client and server corresponding to every page. The `.js` file on the server will render the HTML content, and the `.js` file on the client will hydrate it.

Assume you have a React element called App which contains the HTML to be rendered defined in the universal `app.js` file. Both the server and client-side React can recognize the App element.

The `ipage.js` file on the server can have the code:

```
app.get('/', (req, res) => {
  const app = ReactDOMServer.renderToString(<App />);
})
```

The constant App can now be used to generate the HTML to be rendered. The ipage.js on the client side will have the following to ensure that the element App is hydrated.

```
ReactDOM.hydrate(<App />, document.getElementById('root'));
```

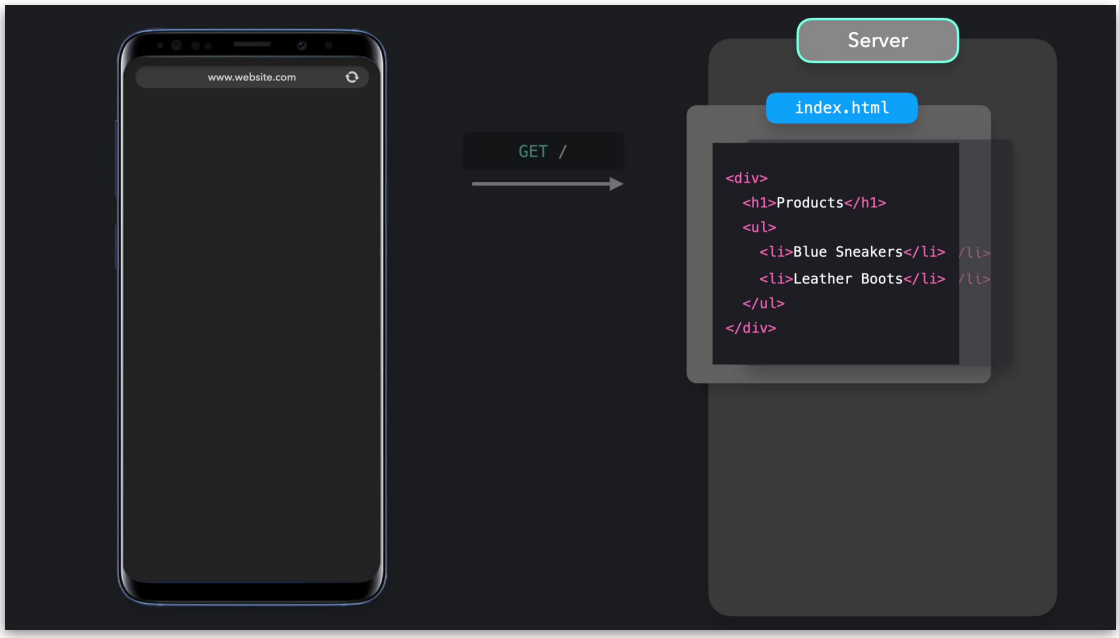
Static Rendering

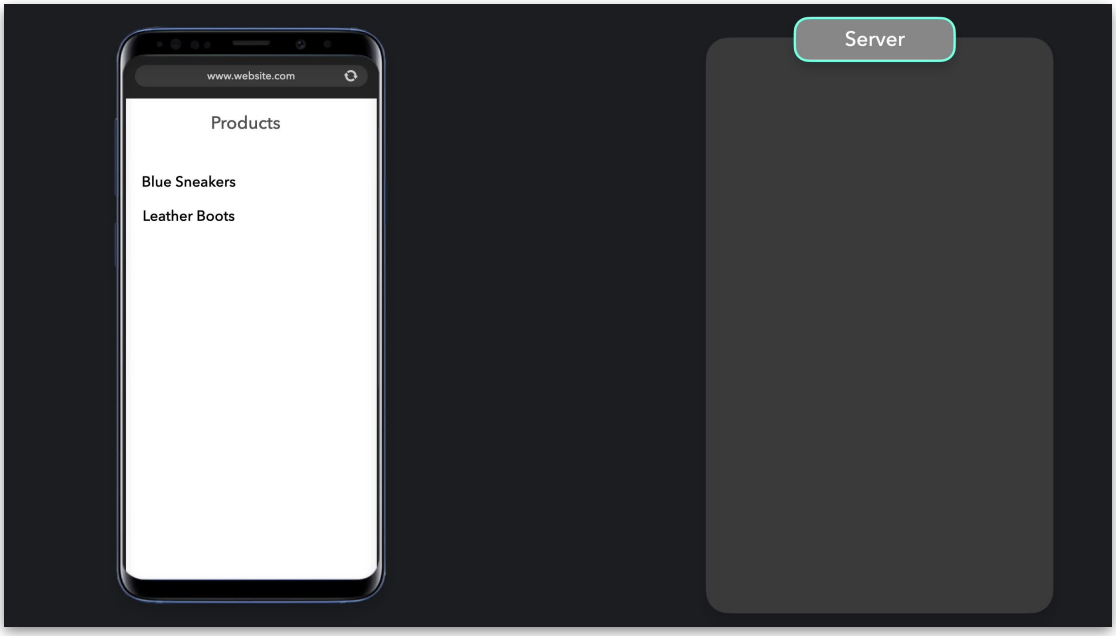
Deliver pre-rendered HTML content that was generated when the site was built

Based on our discussion on SSR, we know that a high request processing time on the server negatively affects the TTFB. Similarly, with CSR, a large JavaScript bundle can be detrimental to the FCP, LCP and TTI of the application due to the time taken to download and process the script.

Static rendering or static generation (SSG) attempts to resolve these issues by delivering pre-rendered HTML content to the client that was generated when the site was built.

A static HTML file is generated ahead of time corresponding to each route that the user can access. These static HTML files may be available on a server or a CDN and fetched as and when requested by the client.





Static files may also be cached thereby providing greater resiliency. Since the HTML response is generated in advance, the processing time on the server is negligible thereby resulting in a faster TTFB and better performance. In an ideal scenario, client-side JS should be minimal and static pages should become interactive soon after the response is received by the client. As a result, SSG helps to achieve a faster FCP/TTI

Basic Structure

As the name suggests, static rendering is ideal for static content, where the page need not be customized based on the logged-in user (e.g personalized recommendations). Thus static pages like the ‘About us', ‘Contact us', Blog pages for websites or product pages for e-commerce apps, are ideal candidates for static rendering. Frameworks like Next.js, Gatsby, and VuePress support static generation. Let us start with this simple Next.js example of static content rendering without any data.

pages/about.js

```
export default function About() {
 return <div>
   <h1>About Us</h1>
   {/* ... */}
 </div>
}
```

.

When the site is built, this page will be pre-rendered into an HTML file `about.html` accessible at the route `/about`.

SSG with Data

Static content like that in 'About us' or 'Contact us' pages may be rendered as-is without getting data from a data-store. However, for content like individual blog pages or product pages, the data from a data-store has to be merged with a specific template and then rendered to HTML at build time.

The number of HTML pages generated will depend on the number of blog posts or the number of products respectively. To get to these pages, you may also have listing pages which will be HTML pages that contain a categorized and formatted list of data items. These scenarios can be addressed using Next.js static rendering. We can generate listing pages or individual item pages based on the available items. Let us see how.

Listing Page - All Items

Generation of a listing page is a scenario where the content to be displayed on the page depends on external data. This data will be fetched from the database at build time to construct the page. In Next.js this can be achieved by exporting the function getStaticProps() in the page component. The function is called at build time on the build server to fetch the data. The data can then be passed to the page's props to pre-render the page component.

```
// This function runs at build time on the build server
export async function getStaticProps() {
 return {
   props: {
     products: await getProductsFromDatabase()
   }
 }
}

// The page component receives products prop from getStaticProps at build time
export default function Products({ products }) {
 return (
   <>
     <h1>Products</h1>
     <ul>
       {products.map((product) => (
         <li key={product.id}>{product.name}</li>
       ))}
     </ul>
   </>
 )
}
```

The function will not be included in the client-side JS bundle and hence can even be used to fetch the data directly from a database.

Individual Details Page - Per Item

In the above example, we could have an individual detailed page for each of the products listed on the listing page. These pages could be accessed by clicking on the corresponding items on the listing page or directly through some other route.

Assume we have products with product ids 101,102 103, and so on. We need their information to be available at routes /products/101, /products/102, /products/103 etc. To achieve this at build time in Next.js we can use the function getStaticPaths() in combination with dynamic routes.
We need to create a common page component products/[id].js for this and export the function getStaticPaths() in it. The function will return all possible product ids which can be used to pre-render individual product pages at build time. The following Next.js skeleton available here shows how to structure the code for this.

pages/products/[id].js

```
// In getStaticPaths(), you need to return the list of
// ids of product pages (/products/[id]) that you'd
// like to pre-render at build time. To do so,
// you can fetch all products from a database.
export async function getStaticPaths() {
 const products = await getProductsFromDatabase()

 const paths = products.map((product) => ({
   params: { id: product.id }
 }))

 // fallback: false means pages that don't have the correct id will 404.
 return { paths, fallback: false }
}

// params will contain the id for each generated page.
export async function getStaticProps({ params }) {
 return {
   props: {
     product: await getProductFromDatabase(params.id)
   }
 }
}

export default function Product({ product }) {
 // Render product
}
```

The details on the product page may be populated at build time by using the function getStaticProps for the specific product id. Note the use of the fallback: false indicator here. It means that if a page is not available corresponding to a specific route or product Id, the 404 error page will be shown.
Thus we can use SSG to pre-render many different types of pages.

Key Considerations

As discussed, SSG results in a great performance for websites as it cuts down the processing required both on the client and the server. The sites are also SEO friendly as the content is already there and can be rendered by web-crawlers with no extra effort. While performance and SEO make SSG a great rendering pattern, the following factors need to be considered when assessing the suitability of SSG for specific applications.

1. **A large number of HTML files:** Individual HTML files need to be generated for every possible route that the user may access. For example, when using it for a blog, an HTML file will be generated for every blog post available in the data store. Subsequently, edits to any of the posts will require a rebuild for the update to be reflected in the static HTML files. Maintaining a large number of HTML files can be challenging.

2. **Hosting Dependency**: For an SSG site to be super-fast and respond quickly, the hosting platform used to store and serve the HTML files should also be good. Superlative performance is possible if a well-tuned SSG website is hosted right on multiple CDNs to take advantage of edge-caching.

3. **Dynamic Conten**t: An SSG site needs to be built and re-deployed every time the content changes. The content displayed may be stale if the site has not been built + deployed after any content change. This makes SSG unsuitable for highly dynamic content.

Incremental Static Generation

Update static content after you have built your site

Static Generation (SSG) addresses most of the concerns of SSR and CSR but is suitable for rendering mostly static content. It poses limitations when the content to be rendered is dynamic or changing frequently.

Think of a growing blog with multiple posts. You wouldn't possibly want to rebuild and redeploy the site just because you want to correct a typo in one of the posts. Similarly, one new blog post should also not require a rebuild for all the existing pages. Thus, SSG on its own is not enough for rendering large websites or applications.

The Incremental Static Generation (iSSG) pattern was introduced as an upgrade to SSG, to help solve the dynamic data problem and help static sites scale for large amounts of frequently changing data. iSSG allows you to update existing pages and add new ones by pre-rendering a subset of pages in the background even while fresh requests for pages are coming in.

Sample Code

iSSG works on two fronts to incrementally introduce updates to an existing static site after it has been built.
1. Allows addition of new pages
2. Allows updates to existing pages also known as Incremental Static “Re"generation

Adding New pages

The lazy loading concept is used to include new pages on the website after the build. This means that the new page is generated immediately on the first request. While the generation takes place, a fallback page or a loading indicator can be shown to the user on the front-end. Compare this to the SSG scenario discussed earlier for individual details page per product. The 404 error page was shown here as a fallback for non-existent pages.

Let us now look at the Next.js code required for lazy-loading the non-existent page with iSSG.

pages/products/[id].js

```
// In getStaticPaths(), you need to return the list of
// ids of product pages (/products/[id]) that you'd
// like to pre-render at build time. To do so,
// you can fetch all products from a database.
export async function getStaticPaths() {
  const products = await getProductsFromDatabase();

  const paths = products.map((product) => ({
     params: { id: product.id }
  }));


  // fallback: true means that the missing pages
  // will not 404, and instead can render a fallback.
  return { paths, fallback: true };
}

// params will contain the id for each generated page.
export async function getStaticProps({ params }) {
  return {
    props: {
      product: await getProductFromDatabase(params.id)
    }
  }
}


export default function Product({ product }) {
  const router = useRouter();

  if (router.isFallback) {
    return <div>Loading...</div>;
  }

  // Render product
}
```

Here, we have used `fallback: true`. Now if the page corresponding to a specific product is unavailable, we show a fallback version of the page, eg., a loading indicator as shown in the Product function above. Meanwhile, Next.js will generate the page in the background. Once it is generated, it will be cached and shown instead of the fallback page. The cached version of the page will now be shown to any subsequent visitors immediately upon request. For both new and existing pages, we can set an expiration time for when Next.js should revalidate and update it. This can be achieved by using the revalidate property as shown in the following section.

Update Existing pages

To re-render an existing page, a suitable timeout is defined for the page. This will ensure that the page is revalidated whenever the defined timeout period has elapsed. The timeout could be set to as low as 1 second. The user will continue to see the previous version of the page, till the page has finished revalidation. Thus, iSSG uses the stale-while-revalidate strategy where the user receives the cached or stale version while the revalidation takes place. The revalidation takes place completely in the background without the need for a full rebuild.

Let us go back to the example for generating a static listing page for products based on the data in the database. To make it serve a relatively dynamic list of products, we will include the code to set the timeout for rebuilding the page. This is what the code will look like after including the timeout.

pages/products/[id].js

```
// This function runs at build time on the build server
export async function getStaticProps() {
  return {
    props: {
      products: await getProductsFromDatabase(),
      revalidate: 60, // This will force the page to revalidate after 60 seconds
    }
  }
}

// The page component receives products prop from getStaticProps at build time
export default function Products({ products }) {
  return (
    <>
      <h1>Products</h1>
      <ul>
        {products.map((product) => (
          <li key={product.id}>{product.name}</li>
        ))}
      </ul>
    </>
  )
}
```

The code to revalidate the page after 60 seconds is included in the getStaticProps() function. When a request comes in the available static page is served first. Every one minute the static page gets refreshed in the background with new data. Once generated, the new version of the static file becomes available and will be served for any new requests in the subsequent minute. This feature is available in Next.js 9.5 and above.

iSSG Advantages

iSSG provides all the advantages of SSG and then some more. The following list covers them in detail.

1. **Dynamic data**: The first advantage is obviously why iSSG was envisioned. Its ability to support dynamic data without a need to rebuild the site.

2. **Speed**: iSSG is at least as fast as SSG because data retrieval and rendering still takes place in the background. There is little processing required on the client or the server.

3. **Availability**: A fairly recent version of any page will always be available online for users to access. Even if the regeneration fails in the background, the old version remains unaltered.

4. **Consistent**: As the regeneration takes place on the server one page at a time, the load on the database and the backend is low and performance is consistent. As a result, there are no spikes in latency.

5. **Ease of Distribution**: Just like SSG sites, iSSG sites can also be distributed through a network of CDN's used to serve pre-rendered web pages.

Progressive Hydration

Delay loading JavaScript for less important parts of the page

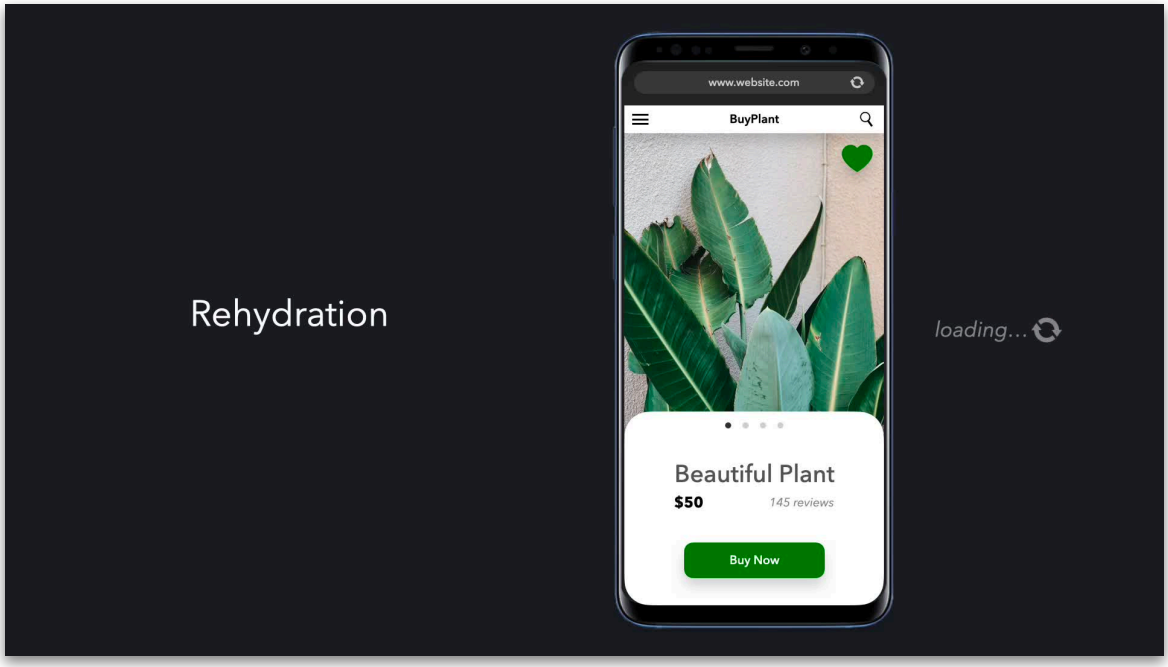
A server rendered application uses the server to generate the HTML for the current navigation. Once the server has completed generating the HTML contents, which also contains the necessary CSS and JSON data to display the static UI correctly, it sends the data down to the client. Since the server generated the markup for us, the client can quickly parse this and display it on the screen, which produces a fast First Contentful Paint!

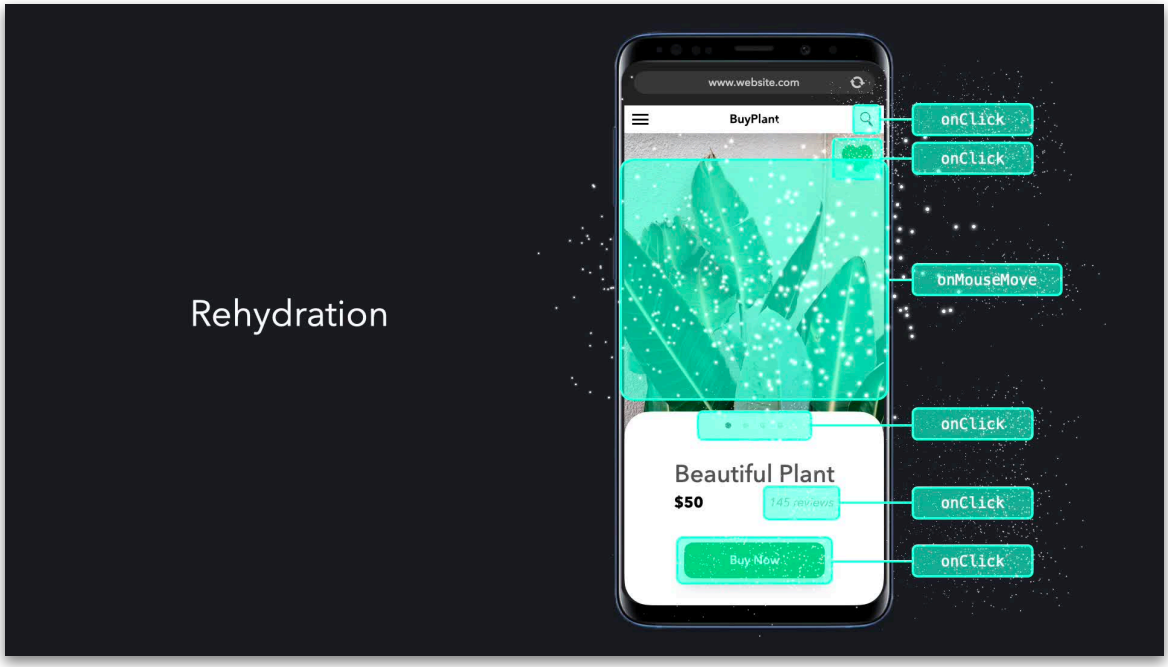
Although server rendering provides a faster First Contentful Paint, it doesn't always provide a faster Time To Interactive. The necessary JavaScript in order to be able to interact with our website hasn't been loaded yet.
Buttons may look interactive, but they aren't interactive (yet). The handlers will only get attached once the JavaScript bundle has
been loaded and processed. This process is called hydration: React checks the current DOM nodes, and hydrates the nodes with the corresponding JavaScript.

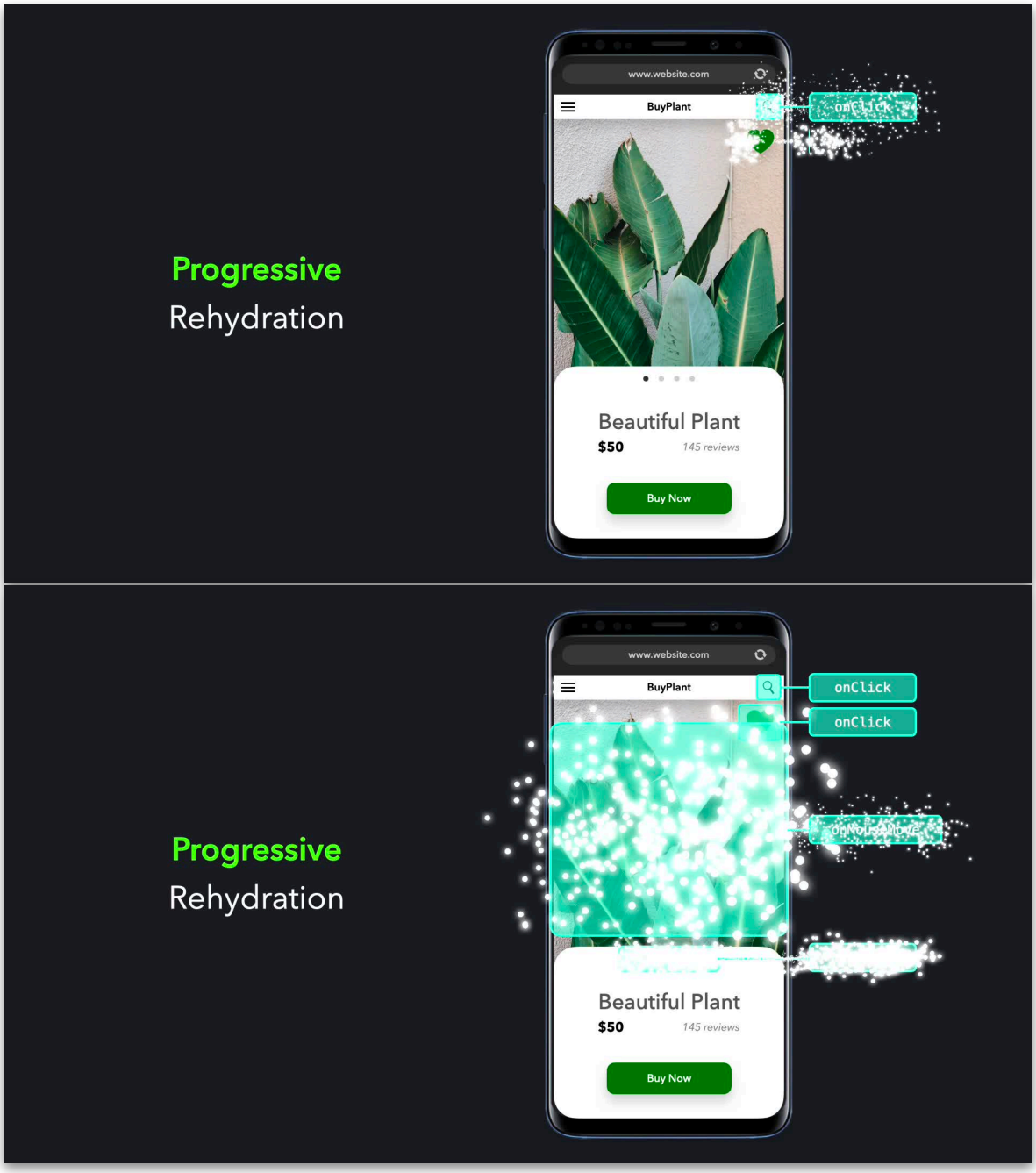
The time that the user sees non-interactive UI on the screen is also refered to as the uncanny valley: although users may think that they can interact with the website, there are no handlers attached to the components yet. This can be a frustrating experience for the user, as the UI may can like it's frozen!

It can take a while before the DOM components that were received from the server are fully hydrated. Before the components can be hydrated, the

JavaScript file needs to be loaded, processed, and executed. Instead of hydrating the entire application at once, like we did previously, we can also progressively hydrate the DOM nodes. Progressive hydration makes it possible to individually hydrate nodes over time, which makes it possible to only request the minimum necessary JavaScript.

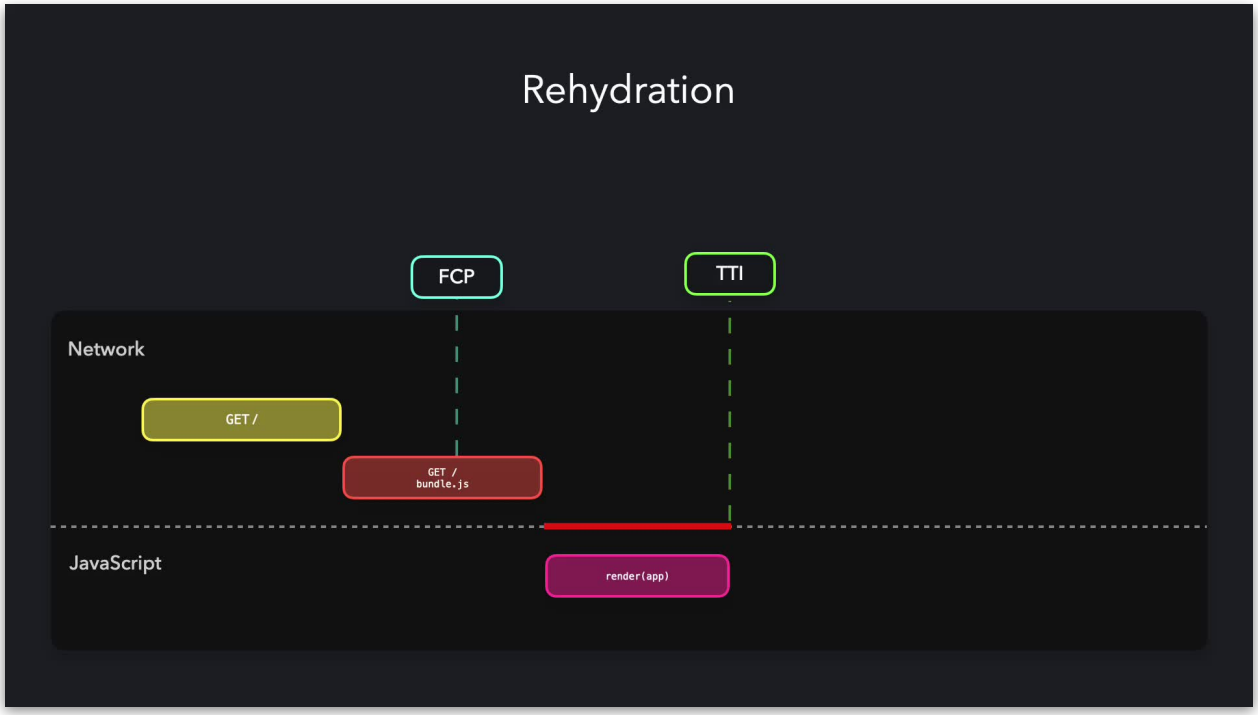


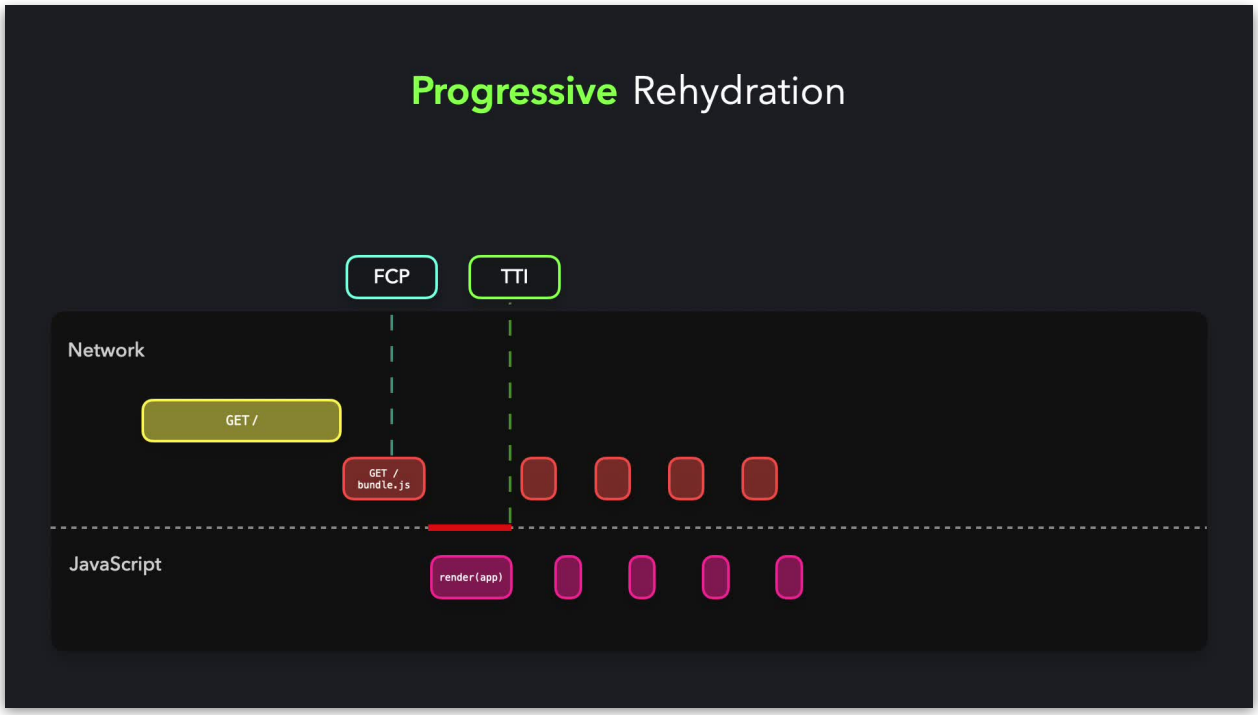




By progressively hydrating the application, we can delay the hydration of less important parts of the page. This way, we can reduce the amount of JavaScript we have to request in order to make the page interactive, and only hydrate the nodes once the user needs it. Progressive hydration also

helps avoid the most common SSR Rehydration pitfalls where a server-rendered DOM tree gets destroyed and then immediately rebuilt.





Progressive hydration allows us to only hydrate components based on a certain condition, for example when a component is visible in the viewport.

In the following example, we have a list of users that gets progressively hydrated once the list is in the viewport. The purple flash shows when the component has been hydrated!

client.js

```
import React from "react";
import { hydrate } from "react-dom";
import App from "./components/App";

hydrate(<App />, document.getElementById("root"));
```

server.js

```
import React from "react";
import { renderToNodeStream } from "react-dom/server";
import App from "./components/App";

export default async () => renderToNodeStream(<App />);
```

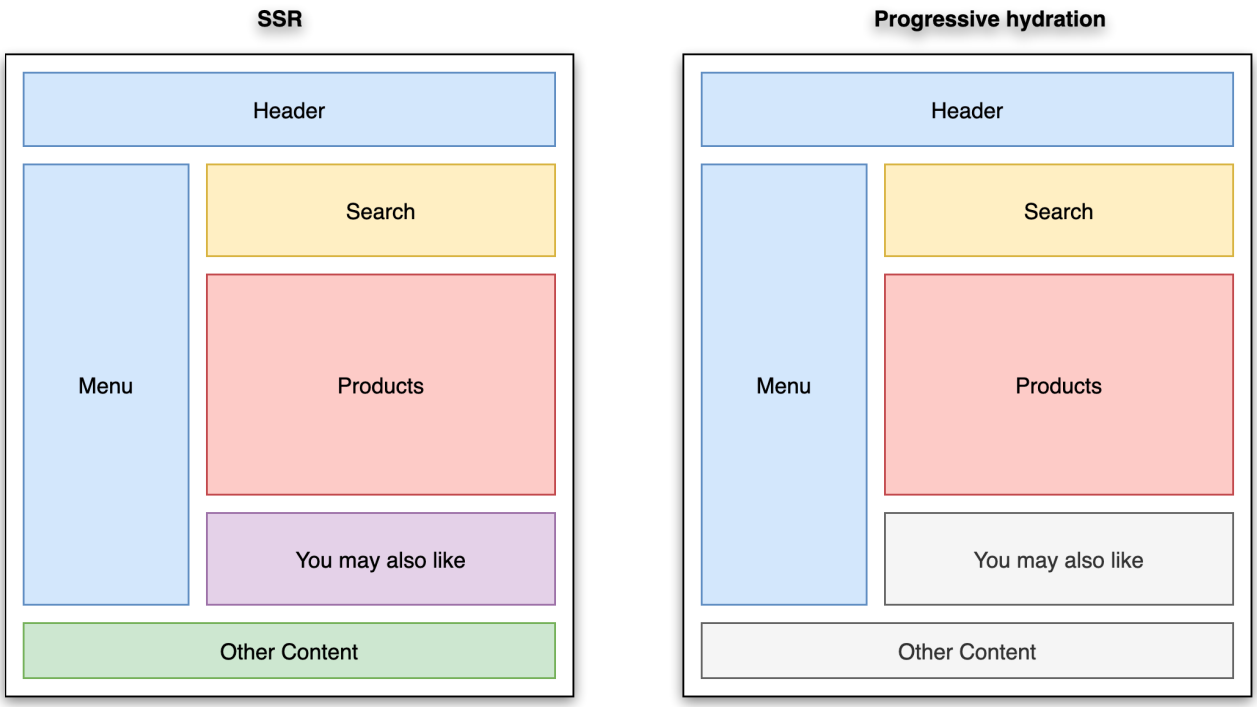
Progressive Hydration Implementation

In the section on implementing SSR with React, we discussed client-side hydration for an app that is rendered on the server. Hydration allows client-side React to recognize the ReactDOM components that are rendered on the server and attach events to these components. Thus, it introduces continuity and seamlessness for an SSR app to function like a CSR app once it is

available on the client.

For all components on the page to become interactive via hydration, the React code for these components should be included in the bundle that gets downloaded to the client. Highly interactive SPAs that are largely controlled by JavaScript would need the entire bundle at once. However, mostly static websites with a few interactive elements on the screen, may not need all components to be active immediately. For such websites sending a huge React bundle for each component on the screen becomes an overhead.

Progressive Hydration solves this problem by allowing us to hydrate only certain parts of the application when the page loads. The other parts are hydrated progressively as required.



With Progressive hydration, the "You may also like" and "Other content" components can be hydrated later.

Instead of initializing the entire application at once, the hydration step begins at the root of the DOM tree, but the individual pieces of the server-rendered application are activated over a period of time. The hydration process may be arrested for various branches and resumed later when they enter the viewport or based on some other trigger. Note that, the loading of resources required to perform each hydration is also deferred using code-splitting techniques, thereby reducing the amount of JavaScript required to make pages interactive.

The idea behind progressive hydration is to provide a great performance by activating your app in chunks. Any progressive hydration solution should also take into account how it will impact the overall user experience. You cannot have chunks of screen popping up one after the other but blocking any activity or user input on the chunks that have already loaded. Thus, the requirements for a holistic progressive hydration implementation are as follows.

- Allows usage of SSR for all components.
- Supports splitting of code into individual components or chunks.
- Supports client side hydration of these chunks in a developer defined sequence.
- Does not block user input on chunks that are already hydrated.\
- Allows usage of some sort of a loading indicator for chunks with deferred hydration.

React concurrent mode will address all these requirements once it is available to all. It allows React to work on different tasks at the same time and switch between them based on the given priority. When switching, a partially rendered tree need not be committed, so that the rendering task can continue once React switches back to the same task.

Concurrent mode can be used to implement progressive hydration. In this case, hydration of each of the chunks on the page, becomes a task for React concurrent mode. If a task of higher priority like user input needs to be performed, React will pause the hydration task and switch to accepting the user input. Features like lazy(), Suspense() allow you to use declarative loading states. These can be used to show the loading indicator while chunks are being lazy loaded. SuspenseList() can be used to define the priority for lazy loading components.This demo shared by Dan Abramov shows concurrent mode in action and implements progressive hydration.

React concurrent mode can also be combined with another React feature **Server Components**. This will allow you to refetch components from the server and render them on the client as they stream in instead of waiting for the whole fetch to finish. Thus, the client's CPU is put to work even as we wait for the network fetch to finish.

While the React concurrent mode based progressive hydration implementation is still getting ready, many other contenders for a partial hydration implementation are available. Progressive hydration was demonstrated at Google I/O '19. The demo for progressive hydration showed the use of a Hydrator component to hydrate selected sections of the page.

Multiple implementations have spawned from this for different client-side frameworks. Implementations are also available for Vue, Angular and Next.js. Let is take a quick look at one such method using Preact and Next.js

This is a POC for partial hydration using

- `pool-attendant-preact`: A library that implements partial hydration with preact x.
- `next-super-performance`: A Next.js plugin that uses this library to improve client-side performance.

The `pool-attendant-preact` library includes an API called withHydration which lets you mark your more interactive components for hydration. These will be hydrated first. You can use this to define your page content as follows.

```
import Teaser from "./teaser";
import { withHydration } from "next-super-performance";

const HydratedTeaser = withHydration(Teaser);

export default function Body() {
 return (
   <main>
     <Teaser column={1} />
     <HydratedTeaser column={2} />
     <HydratedTeaser column={3} />

     <Teaser column={1} />
     <Teaser column={2} />
     <Teaser column={3} />

     <Teaser column={1} />
     <Teaser column={2} />
     <Teaser column={3} />
   </main>
 );
}
```

The component HydratedTeaser in columns 2 and 3 will be hydrated first. You can now hydrate the remaining components on the client using the hydrate() API which is also included in the library.

```
import { hydrate } from "next-super-performance";
import Teaser from "./components/teaser";

hydrate([Teaser]);
```

The component HydrationData is used to write serialized props to the client. It will ensure that the required props are available to the components being hydrated.

```
import Header from "../components/header";
import Main from "../components/main";
import { HydrationData } from "next-super-performance";

export default function Home() {
 return (
   <section>
     <Header />
     <Main />
     <HydrationData />
   </section>
 );
}
```

Pros and Cons

Progressive hydration provides server-side rendering with client-side hydration while also minimizing the cost of hydration. Following are some of the advantages that can be gained from this.

1. **Promotes code-splitting**: Code-splitting is an integral part of progressive hydration because chunks of code need to be created for individual components that are lazy- loaded.

2. **Allows on-demand loading for infrequently used parts of the page**: There may be components of the page that are mostly static, out of the viewport and/or not required very often. Such components are ideal candidates for lazy loading. Hydration code for these components need not be sent when the page loads. Instead, they may be hydrated based on a trigger.

3. **Reduces bundle size**: Code-splitting automatically results in a reduction of bundle size. Less code to execute on load helps reduce the time between FCP and TTI.

On the downside, progressive hydration may not be suitable for dynamic apps where every element on the screen is available to the user and needs to be made interactive on load. This is because, if developers do not know where the user is likely to click first, they may not be able to identify which components to hydrate first.

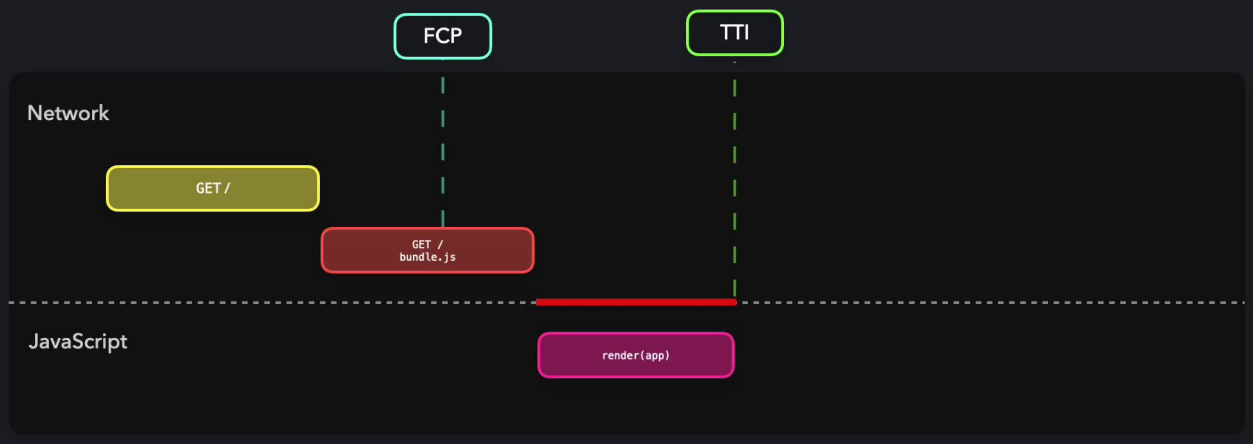
Streaming Server-Side Rendering

Generate HTML to be rendered on the server in response to a user request

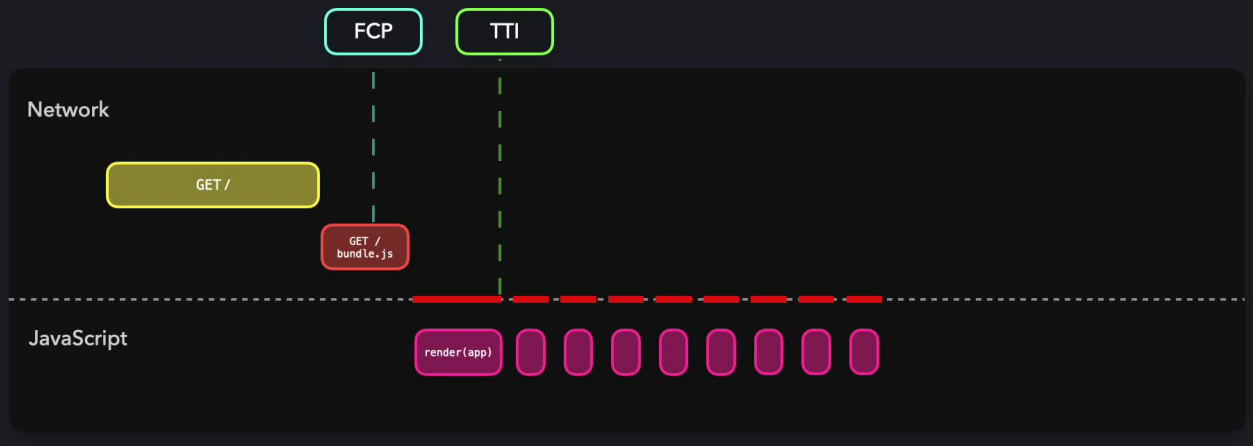
We can reduce the TTI while still server rendering our application by streaming server rendering the contents of our application. Instead of generating one large HTML file containing the necessary markup for the current navigation, we can split it up into smaller chunks! Node streams allow us to data into the response object, which means that we can continuously send data down to the client. The moment the client receives the chunks of data, it can start rendering the contents.

React's built-in renderToNodeStream makes it possible for us to send our application in smaller chunks. As the client can start painting the UI when it's still receiving data, we can create a very performant first-load experience. Calling the hydrate method on the received DOM nodes will attach the corresponding event handlers, which makes the UI interactive!

Server Rendering



Streaming Server Rendering



```
import React from "react";
import path from "path";
import express from "express";
import { renderToNodeStream } from "react-dom/server";

import App from "./src/App";

const app = express();

app.get("/favicon.ico", (req, res) => res.end());
app.use("/client.js", (req, res) => res.redirect("/build/client.js"));

const DELAY = 500;
app.use((req, res, next) => {
  setTimeout(() => next(), DELAY);
});

const BEFORE = `
<!DOCTYPE html>
  <html>
    <head>
      <title>Cat Facts</title>
      <link rel="stylesheet" href="/style.css">
      <script type="module" defer src="/build/client.js"></script>
    </head>
    <body>
      <h1>Stream Rendered Cat Facts!</h1>
      <div id="approot">
`.replace(/
s*/g, "");
```

```
app.get("/", async (request, response) => {
  try {
    const stream = renderToNodeStream(<App />);
    const start = Date.now();

    stream.on("data", function handleData() {
      console.log("Render Start: ", Date.now() - start);
      stream.off("data", handleData);
      response.useChunkedEncodingByDefault = true;
      response.writeHead(200, {
        "content-type": "text/html",
        "content-transfer-encoding": "chunked",
        "x-content-type-options": "nosniff"
      });
      response.write(BEFORE);
      response.flushHeaders();
    });
    await new Promise((resolve, reject) => {
      stream.on("error", err => {
        stream.unpipe(response);
        reject(err);
      });
      stream.on("end", () => {
        console.log("Render End: ", Date.now() - start);
        response.write("</div></body></html>");
        response.end();
        resolve();
      });
      stream.pipe(response, { end: false });
    });
  } catch (err) {
    response.writeHead(500, { "content-type": "text/plain" });
    response.end(String((err && err.stack) || err));
    return;
  }
});

app.use(express.static(path.resolve(__dirname, "src")));
app.use("/build", express.static(path.resolve(__dirname, "build")));
```

Let's say we have an app that shows the user thousands of cat facts in the App component!

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cat Facts</title>
    <link rel="stylesheet" href="/style.css" />
    <script type="module" defer src="/build/client.js"></script>
  </head>
  <body>
    <h1>Stream Rendered Cat Facts!</h1>
    <div id="approot"></div>
  </body>
</html>
```

The App component gets stream rendered using the built-in renderToNodeStream method. The initial HTML gets sent to the response object alongside the chunks of data from the App component,

This data contains useful information that our app has to use in order to render the contents correctly, such as the title of the document and a stylesheet. If we were to server render the App component using the renderToString method, we would have had to wait until the application has received all data before it can start loading and processing this metadata. To speed this up, renderToNodeStream makes it possible for the app to start loading and processing this information as it's still receiving the chunks of data from the App component!

Concepts

Like progressive hydration, streaming is another rendering mechanism that can be used to improve SSR performance. As the name suggests, streaming implies chunks of HTML are streamed from the node server to the client as they are generated. As the client starts receiving "bytes" of HTML earlier even for large pages, the TTFB is reduced and relatively constant. All major browsers start parsing and rendering streamed content or the partial response earlier. As the rendering is progressive, it results in a fast FP and FCP. Streaming responds well to network backpressure. If the network is clogged and not able to transfer any more bytes, the renderer gets a signal and stops streaming till the network is cleared up. Thus, the server uses less memory and is more responsive to I/O conditions. This enables your Node.js server to render multiple requests at the same time and prevents heavier requests from blocking lighter requests for a long time. As a result, the site stays responsive even in challenging conditions.

React for streaming

React introduced support for streaming in React 16 released in 2016. The following API's were included in the `ReactDOMServer` to support streaming.

- `ReactDOMServer.renderToNodeStream(element)`: The output HTML from this function is the same as `ReactDOMServer.renderToNodeStream(element)` but is in a Node.js readablestream format instead of a string. The function will only work on the server to render HTML as a stream. The client receiving this

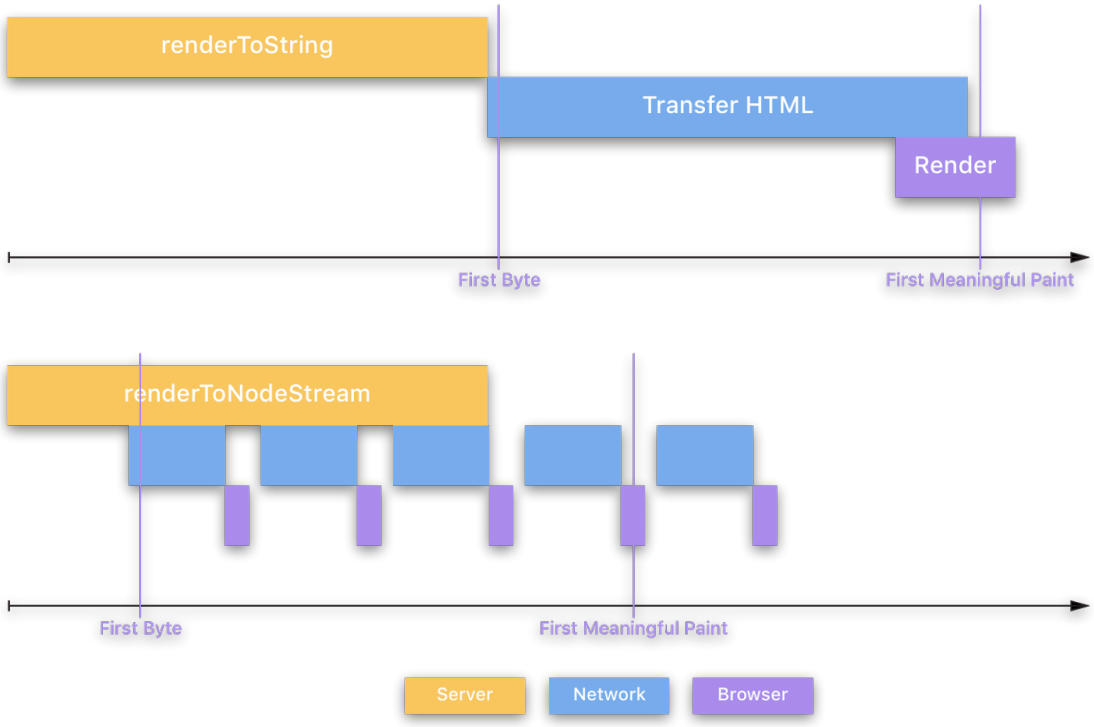
stream can subsequently call `ReactDOM.hydrate()` to hydrate the page and make it interactive.

- `ReactDOMServer.renderToStaticNodeStream(element)`: This corresponds to `ReactDOMServer.renderToStaticNodeStream(element)`. The HTML output is the same but in a stream format. It can be used for rendering static, non-interactive pages on the server and then streaming them to the client.

```
import { renderToNodeStream } from 'react-dom/server';
import Frontend from '../client';

app.use('*', (request, response) => {
  // Send the start of your HTML to the browser
  response.write('<html><head><title>Page</title></head><body><div id="root">');

  // Render your frontend to a stream and pipe it to the response
  const stream = renderToNodeStream(<Frontend />);
  stream.pipe(response, { end: 'false' });
  // Tell the stream not to automatically end the response when the renderer
finishes.

  // When React finishes rendering send the rest of your HTML to the browser
  stream.on('end', () => {
    response.end('</div></body></html>');
  });
});
```

The readable stream output by both functions can emit bytes once you start reading from it. This can be achieved by piping the readable stream to a writable stream such as the response object. The response object progressively sends chunks of data to the client while waiting for new chunks to be rendered.

A comparison between TTFB and First Meaningful Paint for normal SSR Vs Streaming is available in the following image.



Pros and cons

Streaming aims to improve the speed of SSR with React and provides the following benefits

1. **Performance Improvement**: As the first byte reaches the client soon after rendering starts on the server, the TTFB is better than that for SSR. it is also more consistent irrespective of the page size. Since the client can start parsing HTML as soon as it receives it, the FP and FCP are also lower.

2. **Handling of Backpressure**: Streaming responds well to network backpressure or congestion and can result in responsive websites even under challenging conditions.

3. **Supports SEO**: The streamed response can be read by search engine crawlers, thus allowing for SEO on the website.

It is important to note that streaming implementation is not a simple find-replace from `renderToString` to `renderToNodeStream`(). There are cases where the code that works with SSR may not work as-is with streaming. Following are some examples where migration may not be easy.

1. Frameworks that use the server-render-pass to generate markup that needs to be added to the document before the SSR-ed chunk. Examples are frameworks that dynamically determine which CSS to add to the page in a preceding `<style>` tag, or frameworks that add elements to the document `<head>` while rendering. A workaround for this has been discussed here.

2. Code, where `renderToStaticMarkup` is used to generate the page template and `renderToString` calls are embedded to generate dynamic content. Since the string corresponding to the component is expected in these cases, it cannot be replaced by a stream. An example of such code provided here is as follows.

Both Streaming and Progressive Hydration can help to bridge the gap between a pure SSR and a CSR experience. Let us now compare all the patterns that we have explored and try to understand their suitability for different situations.

React Server Components

Server Components compliment SSR, rendering to an intermediate abstraction without needing to add to the JavaScript bundle

The React team are working on zero-bundle-size React Server Components, which aim to enable modern UX with a server-driven mental model. This is quite different to Server-side Rendering (SSR) of components and could result in significantly smaller client-side JavaScript bundles.

The direction of this work is exciting, and while it isn't yet production ready, is worth keeping on your radar. The RFC is worth reading as is Dan and Lauren's talk worth watching for more detail.

Server-side rendering limitations

Today's Server-side rendering of client-side JavaScript can be suboptimal. JavaScript for your components is rendered on the server into an HTML string. This HTML is delivered to the browser, which can appear to result in a fast First Contentful Paint or Largest Contentful Paint.

However, JavaScript still needs to be fetched for interactivity which is often achieved via a hydration step. Server-side rendering is generally used for the initial page load, so post-hydration you're unlikely to see it used again.

Note: While it's true that one could build a server-only React app leveraging SSR and avoiding hydrating on the client at all, heavy interactivity in the model often involves stepping outside of React. The hybrid model that Server Components enable will allow deciding this on a per-component basis.
With React Server Components, our components can be refetched regularly. An application with components which rerender when there is new data can be run on the server, limiting how much code needs to be sent to the client.

```
// *Before* Server Components
import marked from "marked"; // 35.9K (11.2K gzipped)
import sanitizeHtml from "sanitize-html"; // 206K (63.3K gzipped)

function NoteWithMarkdown({text}) {
  const html = sanitizeHtml(marked(text));
  return (/* render */);
}
```

Server Components

React's new Server Components compliment Server-side rendering, enabling rendering into an intermediate abstraction format without needing to add to the JavaScript bundle. This both allows merging the server-tree with the client-side tree without a loss of state and enables scaling up to more components.

Server Components are not a replacement for SSR. When paired together, they support quickly rendering in an intermediate format, then having Server-

side rendering infrastructure rendering this into HTML enabling early paints to still be fast. We SSR the Client components which the Server components emit, similar to how SSR is used with other data-fetching mechanisms.

This time however, the JavaScript bundle will be significantly smaller. Early explorations have shown that bundle size wins could be significant (-18-29%), but the React team will have a clearer idea of wins in the wild once further infrastructure work is complete.

```
import marked from "marked"; // zero bundle size
import sanitizeHtml from "sanitize-html"; // zero bundle size

function NoteWithMarkdown({text}) {
  // same as before
}
```

Automatic Code-Splitting

It's been considered a best-practice to only serve code users need as they need it by using code-splitting. This allows you to break your app down into smaller bundles requiring less code to be sent to the client. Prior to Server Components, one would manually use React.lazy() to define "split-points" or rely on a heuristic set by a meta-framework, such as routes/pages to create new chunks.

Some of the challenges with code-splitting are:

- Outside of a meta-framework (like Next.js), you often have to tackle this optimization manually, replacing import statements with dynamic imports.
- It might delay when the application begins loading the component impacting the user-experience.

Server Components introduce automatic code-splitting treating all normal imports in Client components as possible code-split points. They also allow developers to select which component to use much earlier (on the server), allowing the client to fetch it earlier in the rendering process.

Will Server Components replace Next.js SSR?

No. They are quite different. Initial adoption of Server Components will actually be experimented with via meta-frameworks such as Next.js as research and experimentation continue.

To summarize a good explanation of the differences between Next.js SSR and Server Components from Dan Abramov:

- **Code for Server Components is never delivered to the client**. In many implementations of SSR using React, component code gets sent to the client via JavaScript bundles anyway. This can delay interactivity.
- **Server components enable access to the back-end from anywhere in the tree**. When using Next.js, you're used to accessing the back-end via `getServerProps()` which has the limitation of only working at the top-level page. Random npm components are unable to do this.

- **Server Components may be refetched while maintaining Client-side state inside of the tree**. This is because the main transport mechanism is much richer than just HTML, allowing the refetching of a server-rendered part (e.g such as a search result list) without blowing away state inside (e.g search input text, focus, text selection)

Some of the early integration work for Server Components will be done via a webpack plugin which:

- Locates all Client components
- Creates a mapping between IDs => chunk URLs
- A Node.js loader replaces imports to Client components with references to this map.
- Some of this work will require deeper integrations (e.g with pieces such as Routing) which is why getting this to work with a framework like Next.js will be valuable.

As Dan notes, one of the goals of this work is to enable meta-frameworks to get much better.

Selective Hydration

How to use combine streaming server-side rendering with a new approach to hydration, selective hydration

In previous articles, we covered how SSR with hydration can improve user experience. React is able to (quickly) generate a tree on the server using the renderToString method that the react-dom/server library provides, which gets sent to the client after the entire tree has been generated. The rendered HTML is non interactive, until the JavaScript bundle has been fetched and loaded, after which React walks down the tree to hydrate and attaches the handlers. However, this approach can lead to some performance issues due to some limitations with the current implementation.

Before the server-rendered HTML tree is able to get sent to the client, all components need to be ready. This means that components that may rely on an external API call or any process that could cause some delays, might end up blocking smaller components from being rendered quickly.

Besides a slower tree generation, another issue is the fact that React only hydrates the tree once. This means that before React is able to hydrate any of the components, it needs to have fetched the JavaScript for all of the components before it's able to hydrate any of them. This means that smaller components (with smaller bundles) have to wait for the larger components's code to be fetched and loaded, until React is able to hydrate anything on your website. During this time, the website remained non-interactive.

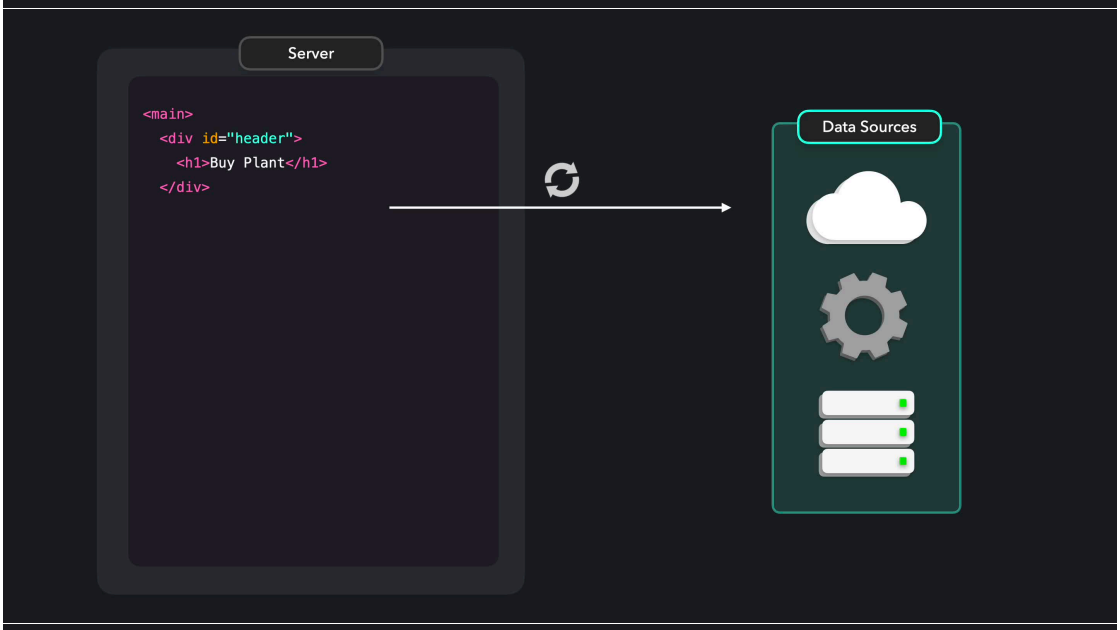

Server
<main>

<h1>Buy Plant</h1>

I love this plant!
The plant was very green.
Looks amazing!
Easy to water
Gorgeous flower!

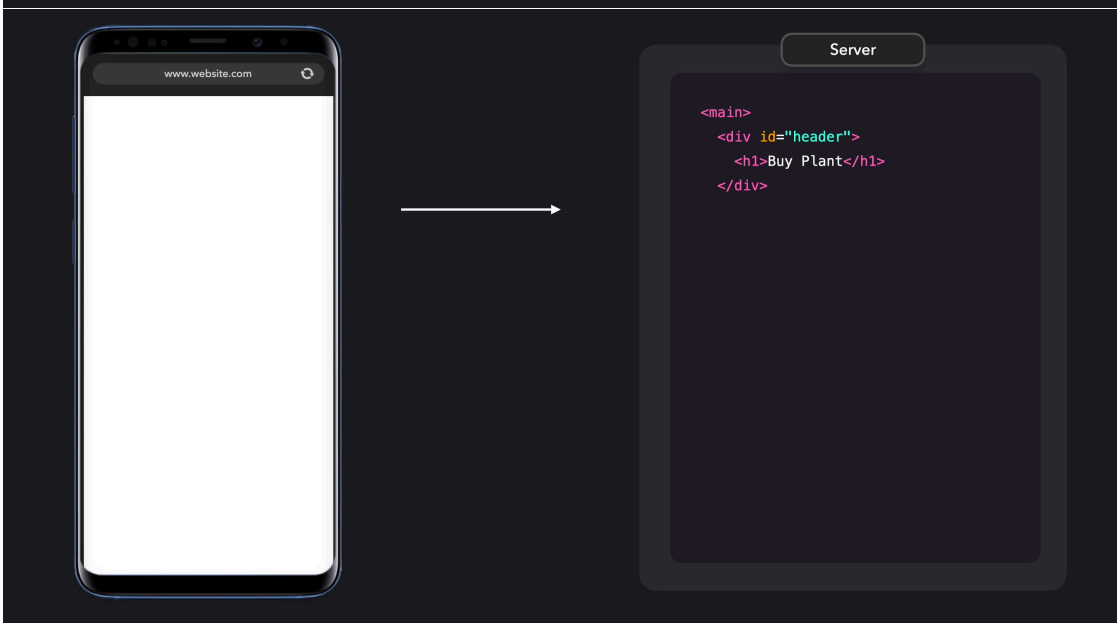
Contact

</main>
Data Sources

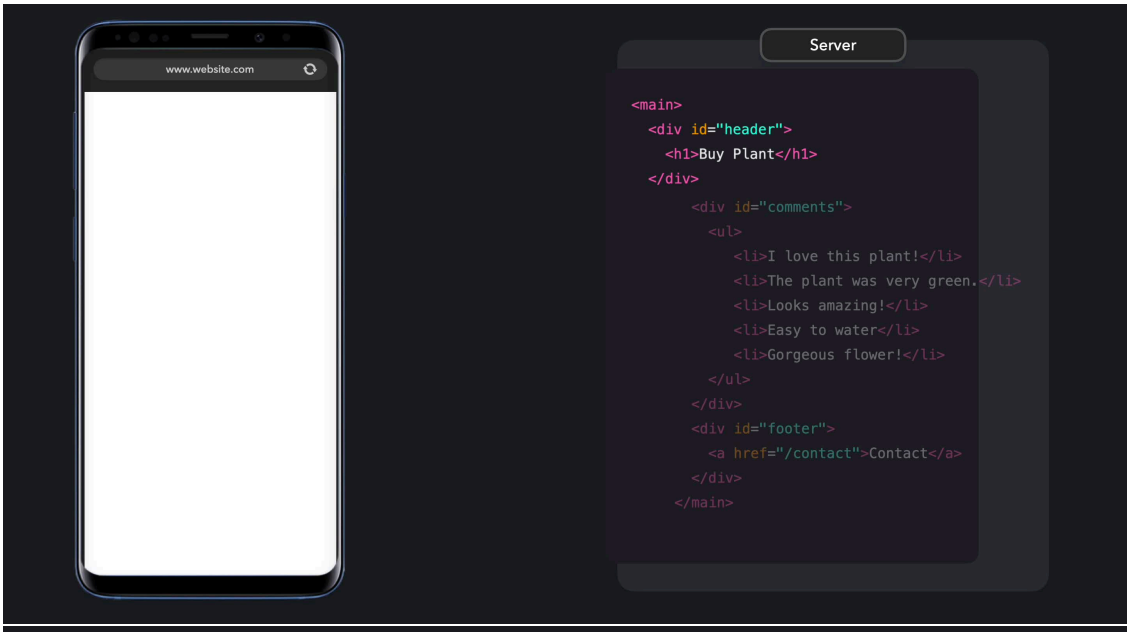

Server
<main>

<h1>Buy Plant</h1>

Data Sources


www.website.com
Server
<main>

<h1>Buy Plant</h1>

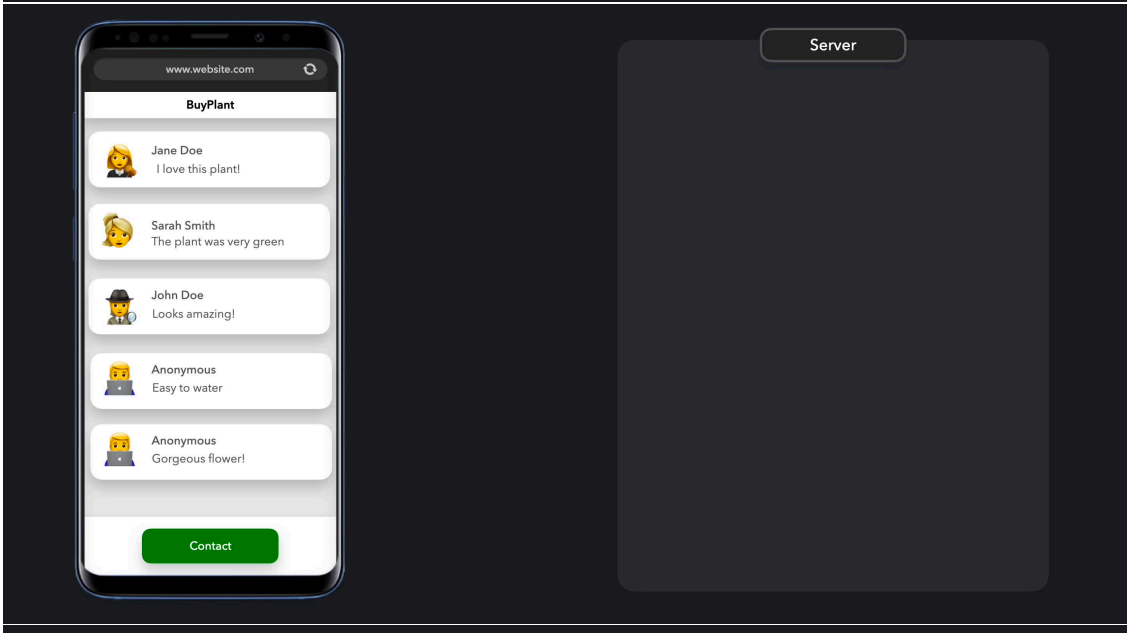

www.website.com
Server
<main>

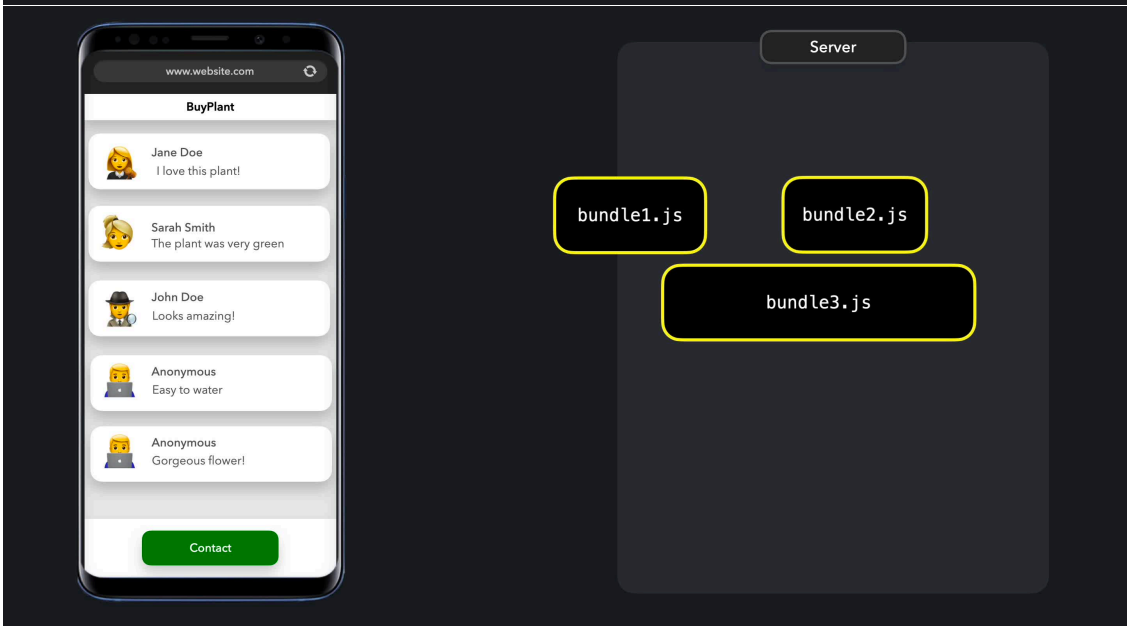
<h1>Buy Plant</h1>

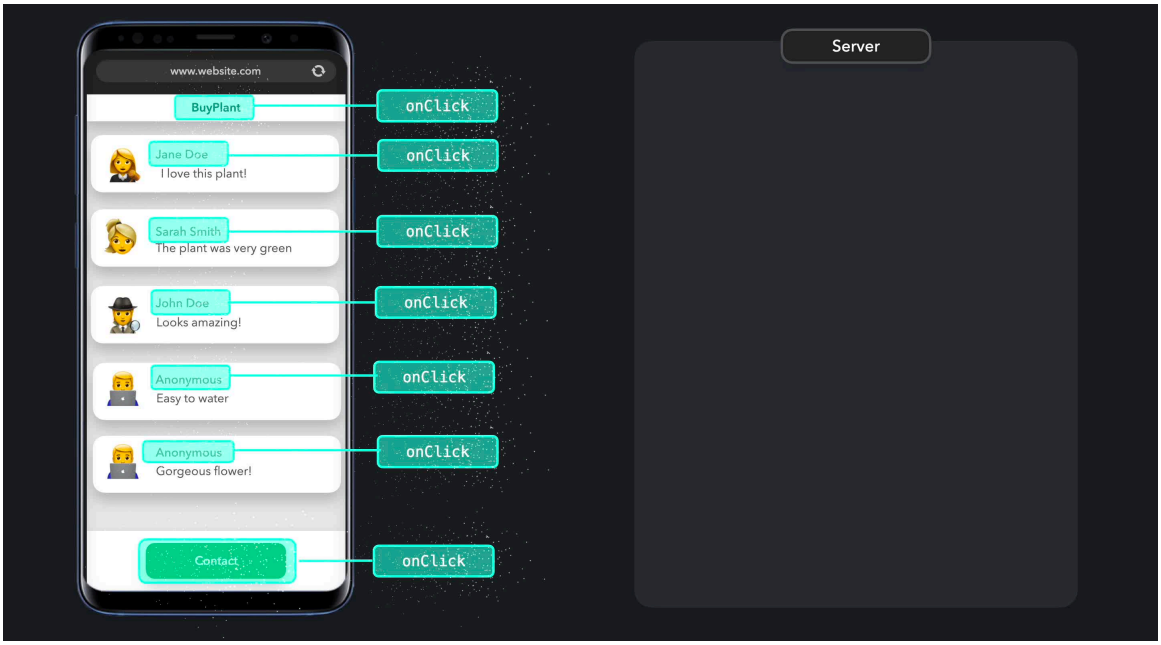
I love this plant!
The plant was very green.
Looks amazing!
Easy to water
Gorgeous flower!

Contact

</main>


www.website.com
BuyPlant
Jane Doe
I love this plant!
Sarah Smith
The plant was very green
John Doe
Looks amazing!
Anonymous
Easy to water
Anonymous
Gorgeous flower!
Contact
Server


www.website.com
BuyPlant
Jane Doe
I love this plant!
Sarah Smith
The plant was very green
John Doe
Looks amazing!
Anonymous
Easy to water
Anonymous
Gorgeous flower!
Contact
Server
bundle1.js
bundle2.js
bundle3.js



React 18 solves these problems by allowing us to combine streaming server-side rendering with a new approach to hydration: Selective Hydration!

Instead of using the renderToString method that we covered earlier, we can now stream render HTML using the new pipeToNodeStream method on the server.

This method, in combination with the createRoot method and Suspense, makes it possible to start streaming HTML without having to wait for the larger components to be ready. This means that we can lazy-load components when using SSR, which wasn't (really) possible before!

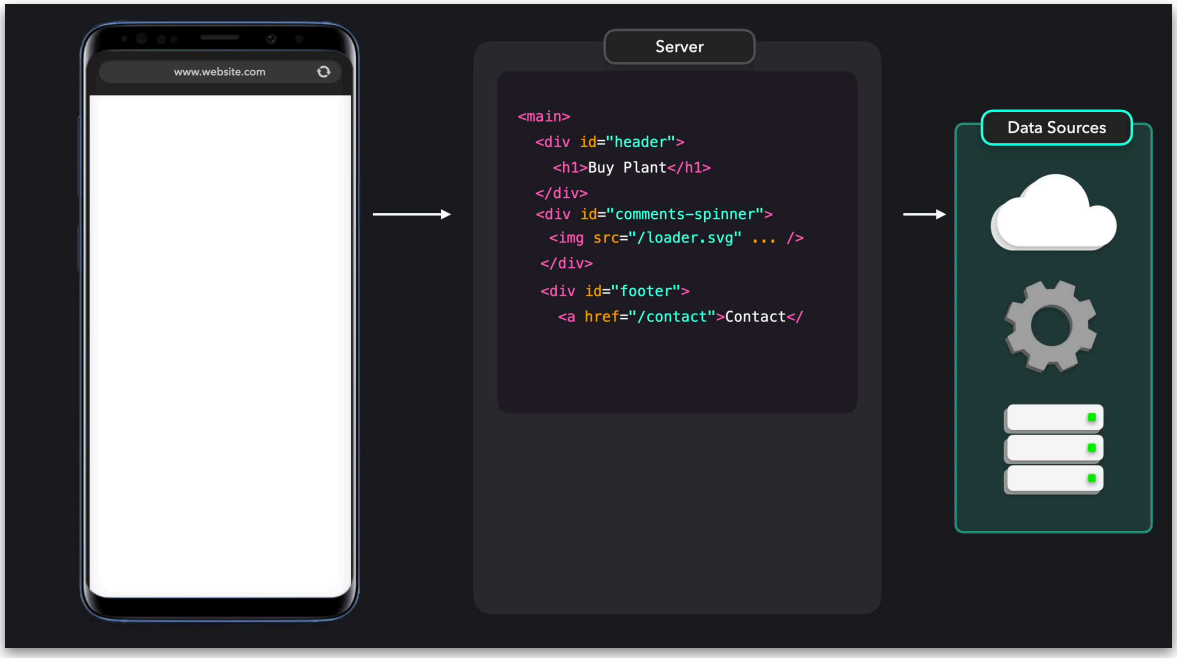
index.js

```
import { hydrateRoot } from "react-dom";
import App from './App';

hydrateRoot(document, <App />);
```

server.js

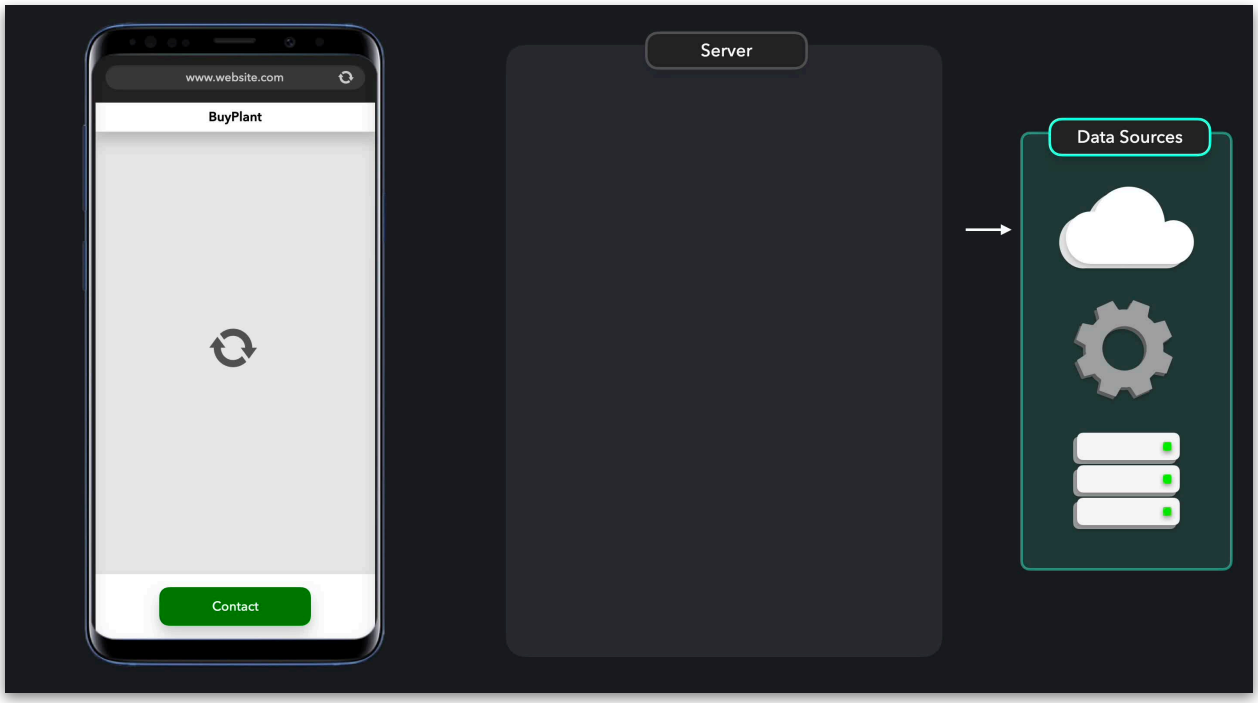
```
import { pipeToNodeStream} from "react-dom/server";

export function render(res) {
  const data = createServerData();
  const { startWriting, abort } = pipeToNodeWritable(
    <DataProvider data={data}>
      <App assets={assets} />
    </DataProvider>,
    res,
    {
      onReadyToStream() {
        res.setHeader('Content-type', 'text/html');
        res.write('<!DOCTYPE html>');
        startWriting();
      }
    }
  );
};
```

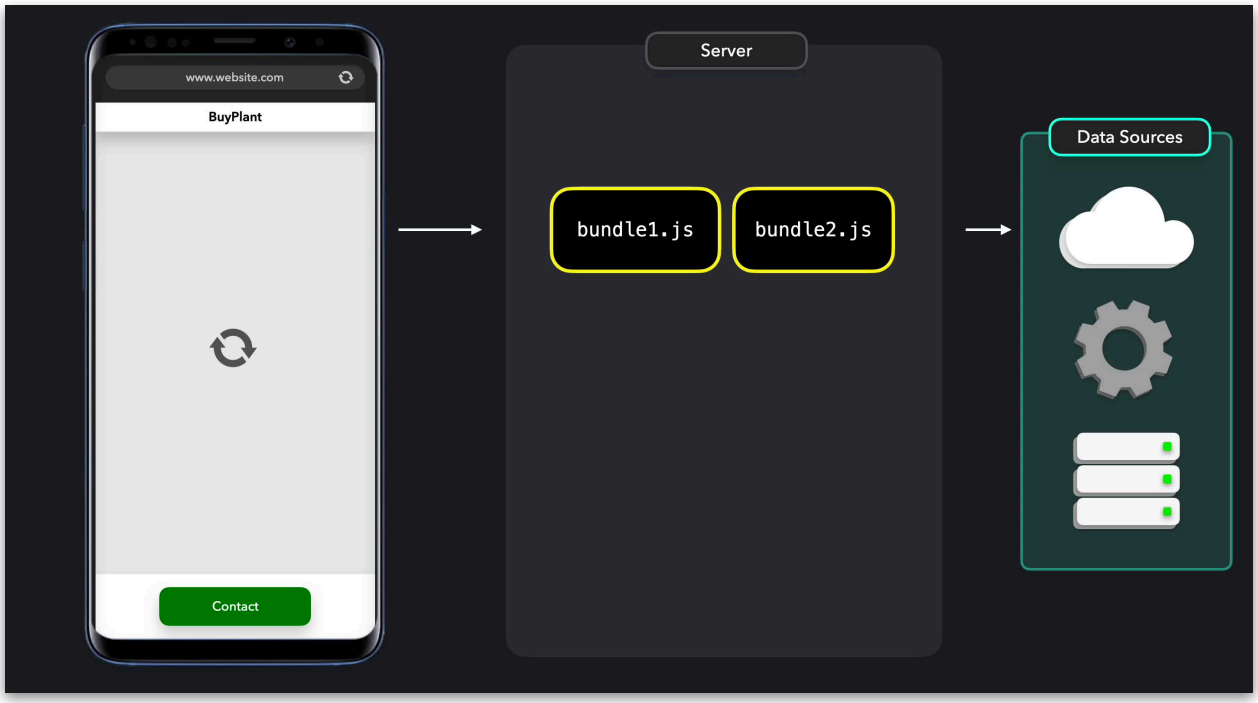
The Comments component, which earlier slowed down the tree generation and TTI, is now wrapped in Suspense. This tells React to not let this component slow down the rest of the tree generation. Instead, React inserts the fallback components as the initially rendered HTML, and continues to generate the rest of the tree before it's sent to the client.

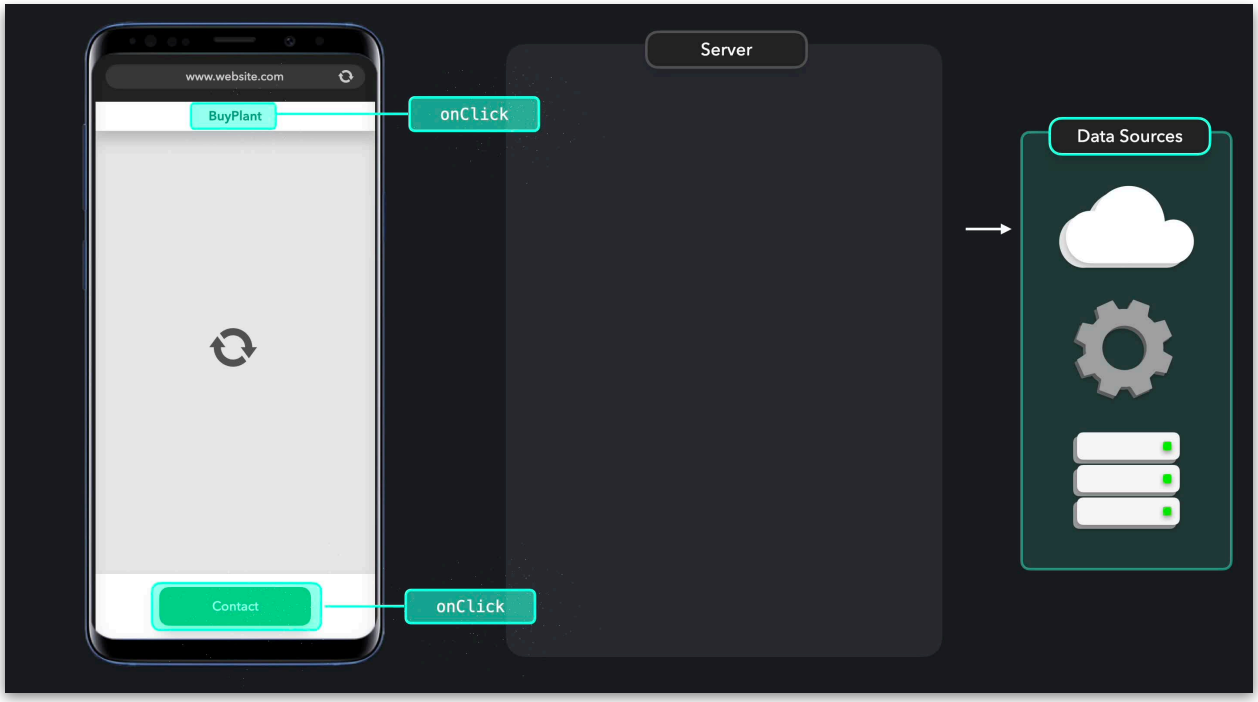


In the meantime, we're still fetching the external data that we need for the Comments component.

Selective hydration makes it possible to already hydrate the components that were sent to the client, even before the Comments component has been sent!

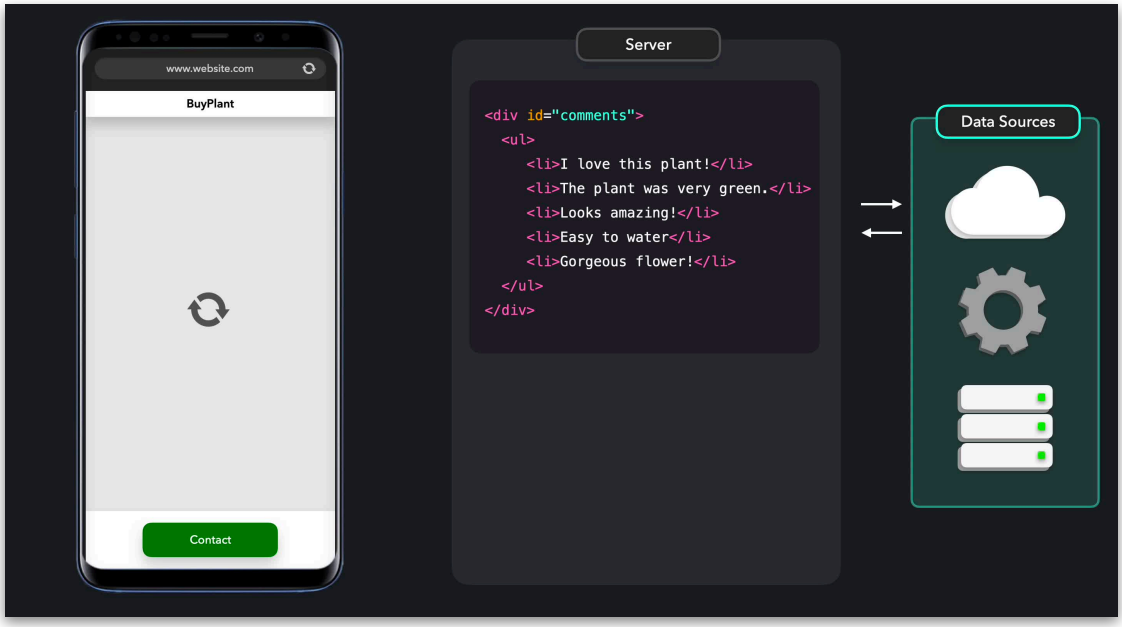


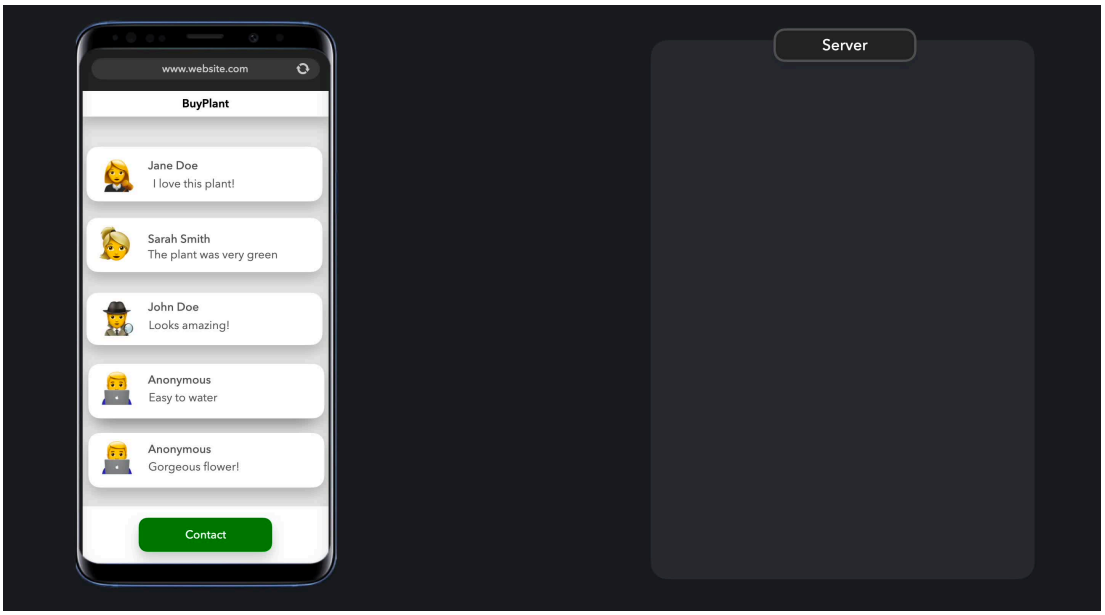


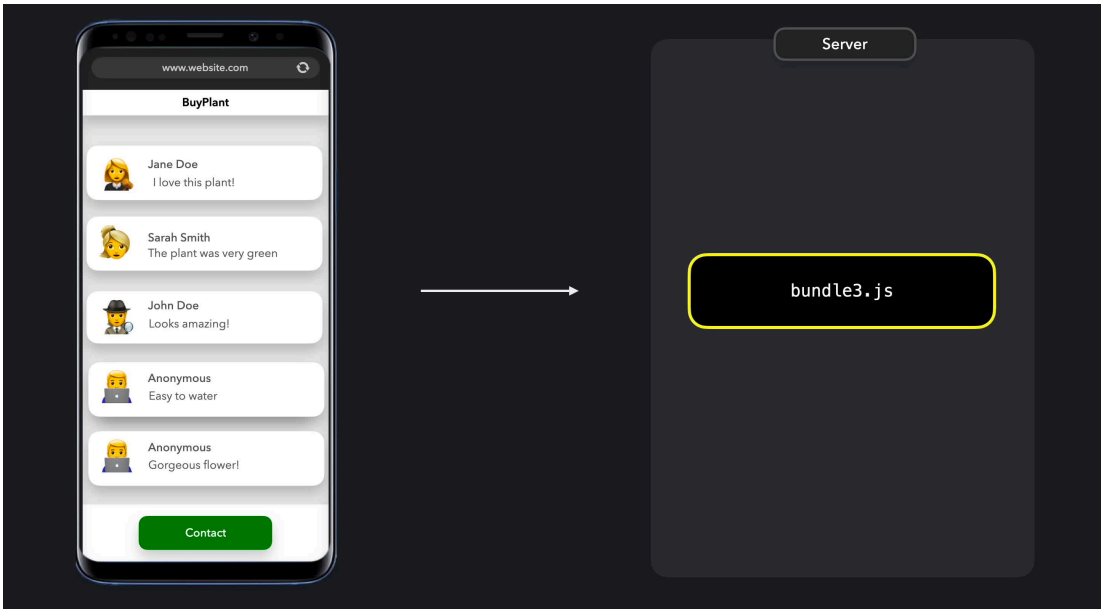


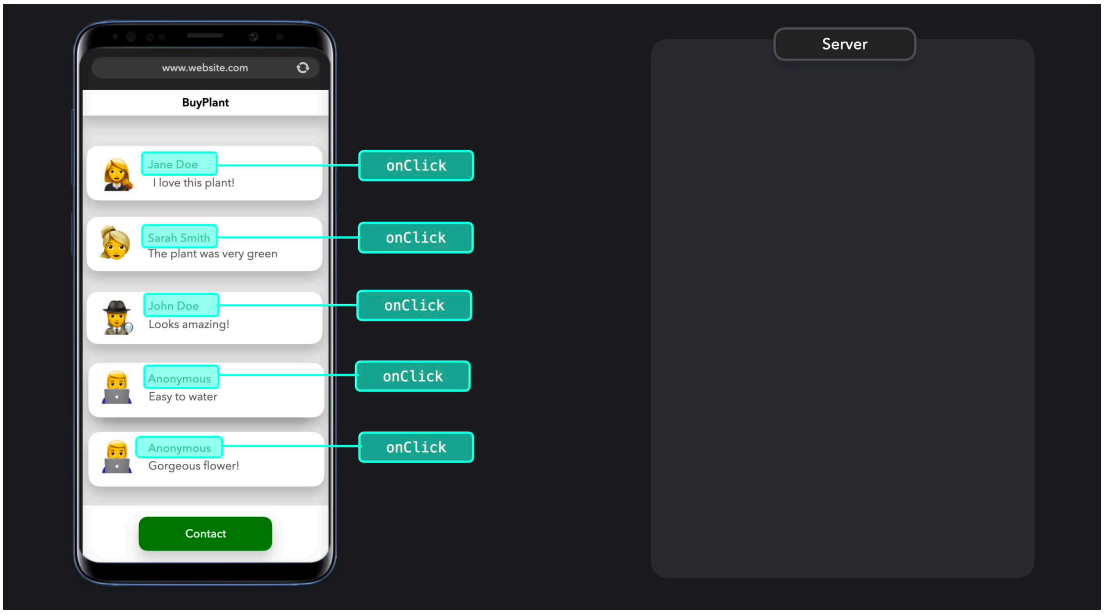
Once the data for the Comments component is ready, React starts streaming the HTML for this component, as well as a small <script> to replace the fallback loader.

React starts the hydration after the new HTML has been injected.




www.website.com
BuyPlant
Jane Doe
I love this plant!
Sarah Smith
The plant was very green
John Doe
Looks amazing!
Anonymous
Easy to water
Anonymous
Gorgeous flower!
Contact
Server


www.website.com
BuyPlant
Jane Doe
I love this plant!
Sarah Smith
The plant was very green
John Doe
Looks amazing!
Anonymous
Easy to water
Anonymous
Gorgeous flower!
Contact
Server
bundle3.js


www.website.com
BuyPlant
Jane Doe
I love this plant!
Sarah Smith
The plant was very green
John Doe
Looks amazing!
Anonymous
Easy to water
Anonymous
Gorgeous flower!
Contact
onClick
onClick
onClick
onClick
onClick
Server

React 18 fixes some issues that people often encountered when using SSR with React.

Streaming rendering allows you to start streaming components as soon as they're ready, without risking a slower FCP and TTI due to components that might take longer to generate on the server.

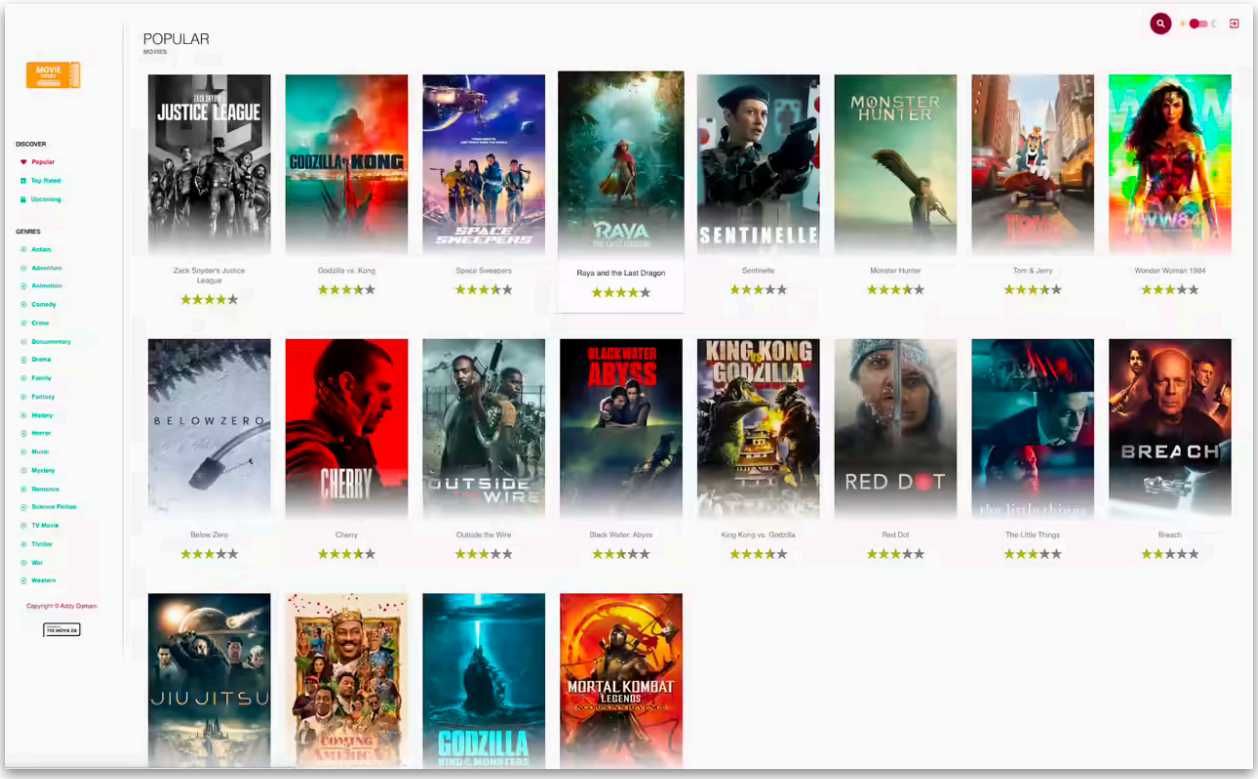
Components can be hydrated as soon as they're streamed to the client, since we no longer have to wait for all JavaScript to load to start hydrating and can start interacting with the app before all components have been hydrated.

Optimizing for the Core Web Vitals on a Next.js app

A case study optimizing a Next.js app for performance

Next.js by Vercel is a React meta-framework that enhances the React development experience. It enables the creation of production-ready apps and supports static site generation and server-side rendering, easy configuration, fast refresh, image optimization, file-system routing, and optimized code-splitting and bundling.

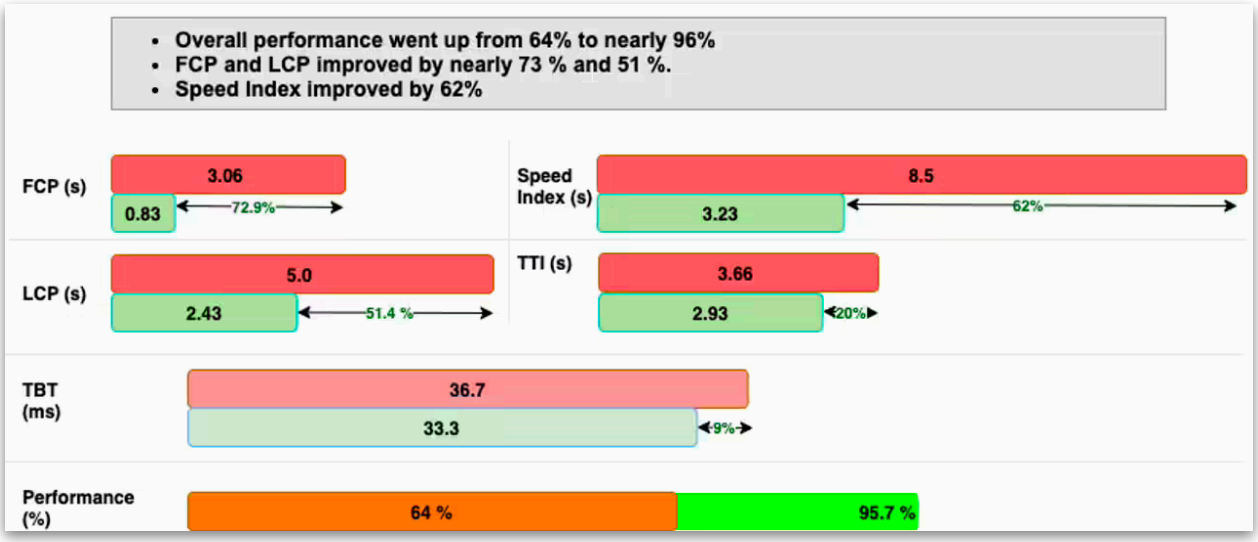
To evaluate how to optimize a React + Next.js application using third-party dependencies, we created the Next.js Movies app. This is a non-trivial movie browsing application and a fully-featured client of TMDB. It incorporates a rich set of features that allow you to search and browse through a comprehensive and categorized movie listing, view details and manage personal favorites through membership and authentication.



Subsequently, the Next.js Movies app was the benchmark that we used to implement a series of performance tweaks and identify the ones that were beneficial from an overall user experience perspective. Today, I want to talk about the performance improvement achieved on the whole and dig into each of the tweaks that we tried with their outcomes.

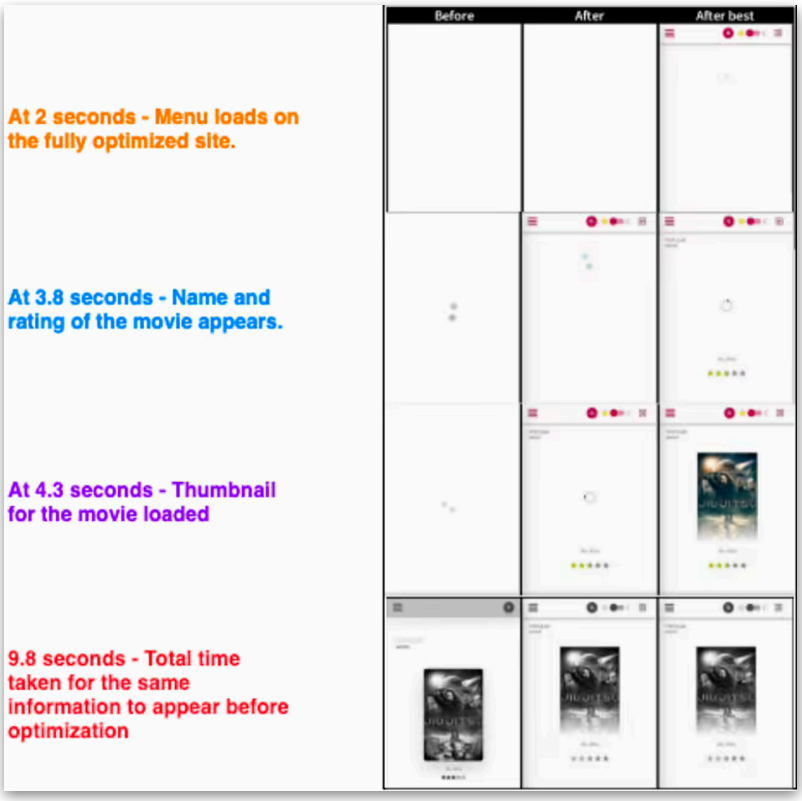
Cumulative Improvements

We were able to achieve a significant aggregate performance improvement with all optimizations in place. This automatically translated to a better user experience. The metrics depicted here were captured before and after the implementation of all the code changes meant to optimize the performance.



To understand what the overall improvement implies from a user experience perspective, take a look at the following comparison for the actual page load experience

- Before optimization
- With a few changes
- With all optimizations in place.



In addition to the overall performance improvement, metrics were captured using Lighthouse and WPT after every code change for relevant pages. The tests were repeated multiple times to eliminate any lags due to sleeping servers or other conditions both before and after the change. The average calculated for each parameter thus gave us a reliable value to use for our analysis.

With that background, let's talk about every change implemented and how it contributed to the overall performance improvement we achieved.

Packages Switched

Initially, a number of third-party React components helped to quickly implement the different features required for the Movies app. We decided to analyze the impact on metrics by trying other available alternatives for individual third-party components especially those that were heavy or blocking the main thread.

Most of these attempts were extremely fruitful in bringing down the values of different metrics

- Using `@svgr/webpack` instead of Font-Awesome for SVG icons helped to boost Speed Index by 34%, LCP by 23%, and TBT by 51%
- Using a custom-built component to replace react-burger-menu and removing the `resize-observer-polyfill` from `react-sicky-box` led to a reduction in bundle size by 34.28 kB (gZipped).

- React Select Search was used instead of React Select which led to a 35% improvement in the LCP with a 100% improvement in CLS and 18% in TBT.
- The use of React Glider instead of React Slick improved TBT by 77%.
- Usage of React Scrolling instead of native smooth scrolling provided cross-browser compatibility for the scrolling feature.
- React Stars component was used instead of React Rating which helped to boost TBT by 33%.

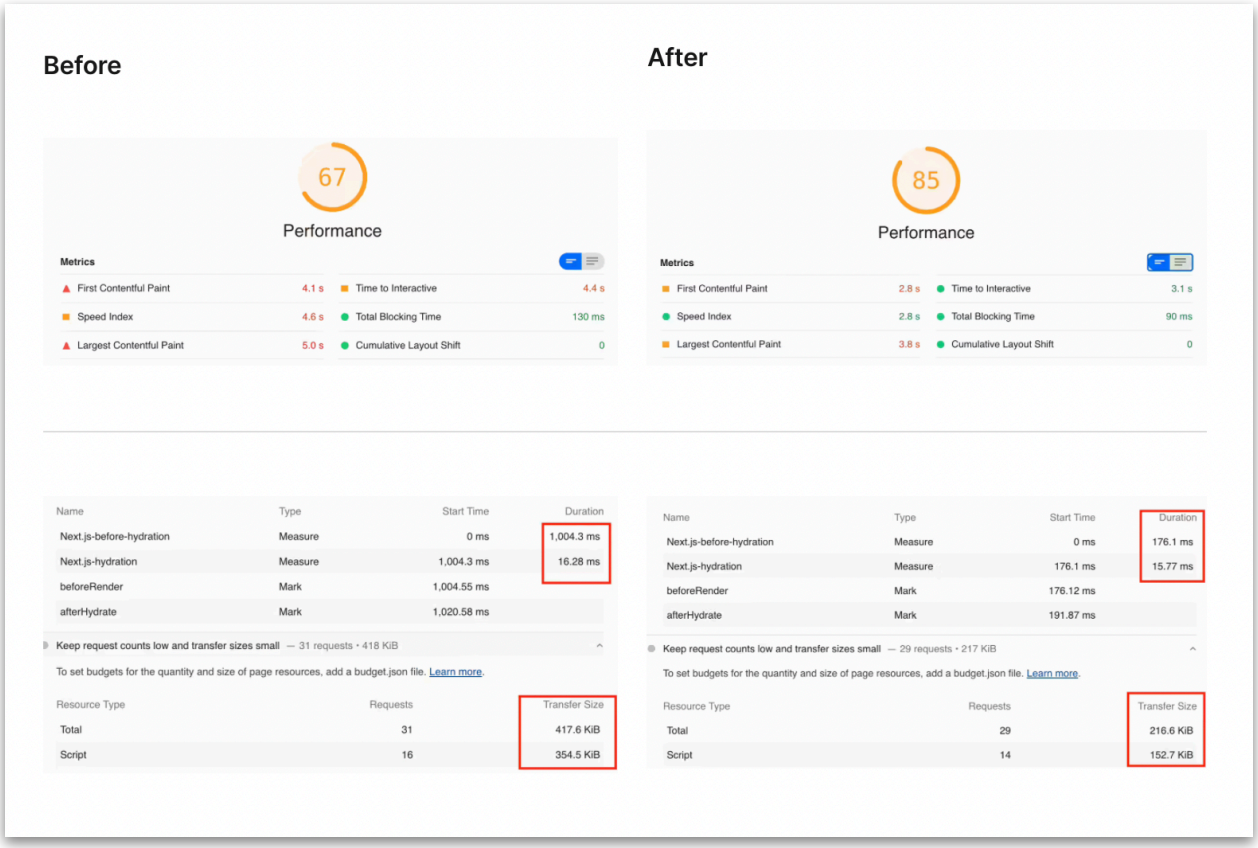
SVG icon library

SVG icons were the obvious choice for all our icon needs across the Movies app. We initially chose Font-Awesome due to its popularity and ease of use as a scalable vector icon library with icons that are customizable using CSS. However, there had been concerns that Font-Awesome may be slow to load on web pages due to the large transfer sizes when loading the library. This affects Lighthouse performance score.

We replaced Font-Awesome with `@svgr/webpack` as our SVG icon provider. Another change was to import individual icons on all our pages instead of the library itself even if the page uses multiple icons. For example:

```
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';

// replaced by
import HeartIcon from 'public/assets/svgs/icons/heart.svg';
import PollIcon from 'public/assets/svgs/icons/poll.svg';
import CalendarIcon from 'public/assets/svgs/icons/calendar.svg';
import DotCircleIcon from 'public/assets/svgs/icons/dot-circle.svg';
```

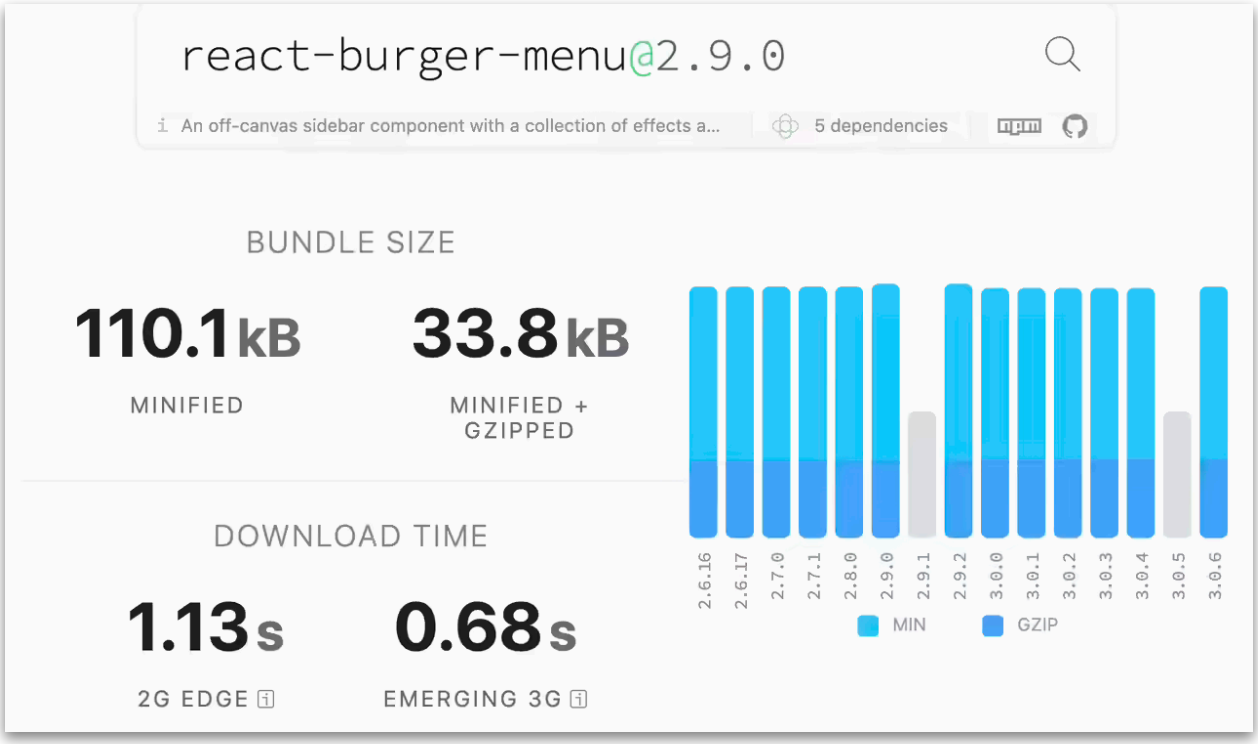
This helped to improve the Lighthouse score across the board. Here is a snapshot of the score before and after the change. Also, note a difference of almost 200 KiB in request transfer size and the change in user timings before and after the change.

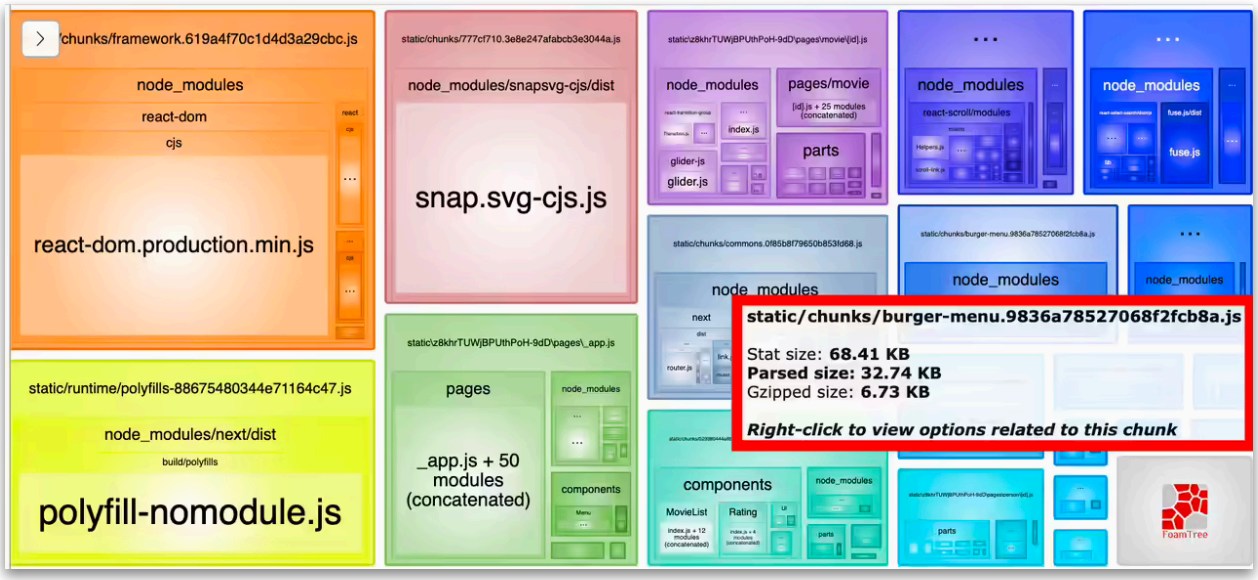


Application Menu

The initial version of the app used `react-burger-menu` as an off-canvas side-bar component to display the application menu by clicking the burger icon. The component comes with a collection of inbuilt CSS styles and animations that provide options to customize the menu.

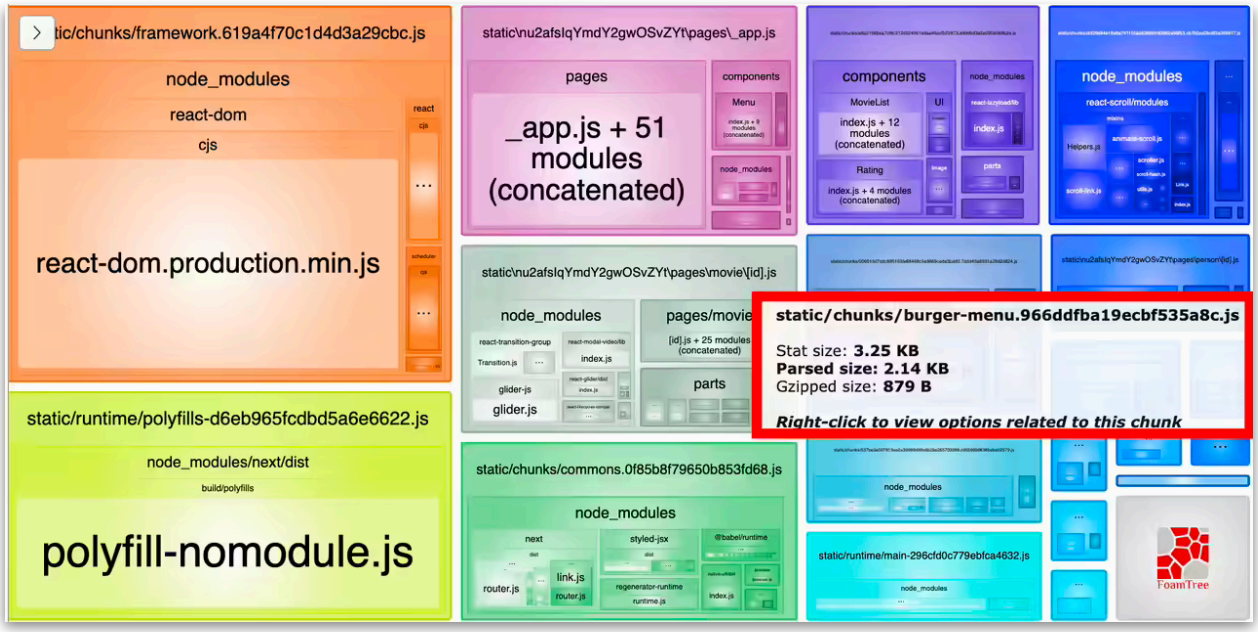
An analysis of bundle sizes for `react-burger-menu` and the app revealed that we could do better.





We did not need all the features included in the react-burger-menu component and thought that a simple custom component would serve our needs just as well.

This helped to reduce the bundle size corresponding to the burger menu component considerably without affecting the required functionality. As seen in the treemap analysis of the chunks before and after the change, the gzipped size of the burger-menu chunk was 6.73 kB earlier but reduced to 879 B after the change. The parsed size also went down from 32.74 kB to 2.14 kB. Thus, the change helped to reduce both the download time as well as the parse time for the chunk



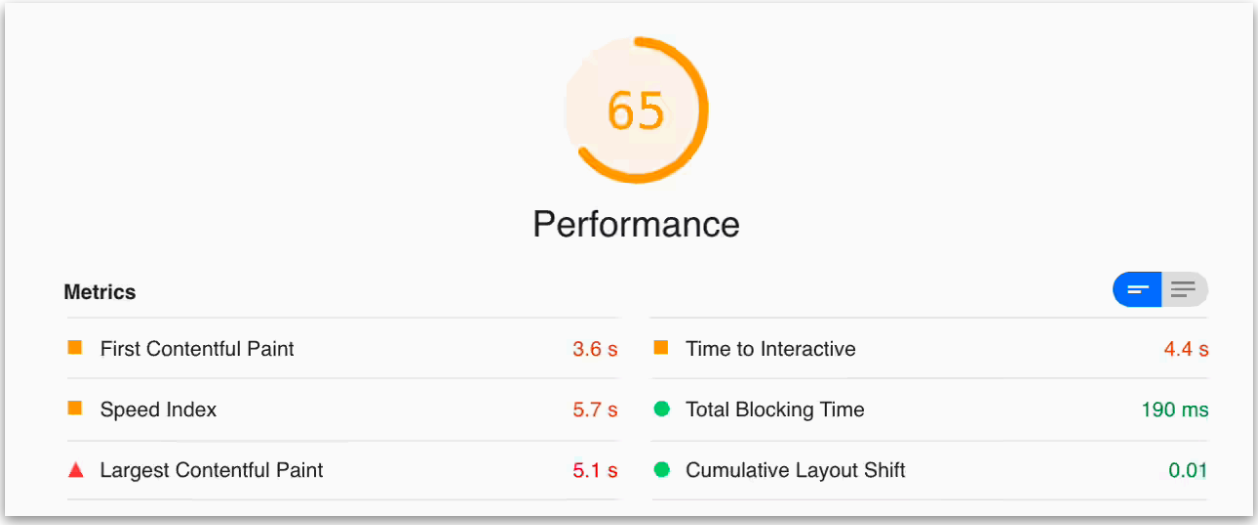
Dropdown for Sort feature

The Movies app allows you to sort movies belonging to a particular genre or starring a selected actor. You can sort by Popularity, Votes, Release Date, or Original Title. To allow users to select a sort option, we had previously used the react-select component. The component allows for multiple-select, search, animation, and access to styling API using emotion. The bundle size for the component is 27.2 kB minified and gzipped with 7 dependencies.

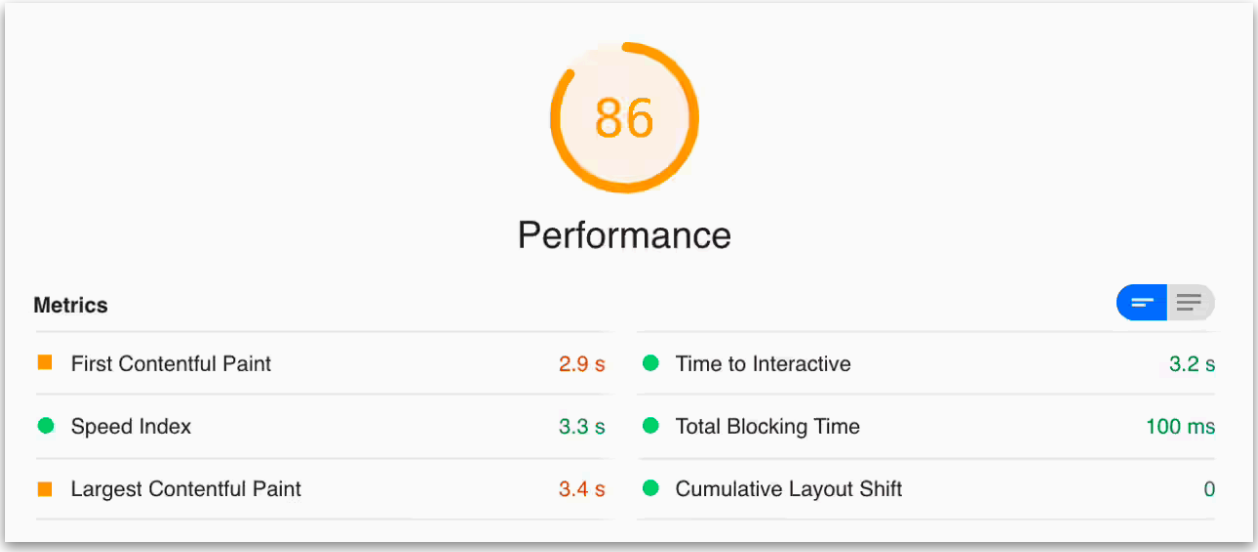
For the sort dropdown, we merely needed a simple single-select component without any styling features. As such, we decided to go with the react-select-search component. It is a lightweight component (3.2 kB minified and gzipped) with zero dependencies. While it supports multi-select and search features, styling features can be included by developers as required.

The following highlights the changes in the UI itself due to the component change and corresponding improvement in Lighthouse performance.

Before

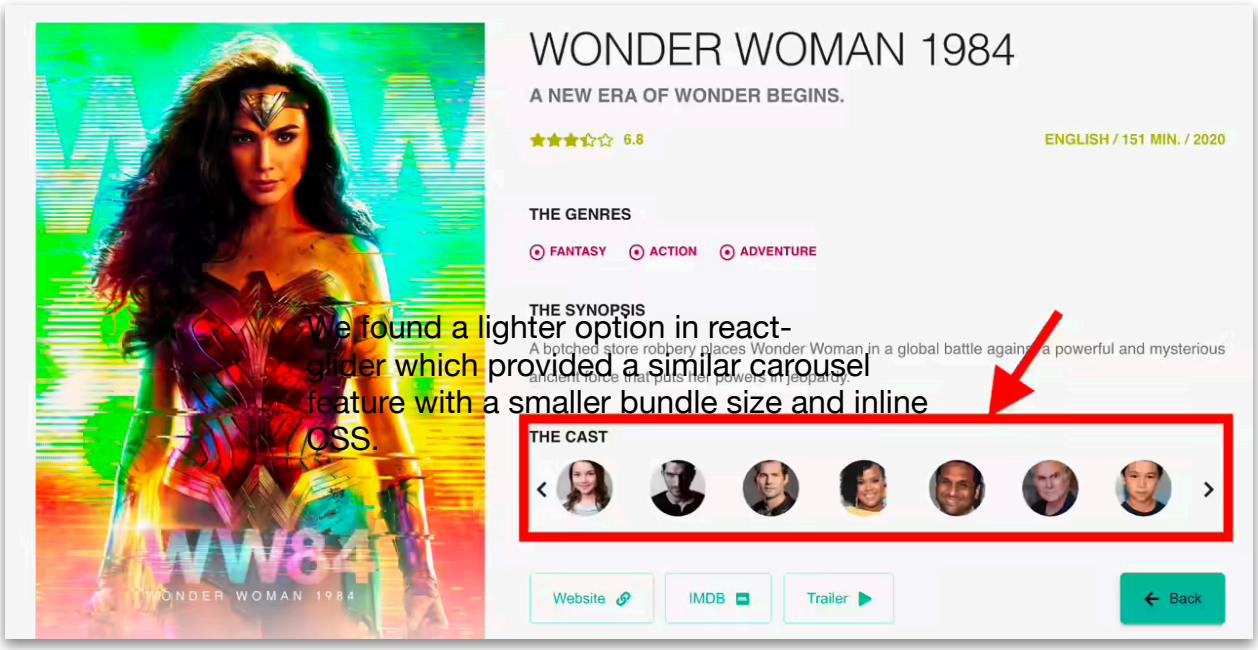


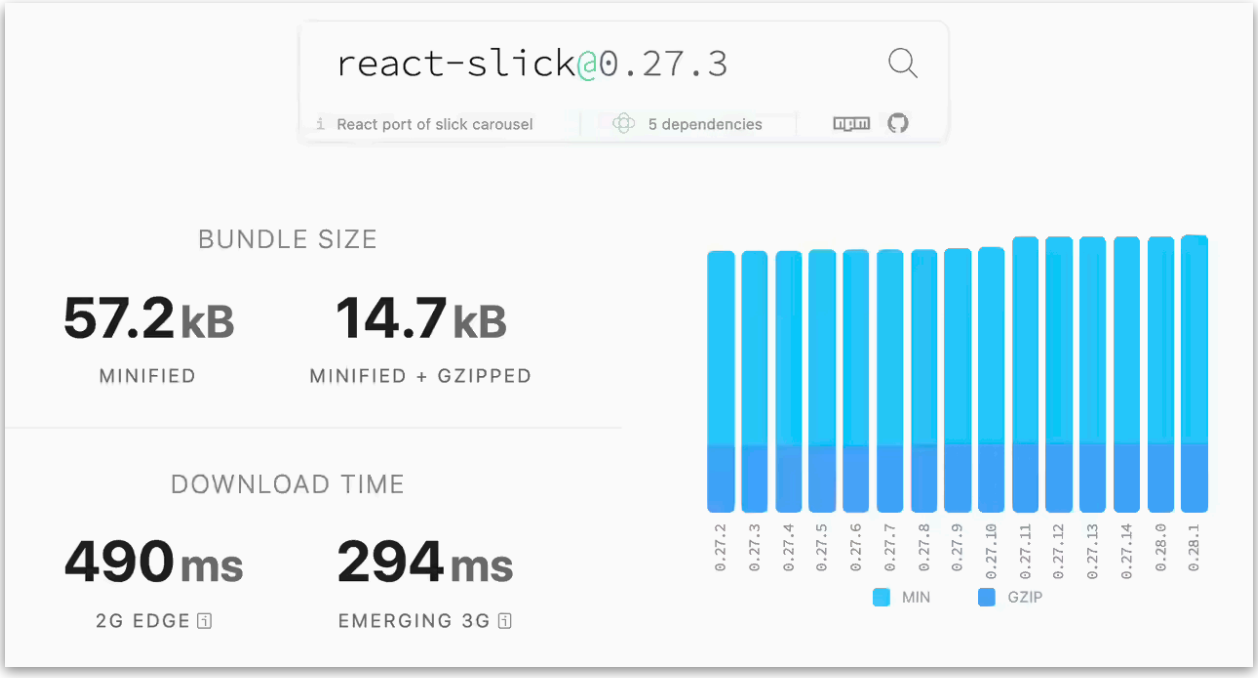
After



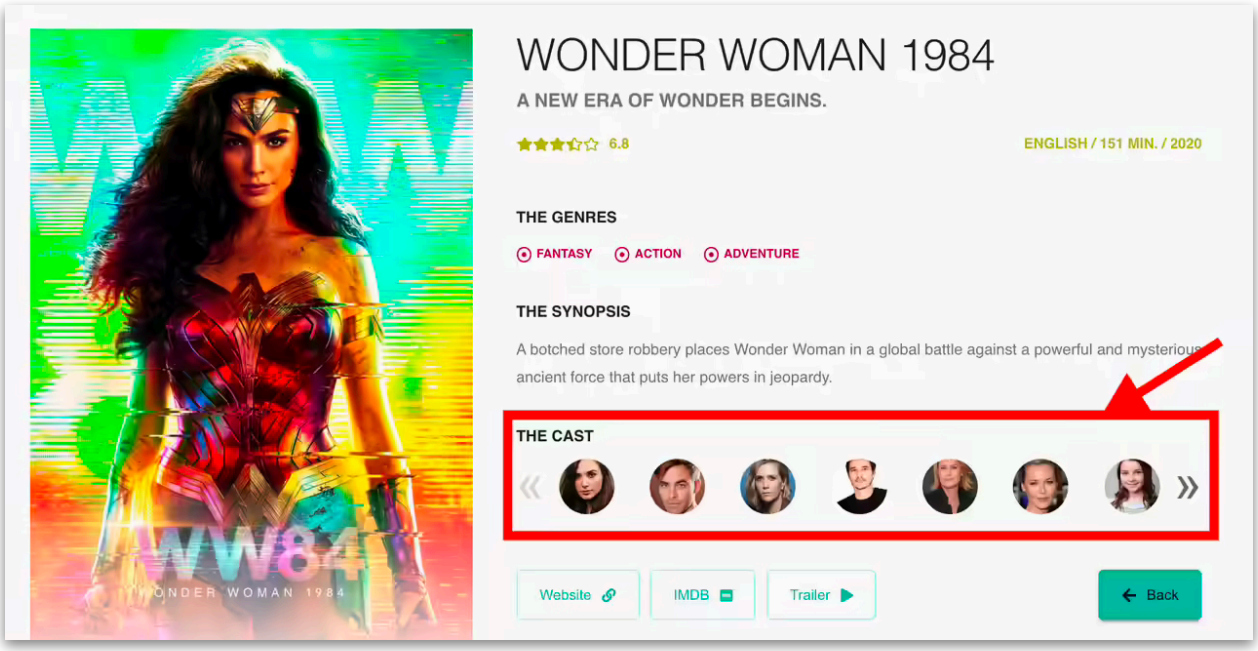
Cart Carousel

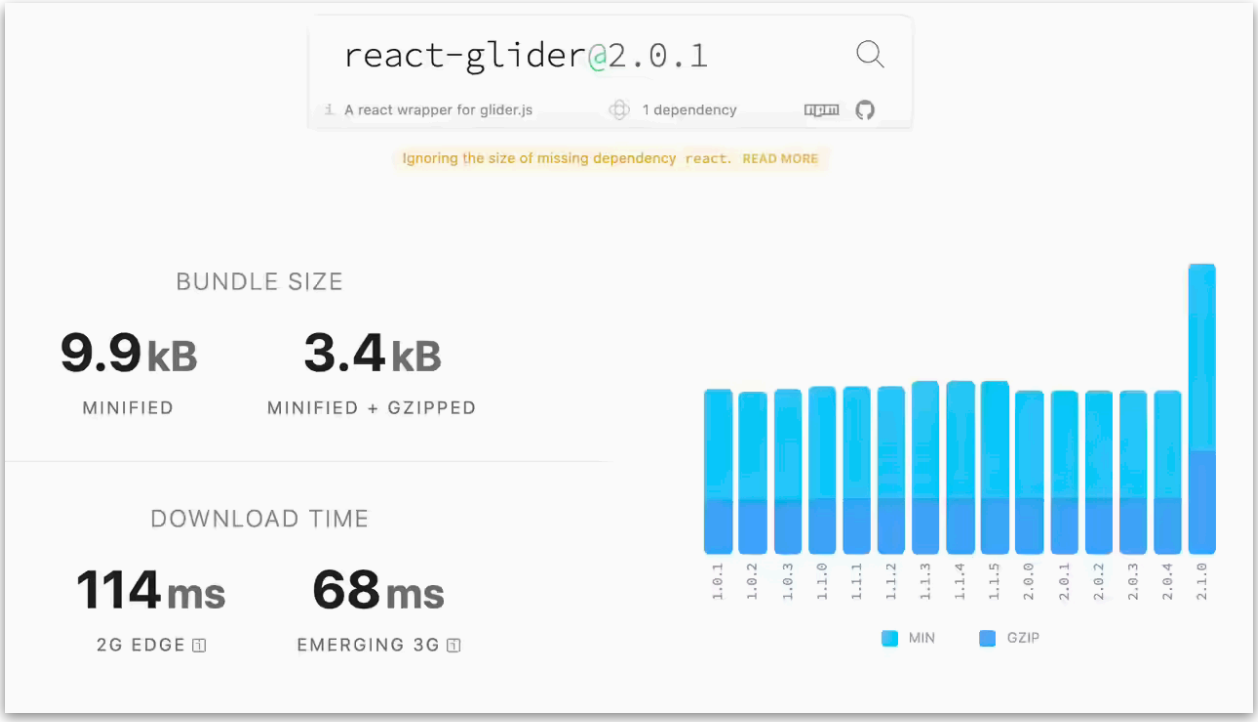
We had used the `react-slick` component on our movie pages that allowed users to horizontally "glide" through the movie cast. The react-slick component however is quite heavy when it comes to the bundle size. At 14.7 kB it comes with 5 dependencies.





We found a lighter option in r `react-glider` which provided a similar carousel feature with a smaller bundle size and inline CSS.





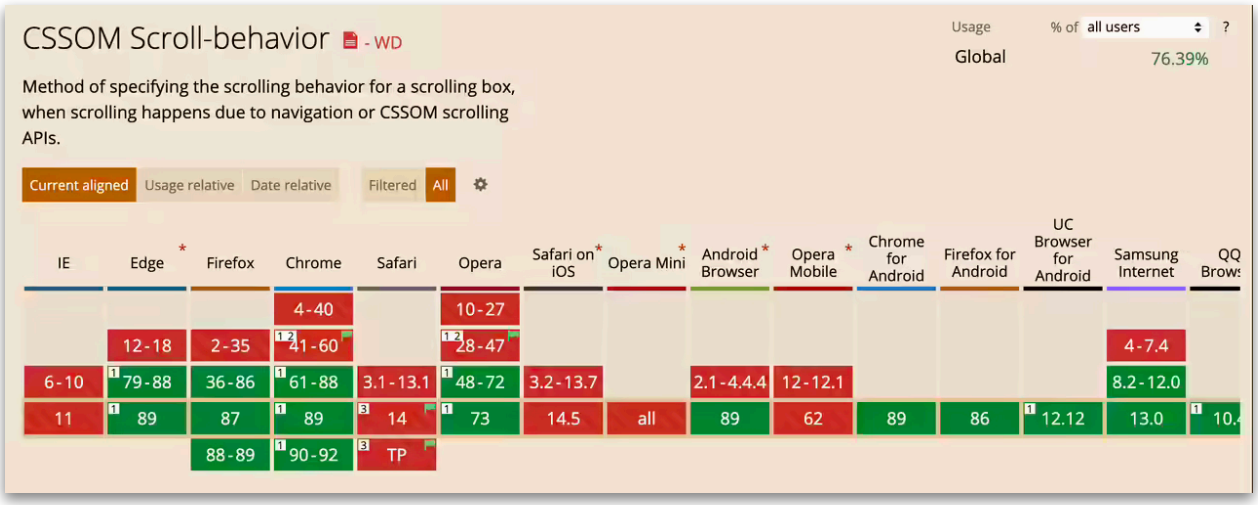
A reduction in bundle size from 14.7 kB to 3.4 kB was quite a jump (78% improvement) with zero impact on functionality. This change was a welcome addition. In the future, we may rewrite this component to use CSS Scroll Snap.

The scrolling component

The Movies App implements pagination on the movie listing pages to switch from one page to the other. Every time the previous or next page button is clicked, the view needs to scroll to the top of the new page. For the transition to be smooth, we had used the native smooth scroll function as follows.

```
window.scroll({
  top: 0,
  left: 0,
  behavior: 'smooth'
});

document.querySelector(`.${SCROLL_TO_ELEMENT}`)
  ?.scrollIntoView({  behavior: 'smooth' });
```

Native smooth scroll functions are however not supported across all browsers.

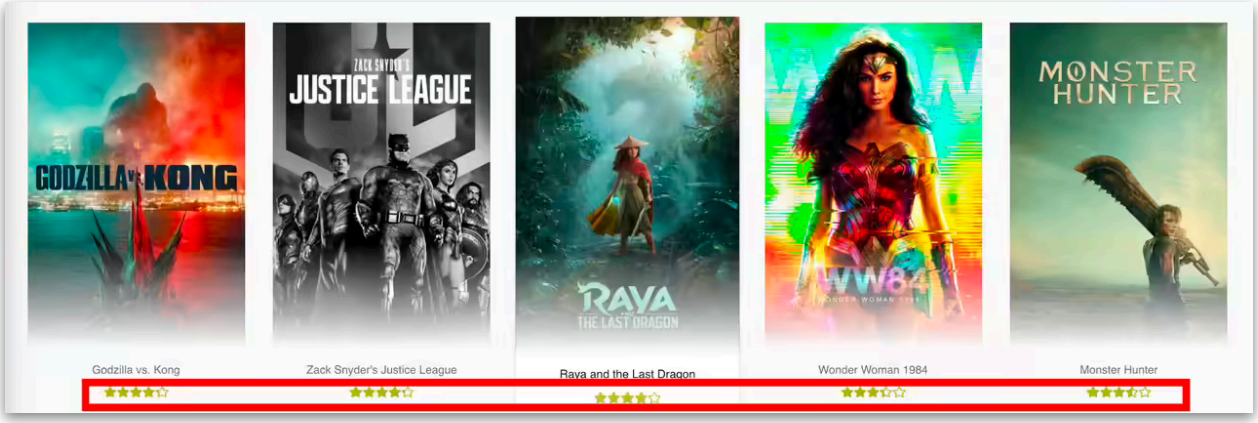


To allow us to animate the vertical scrolling, we decided to use a scrolling library called `react-scroll` (6.8 kB gzipped). This not only helped to recreate the same scroll effect with a small regression in performance as can be seen in the following comparison.

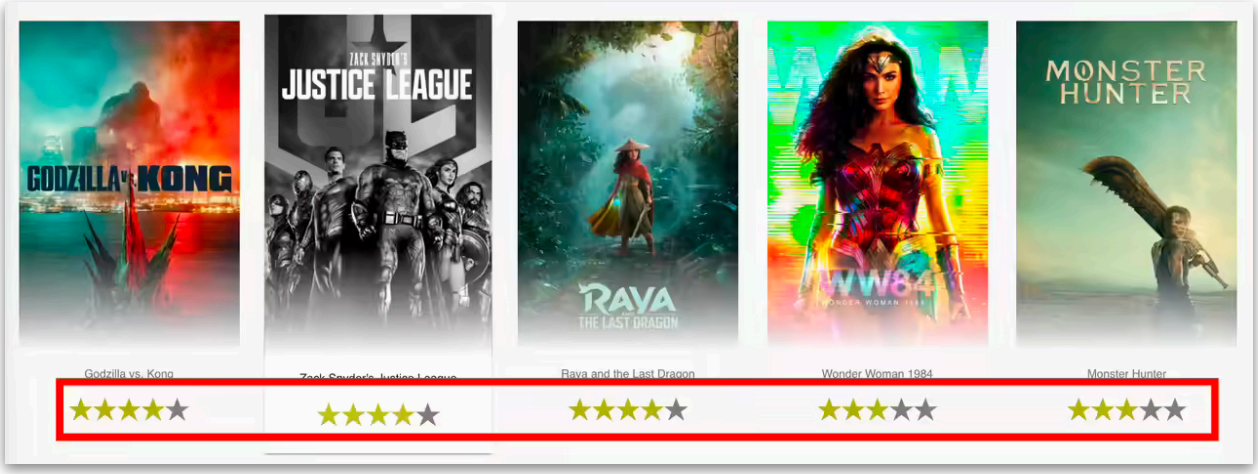
Performance Metric	FCP (s)	Speed Index (s)	LCP (s)	TTI (s)	TBT (ms)	CLS	Performance (%)
Before	0.8	3.93	2.63	1.73	16.66	0	94.33
After	0.92	3.78	2.9	2.26	66	0	92.8
% Change	15	3.81	10.26	39.63	296.15	0	

The rating component

`react-rating`, the rating component that we had originally used, allows you to customize the look by using different symbols for rating; eg., stars, circles, thumbs-up, etc. We had used the star symbol for rating earlier and did not need the other features that were part of the library. The cost of including the bundle for this component was 2.6kB.

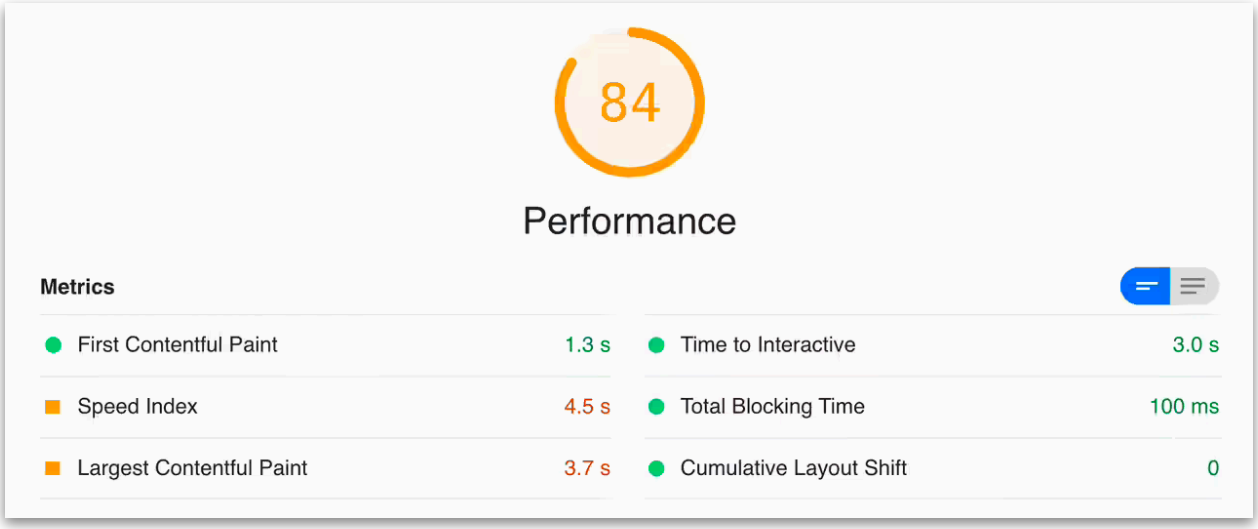


The `react-stars` component served our purpose and we were able to show star ratings for movies on the movie listing screen using this component too. This component was only 2 kB minified and gzipped. We used this component and inlined the source for further optimization.

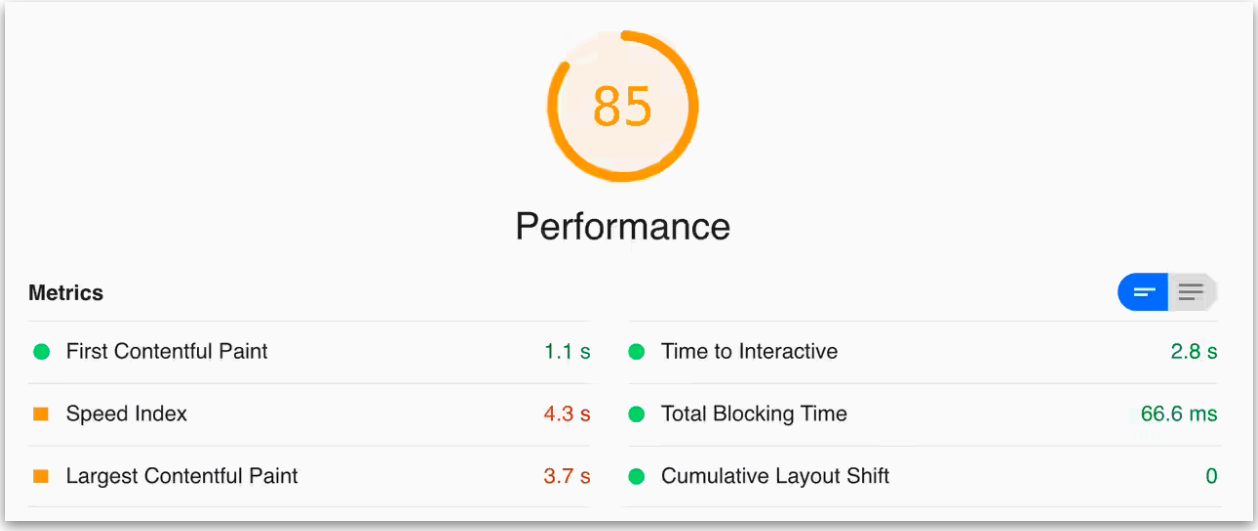


Although, the library sizes do not look very different, the react-rating component uses SVG icons for ratings while the react-stars component uses the symbol "★". As the component gets repeated 20 times on the movie listing page, the size of the icon/image also contributes to the overall savings due to the component change. This is apparent from the Lighthouse scores before and after the change.

Before



After



Although the other parameters are more or less unchanged, we noticed a significant difference in TBT (33%). This was because the chunk that included the rating component (react-rating package) was excluded from the long main-thread tasks.

Other techniques used for Optimization

Experimenting with alternate libraries was one part of the performance analysis and optimization project. We also tried other mechanisms that have been known to enhance performance. Let's talk about what was attempted and what worked or didn't work for us.

Code-Splitting

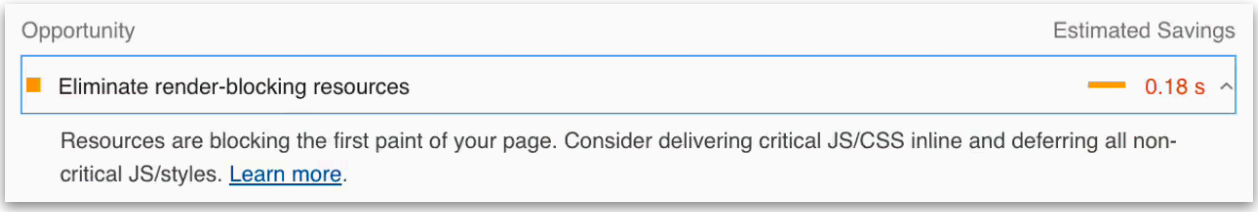
We used code-splitting to lazy-load the Menu component - being collapsed by default on mobile, this was an opportunity to only do work when a user actually needed it. We had initially tried lazy loading with the Burger Menu sidebar component and observed some gain in performance. After we replaced this with a custom component for the sidebar menu, we lazy-loaded the custom component.

We used the `LazyLoadingErrorBoundary` component which acts as a wrapper for react lazy and react suspense. This ensures that the menu component is loaded after page load. While FCP and LCP remained about the same, there was a substantial reduction in TBT by 71% as can be seen in the following comparison.

Performance Metric	FCP (s)	Speed Index (s)	LCP (s)	TTI (s)	TBT (ms)	CLS	Performance (%)
Before	0.86	4.2	3.46	2.53	70	0	87.66
After	0.83	3.63	3.3	1.73	20	0	**90.33**
% Change	**3.48**	**13.57**	**4.62**	**31.62**	**71.42**	0	

Inline the critical, defer the non-critical

Our Lighthouse audits were consistently generating this suggestion that we certainly needed to act upon.



CSS is a render-blocking resource, i.e., it must be loaded and processed before the page is rendered. Some of the CSS may be required to style the content that is visible on the initial page load. This is the critical CSS that needs to be inlined to optimize the page. There may be other CSS that is not required initially and can be deferred.

As part of our optimizations, we in-lined the CSS required for dark/light modes transition which was identified as critical CSS.

As per Next.js documentation, we had initially imported all our node module CSS files in the `/pages/_app.js` file. We are using two components react-glider and `react-modal-video` that require CSS import from node modules. Importing this CSS through `_app.js` would be render-blocking for the app as these components are not required on all the pages.

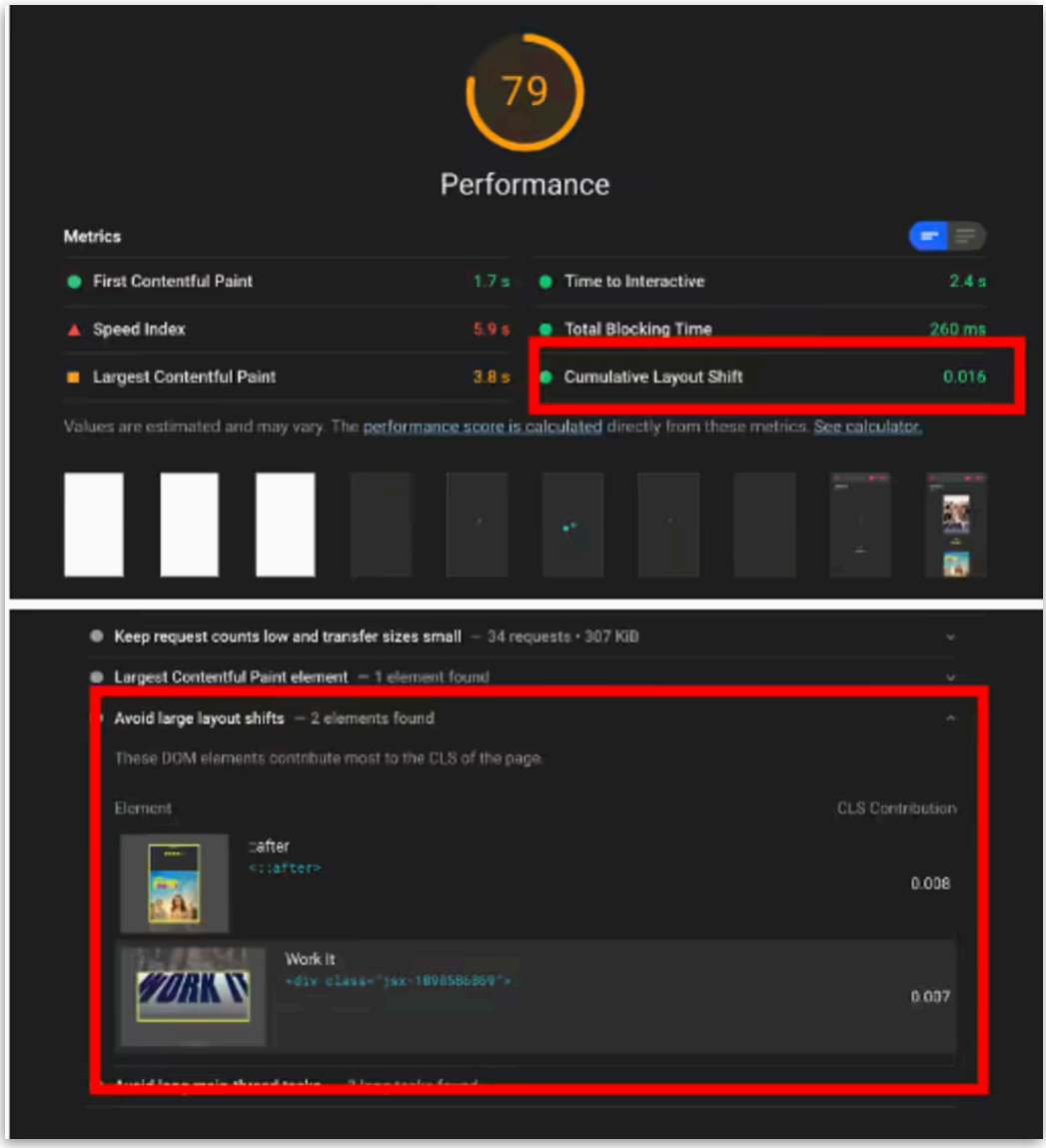
The CSS required by these components was inlined in the files where the component was used. For example, after optimization, the code in our cast component includes the syntax to render the Glider along with the styles that it uses.

```
<div ref={ref} className='cast'>
  <Glider
    hasArrows
    slidesToShow={slidesToShow}
    slidesToScroll={1}
    itemWidth={GLIDER_ITEM_WIDTH}
  >
    {cast.map(person => (
      <PersonLink
        key={person.id}
        person={person}
        baseUrl={baseUrl}
      />
    ))}
  </Glider>
</div>
<style jsx>{`/*CSS Classes required for Glider*/`}</style>
```

With this change, we were able to observe a slight change of 2% to 5% in FCP, LCP, and TTI. The performance improved from 79% to 81% for the page.

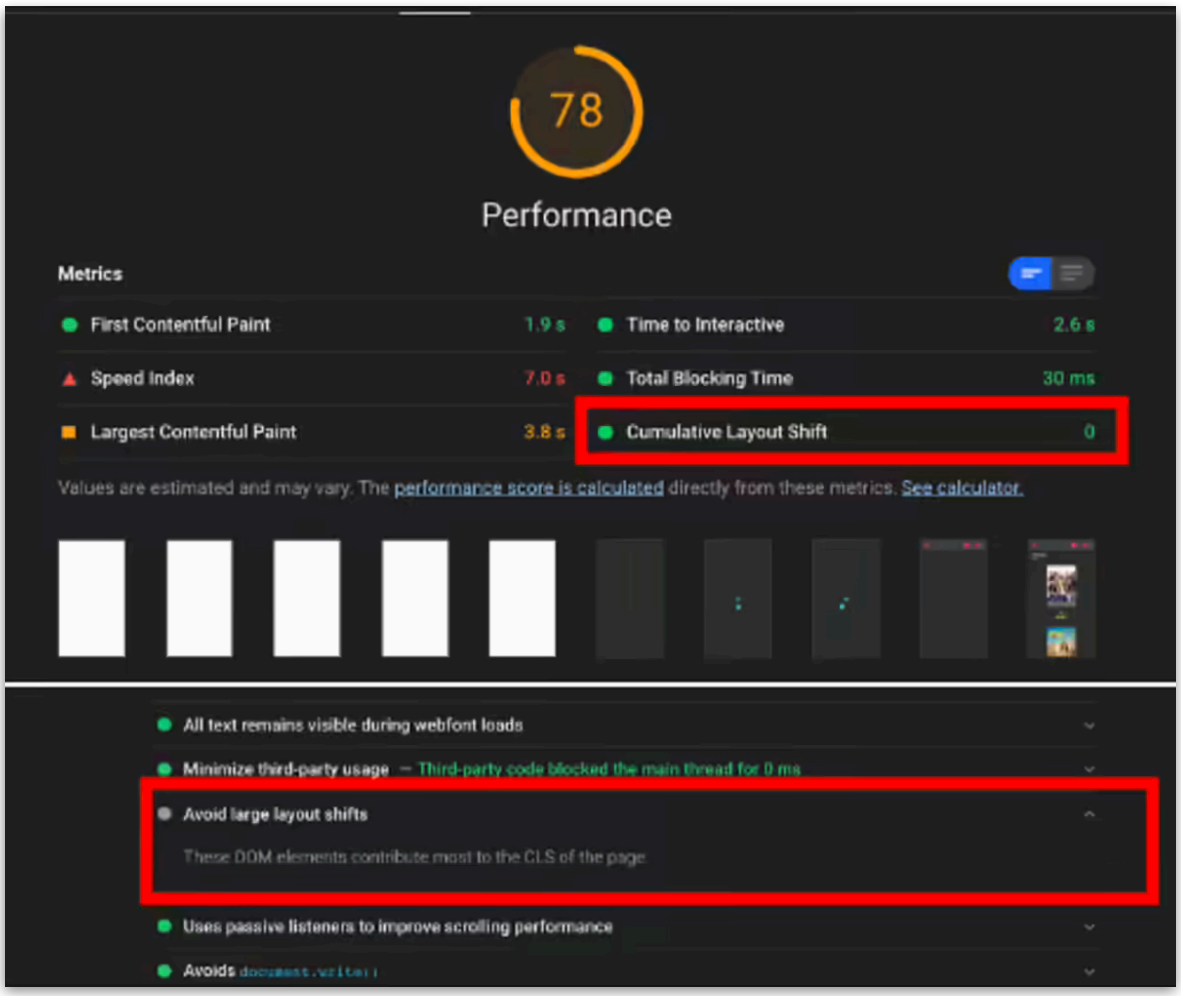
Aspect Ratio for Images

The changes we discussed so far helped us to improve the FCP, LCP, TBT, and TTI on different pages. Let us now talk about improving the last parameter on the Lighthouse report, the Cumulative Layout Shift (CLS). For an in-depth understanding of CLS and its causes, refer to my article on optimizing CLS. The Lighthouse report for the movies page before optimization gave us a clear indication of what was causing the CLS.



Even though a CLS of 0.016 is well below the threshold, we did experience the shift when loading the page, especially on a mobile 3G connection. So we worked on the elements that were causing the layout shift as reported. Instead of setting image dimensions, we used the aspect-ratio-boxes technique for setting the aspect ratio for images. This helps to reserve the required space for the image while the page is still loading so that there is no shift once the image is loaded. Using this technique we were able to bring

the CLS for the page down to 0, the image suggestions for layout shifts were eliminated and there was a perceptible improvement in user experience.
Note: Browser support for CSS aspect ratio improved after we worked on the Movies application, but if we were building it today we would likely use that feature.



Preconnects

Preconnects allow you to provide hints to the browser on what resources are going to be required by the page shortly. Adding "`rel=preconnect`" informs the browser that the page will need to establish a connection to another domain so that it can start the process sooner. The hints help to load resources quicker because the browser may have already set up the connection by the time the resources are required.

There was a small but discernible difference in the values of performance parameters after this change as tabulated here.

Performance Metric	FCP (s)	Speed Index (s)	LCP (s)	TTI (s)	TBT (ms)	CLS	Performance (%)
Before	0.9	3.9	3.43	2.93	60	0	88
After	0.83	3.5	2.86	2.63	53.33	0	93.33
% Change	7.77	10.25	16.61	10.23	6.67	0	

Optimize the API call sequence

Being a TMDB client, the movies app makes several API calls to get the list of movies, genres, cast, and other details along with related images. The principle used to optimize the API call sequence should ensure that calls to fetch data to be used for rendering the main page area are not put off until the other API calls have finished. With this in mind, we changed our sequence of execution as follows.

Before	After
Fetch the metadata like genres and configuration while the API call for movie posters was put off until they were finished.	Fetch the metadata (used for populating the side menu) and simultaneously fetch the movie poster data.
Fetch the movie poster data	Render the home page with the fetched movie poster data.
Render the home page with the fetched movie poster data.	

Preloading API response

When a user visits the home page of the Movies app for the first time, we already know that we will be showing them page 1 of the ‘Popular' movies list. The actual list itself comes from the TMDB API, but the API call can be created based on these two values `Genre = "popular"` and `page = 1`

With this knowledge, we were able to preload the data for the home page as follows.

```
<link
  rel='preload'
  as='fetch'
  href='https://api.themoviedb.org/3/movie/popular?api_key=...5&page=1'
  crossOrigin='true'
/>
```

This was used only on the home page as we cannot predict what the users will click/pick on the other pages. If the preloaded data is not used, it will be a waste of resources resulting in a warning like this which can be seen in Chrome Dev Tools - "`The resource https://api.themoviedb.org/3/movie/popular?api_key=844dba0bfd8f3a4f3799f6130ef9e335&page=1 was preloaded using link preload but not used within a few seconds from the window's load event. Please make sure it has an appropriate as value and it is preloaded intentionally.`"

The LCP and TTI improved by 12.65% and 7.76% respectively after this change while the overall performance went up from 91% to 94% for the home page.

Preloading the logo and the TMDB trademark

The logo and TMDB trademark are displayed on all pages and we found the performance after preloading these to be improved. These were preloaded using a media query.

```
<link
  rel='preload'
  href={LOGO_IMAGE_PATH}
  as='image'
  media='(min-width: 80em)'
/>
<link
  rel='preload'
  href={DARK_TMDB_IMAGE_PATH}
  as='image'
  media='(prefers-color-scheme: dark) and (min-width: 80em)'
/>
<link
  rel='preload'
  href={LIGHT_TMDB_IMAGE_PATH}
  as='image'
  media='(prefers-color-scheme: light) and (min-width: 80em)'
/>
```

This resulted in a 5-6% improvement in FCP and Speed Index.

Making the site responsive

The movies app uses Next.js SSR to render the wrapper for the UI. Since the app can be accessed on both desktop and mobile devices, responsive design was essential. Combining responsive design with SSR has been a challenge because:

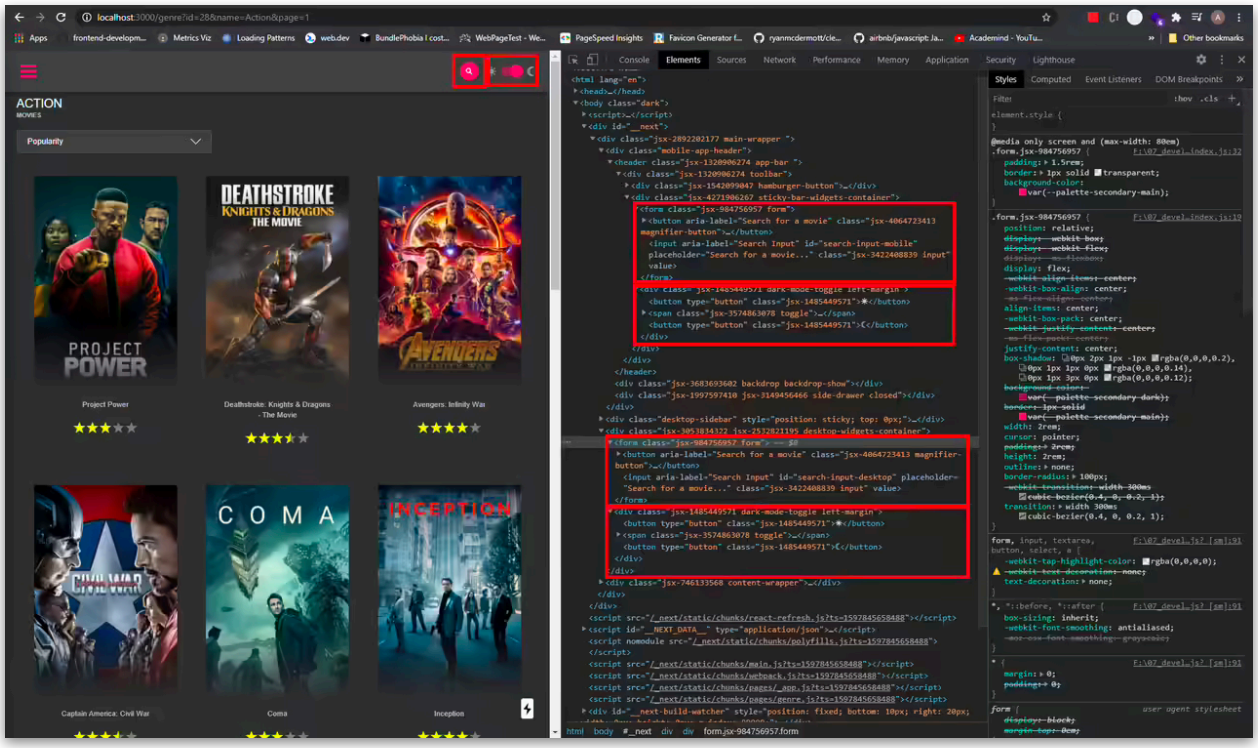
1. The server where the content is rendered does not recognize the client `window` element. Thus methods like `window.matchmedia()` cannot be used to determine client details. Additionally, client hints are not supported across all browsers.

2. Using CSS media query would result in rendering all of the elements regardless of whether they are used either on desktop or mobile.

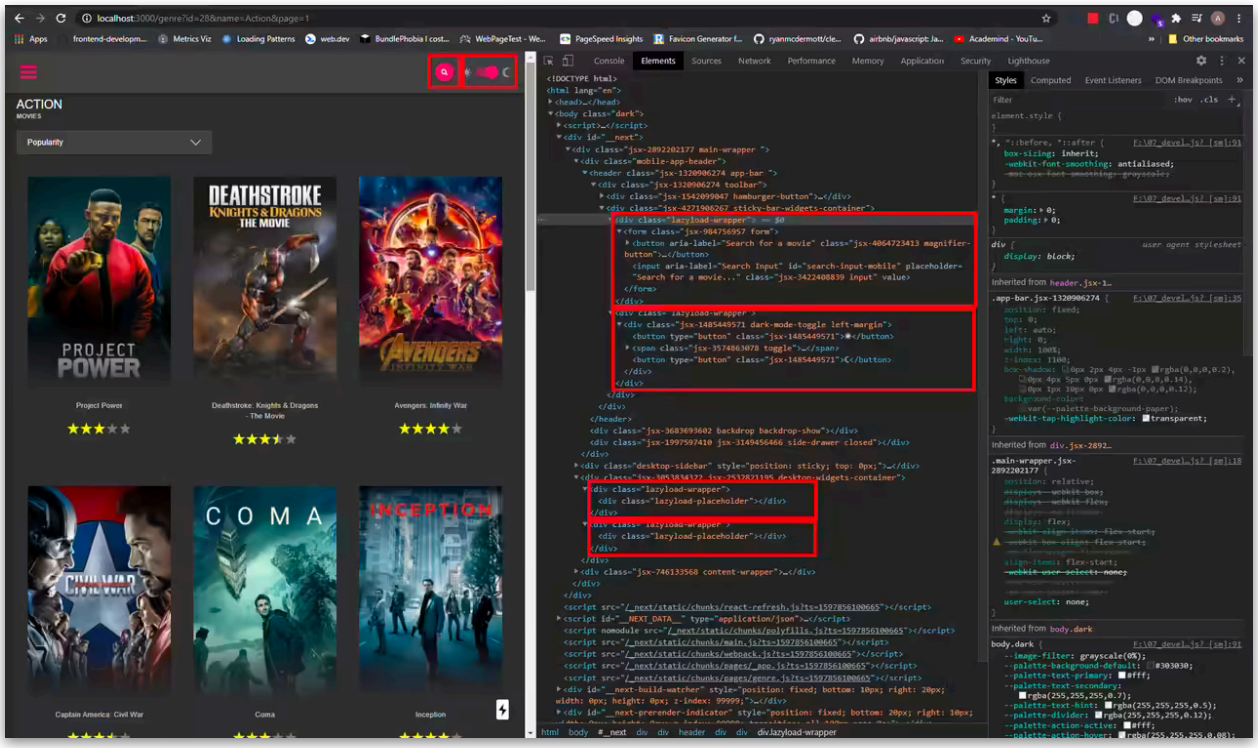
To address these challenges we used the `@artsy/fresnel` library. The approach used here is that the server would still render all elements in the DOM with CSS breakpoints. Only components that match the breakpoints would be mounted. We were thus able to avoid duplicate markup and unnecessary rendering

The following images compare the difference in markup rendered before and after the change for the same content.

Before



After



Following is the change in Lighthouse performance observed after the change.

Performance Parameter	FCP (s)	Speed Index (s)	LCP (s)	TTI (s)	TBT (ms)	CLS	Performance (%)
Before	0.93	3.73	2.6	2.63	60	0.001	94.33
After	1.06	3.23	2.66	2.66	63.33	0	95
% Change	13.97	13.4	2.3	1.14	5.55	100	

While there is some regression in FCP, LCP, TTI, and TBT, the speed index and performance have improved. The chunk size has increased due to the contribution of the artsy/fresnel bundle. However, the reduction in markup may make this a good trade-off.

Enable Google Analytics

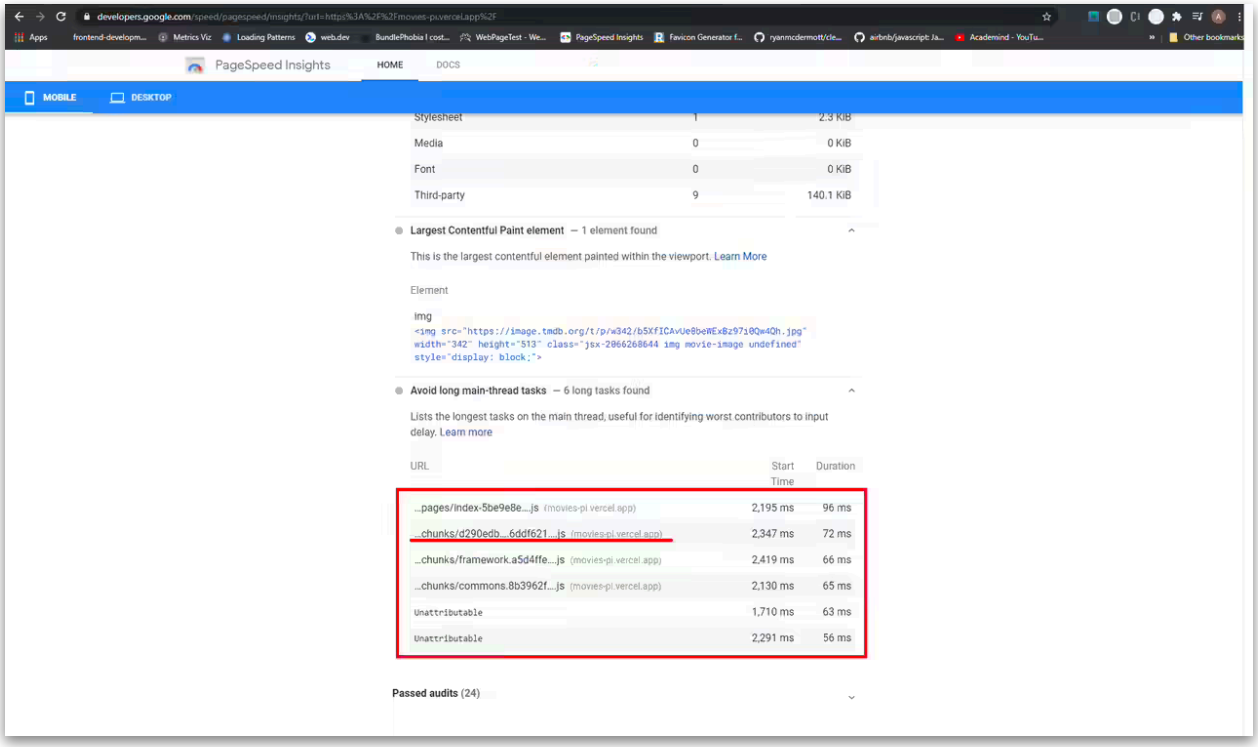
Google analytics was included on the site so that we can get a better picture of how the app engages with its users. Some regression was expected after including Google Analytics. The change in performance was captured as per our process to track performance variations for the code changes. There was some regression as expected due to the inclusion of the analytics component.

Performance Parameter	FCP (s)	Speed Index (s)	LCP (s)	TTI (s)	TBT (ms)	CLS	Performance (%)
Before	0.8	3.4	2.53	1.8	26.66	0	95.66
After	0.95	3.7	2.93	2.13	35	0	92.75
% Change	18.75	8.82	15.61	18.05	31.28	0	

Ideas that did not help

Based on the Lighthouse report's feedback, there were some alternatives and ideas that we tried but gave up because there were no performance benefits.

1. We are using the `react-lazyload` package for lazy loading images. This was listed in the long main thread tasks, along with the scrolling and rating components.



We tried replacing this with native image lazy-loading. Based on subsequent testing, we noticed that TBT increased from 10 ms to 117 ms for a negligible reduction in LCP. It is possible that native image lazy loading loads a few images that are near the viewport while react lazy-load only loads those that are within the viewport causing this difference in TBT.

Today, one could also use the Next.js Image Component to implement this functionality. However, since the component uses JS internally, using an HTML + CSS-based solution may perform better.

1. Before setting the aspect ratio for images, we had tried to improve CLS by setting image dimensions. Even though it is one of the recommended approaches for reducing CLS, setting image dimensions did not work so well as the aspect ratio technique that we finally implemented.

2. Tried out server-side rendering to reduce LCP but it brought about regression rather than improvement. This could be because the movie-related data and images required to render pages were fetched through TMDB API calls. This caused the server response to be slow because all API requests/responses were processed on the server.

Ideas that might help

There are a few additional opportunities for performance improvement that we might try out in the future. These range from replacing individual components with lighter alternatives to implementing full-fledged SSR. Here's what we could explore to check if it contributes to the performance of the app.

1. Implement responsive images with preloading as discussed here
2. Introduce caching using service-workers.
3. Currently, the `_app.js` file is slightly bloated as it includes redux-related logic eg., actions, reducers, etc. Individual pages do not need all of these files when landing. We could try eliminating redux or apply code-splitting for redux logic.
4. Implement SSR without redux and try SSR caching.
5. Replace `react-modal-video` with a lightweight alternative.
6. Use `keen-slider` instead of `react-slider`.
7. Use `react-cool-inview` instead of `react-lazyload`.
8. Apply lazy-loading/code-splitting techniques to load third party libraries using different React loading patterns
9. Image post-processing to preload the first few images like the hero-image.
10. Replace the SVG loading spinner with something that uses CSS animation.
11. Use lighter components that use HTML and CSS for rendering images instead of component that uses JavaScript internally.

Conclusion

Performance optimization is an ongoing process. Over the last 6 months, we covered a lot of ground with these changes to not only incorporate but also test many recommended best practices. We could always do more. However, at some point, you have to decide whether the gain in performance is justified by time spent on testing different alternatives. The loop will of course be repeated as and when new features are added. We however wanted to

capture our takeaways from this journey so that they serve as a manual for our future endeavors as well as yours.

With special thanks to Anton Karlovskiy and Leena Sohoni-Kasture for their contributions to this article.

Islands Architecture

The islands architecture encourages small, focused chunks of interactivity within server-rendered web pages

-

he islands architecture encourages small, focused chunks of interactivity within server-rendered web pages. The output of islands is progressively enhanced HTML, with more specificity around how the enhancement occurs. Rather than a single application being in control of full-page rendering, there are multiple entry points. The script for these "islands" of interactivity can be delivered and hydrated independently, allowing the rest of the page to be just static HTML.

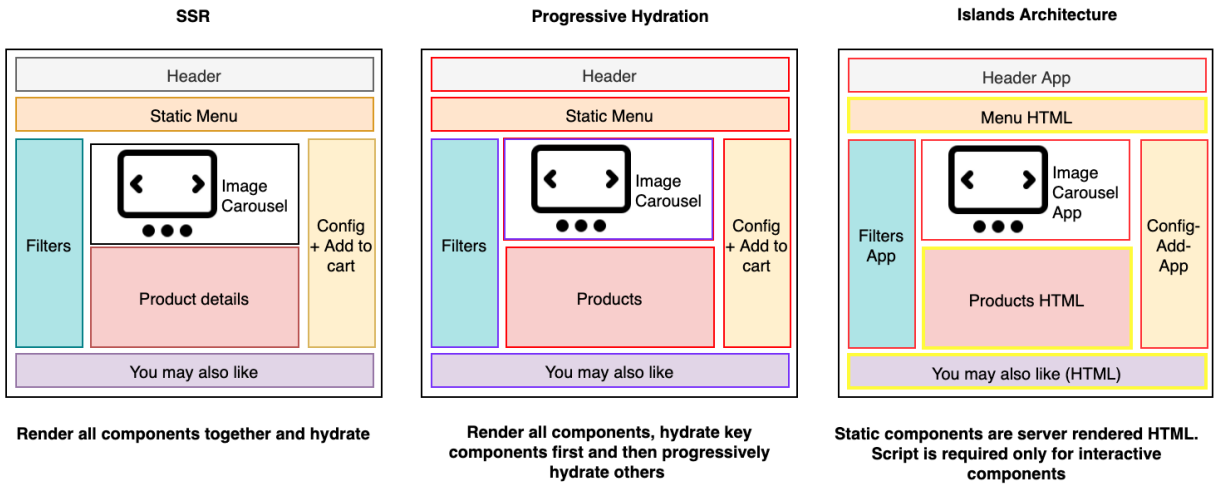
Loading and processing excess JavaScript can hurt performance. However, some degree of interactivity and JavaScript is often required, even in primarily static websites. We have discussed variations of Server Side Rendering (SSR) that enable you to build applications that try to find the balance between:

- Interactivity comparable to Client-Side Rendered (CSR) applications
- SEO benefits that are comparable to SSR applications.

The core principle for SSR is that HTML is rendered on the server and shipped with necessary JavaScript to rehydrate it on the client. Rehydration is the process of regenerating the state of UI components on the client-side after the server renders it. Since rehydration comes at a cost, each variation of SSR tries to optimize the rehydration process. This is mainly achieved

by partial hydration of critical components or streaming of components as they get rendered. However, the net JavaScript shipped eventually in the above techniques remains the same.

The term Islands architecture was popularized by Katie Sylor-Miller and Jason Miller to describe a paradigm that aims to reduce the volume of JavaScript shipped through "islands" of interactivity that can be independent delivered on top of otherwise static HTML. Islands are a component-based architecture that suggests a compartmentalized view of the page with static and dynamic islands. The static regions of the page are pure non-interactive HTML and do not need hydration. The dynamic regions are a combination of HTML and scripts capable of rehydrating themselves after rendering.



Let us explore the Islands architecture in further detail with the different options available to implement it at present.

Islands of dynamic components

Most pages are a combination of static and dynamic content. Usually, a page consists of static content with sprinkles of interactive regions that can be isolated. For example;

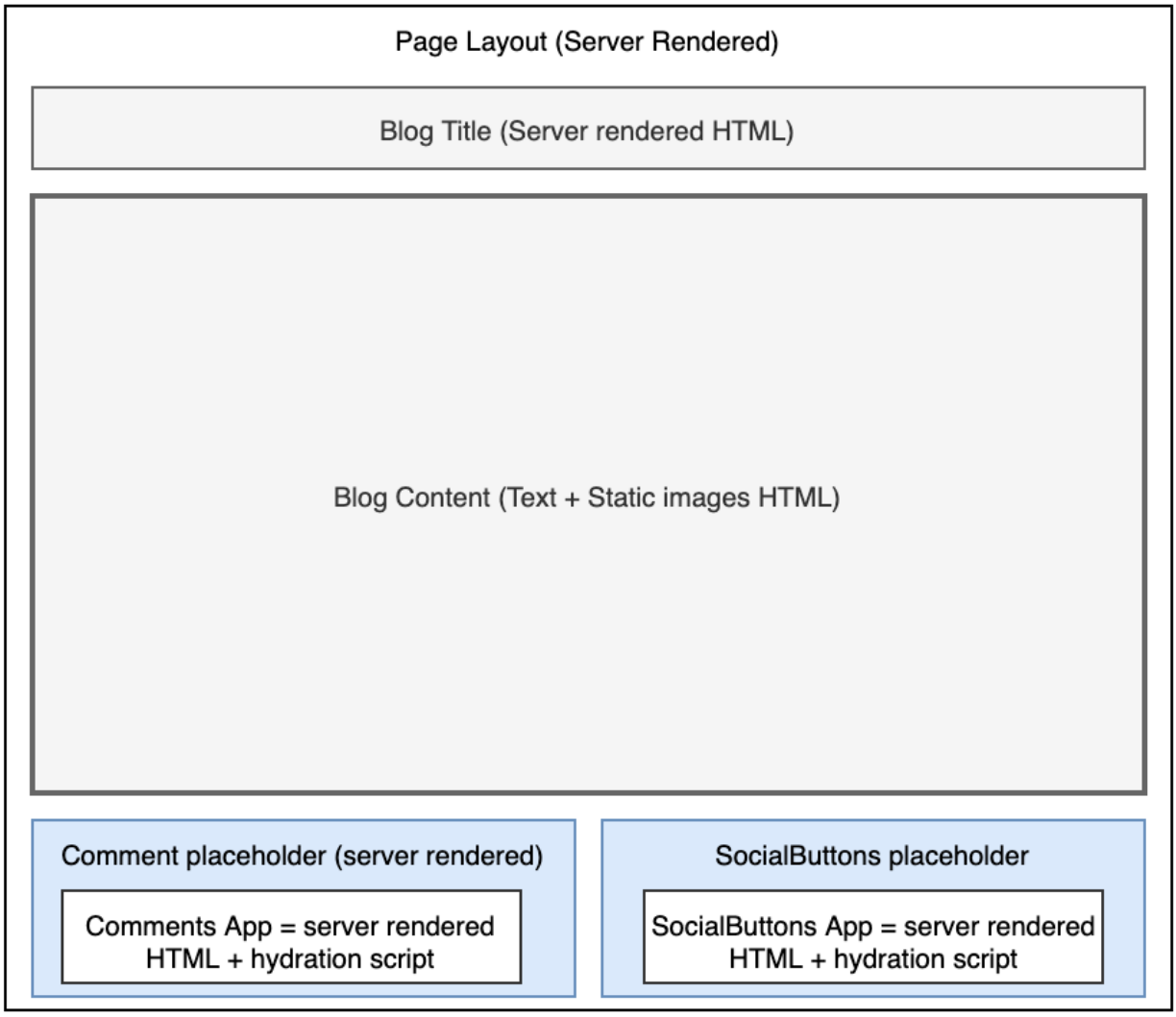
- Blog posts, news articles, and organization home pages contain text and images with interactive components like social media embeds and chat.
- Product pages on e-commerce sites contain static product descriptions and links to other pages on the app. Interactive components such as image carousels and search are available in different regions of the page.
- A typical bank account details page contains a list of static transactions with filters providing some interactivity.

Static content is stateless, does not fire events, and does not need rehydration after rendering. After rendering, dynamic content (buttons, filters, search bar) has to be rewired to its events. The DOM has to be regenerated on the client-side (virtual DOM). This regeneration, rehydration, and event handling functions contribute to the JavaScript sent to the client.

The Islands architecture facilitates server-side rendering of pages with all of their static content. However, in this case, the rendered HTML will include placeholders for dynamic content. The dynamic content placeholders contain

self-contained component widgets. Each widget is similar to an app and combines server-rendered output and JavaScript used to hydrate the app on the client.

In progressive hydration, the hydration architecture of the page is top-down. The page controls the scheduling and hydration of individual components. Each component has its hydration script in the Islands architecture that executes asynchronously, independent of any other script on the page. A performance issue in one component should not affect the other.



Implementing Islands

The Island architecture borrows concepts from different sources and aims to combine them optimally. Template-based static site generators such as Jekyll and Hugo support the rendering of static components to pages. Most modern JavaScript frameworks also support isomorphic rendering, which allows you to use the same code to render elements on the server and client.

Jason's post suggests the use of requestIdleCallback() to implement a scheduling approach for hydrating components. Static isomorphic rendering and scheduling of component level partial hydration can be built into a framework to support Islands architecture. Thus, the framework should

- Support static rendering of pages on the server with zero JavaScript.
- Support embed of independent dynamic components via placeholders in static content. Each dynamic component contains its scripts and can hydrate itself using requestIdleCallback() as soon as the main thread is free.
- Allow isomorphic rendering of components on the server with hydration on the client to recognize the same component at both ends.

You can use one of the out-of-the-box options discussed next to implement this.

Frameworks

Different frameworks today are capable of supporting the Islands architecture. Notable among them are

- **Marko**: Marko is an open-source framework developed and maintained by eBay to improve server rendering performance. It supports Islands architecture by combining streaming rendering with automatic partial hydration. HTML and other static assets are streamed to the client as soon as they are ready. Automatic partial hydration allows interactive components to hydrate themselves. Hydration code is only shipped for interactive components, which can change the state on the browser. It is isomorphic, and the Marko compiler generates optimized code depending on where it will run (client or server).

- **Astro**: Astro is a static site builder that can generate lightweight static HTML pages from UI components built in other frameworks such as React, Preact, Svelte, Vue, and others. Components that need client-side JavaScript are loaded individually with their dependencies. Thus it provides built-in partial hydration. Astro can also lazy-load components depending on when they become visible. We have included a sample implementation using Astro in the next section.

- **Eleventy + Preact:** Markus Oberlehner demonstrates the use of Eleventy, a static site generator with isomorphic Preact components that can be partially hydrated. It also supports lazy hydration. The component itself declaratively controls the hydration of the component. Interactive components use a `WithHydration` wrapper so that they are hydrated on the client.

Note that Marko and Eleventy pre-date the definition of Islands provided by Jason but contain some of the features required to support it. **Astro**, however, was built based on the definition and inherently supports the Islands architecture. In the following section, we demonstrate the use of Astro for a simple blog page example discussed earlier.

Sample implementation

The following is a sample blog page that we have implemented using Astro. The page SamplePost imports one interactive component, SocialButtons. This component is included in the HTML at the required position via markup.

SamplePost.astro

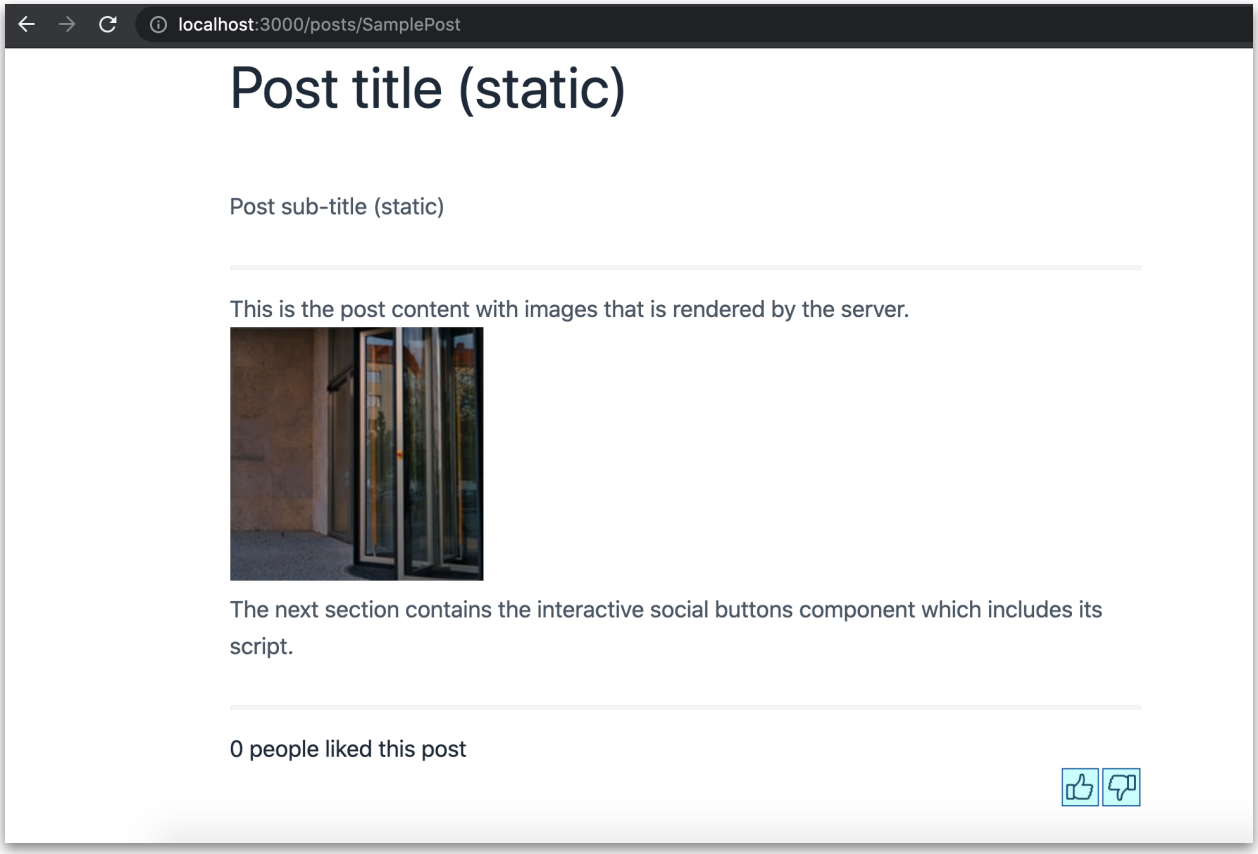
```
---
// Component Imports
import { SocialButtons } from '../../components/SocialButtons.tsx';
---

<html lang="en">
 <head>
   <link rel="stylesheet" href="/blog.css" />
 </head>

 <body>
   <div class="layout">
     <article class="content">
       <section class="intro">
         <h1 class="title">Post title (static)</h1>
         <br/>
         <p>Post sub-title (static)</p>
       </section>
       <section class="intro">
           <p>This is the  post content with images that is rendered by the server.</p>
           <img src="https://source.unsplash.com/user/c_v_r/200x200" />
           <p>The next section contains the interactive social buttons component which
includes its script.</p>
       </section>
       <section class="social">
           <div>
           <SocialButtons client:visible></SocialButtons>
           </div>
       </section>
     </article>
   </div>
 </body>
</html>
```

The `SocialButtons` component is a Preact component with its HTML, and corresponding event handlers included.

SocialButtons.tsx

```
import { useState } from 'preact/hooks';

/** a counter written in Preact */
export function SocialButtons() {
 const [count, setCount] = useState(0);
 const add = () => setCount((i) => i + 1);
 const subtract = () => setCount((i) => i - 1);

 return (
   <>
     <div>
       {count} people liked this post
     </div>
     <div align="right">
       <img src="/like.png" width="32" height="32" onclick={add}></img>
       <img src="/unlike.png" width="32" height="32" onclick={subtract}></img>
     </div>
   </>
 );
}
```

The component is embedded in the page at run time and hydrated on the client-side so that the click events function as required.



Astro allows for a clean separation between HTML, CSS, and scripts and encourages component-based design. It is easy to install and start building websites with this framework.

Pros and Cons

The Islands architecture combines ideas from different rendering techniques such as server-side rendering, static site generation, and partial hydration. Some of the potential benefits of implementing islands are as follows.

- **Performance**: Reduces the amount of JavaScript code shipped to the client. The code sent only consists of the script required for interactive components, which is much less than the script needed to recreate the virtual DOM for the entire page and rehydrate all the elements on the

page. The smaller size of JavaScript automatically corresponds to faster page loads and Time to Interactive (TTI).

Comparisons for Astro with documentation websites created for Next.js and Nuxt.js have shown an 83% reduction in JavaScript code. Other users have also reported performance improvements with Astro.

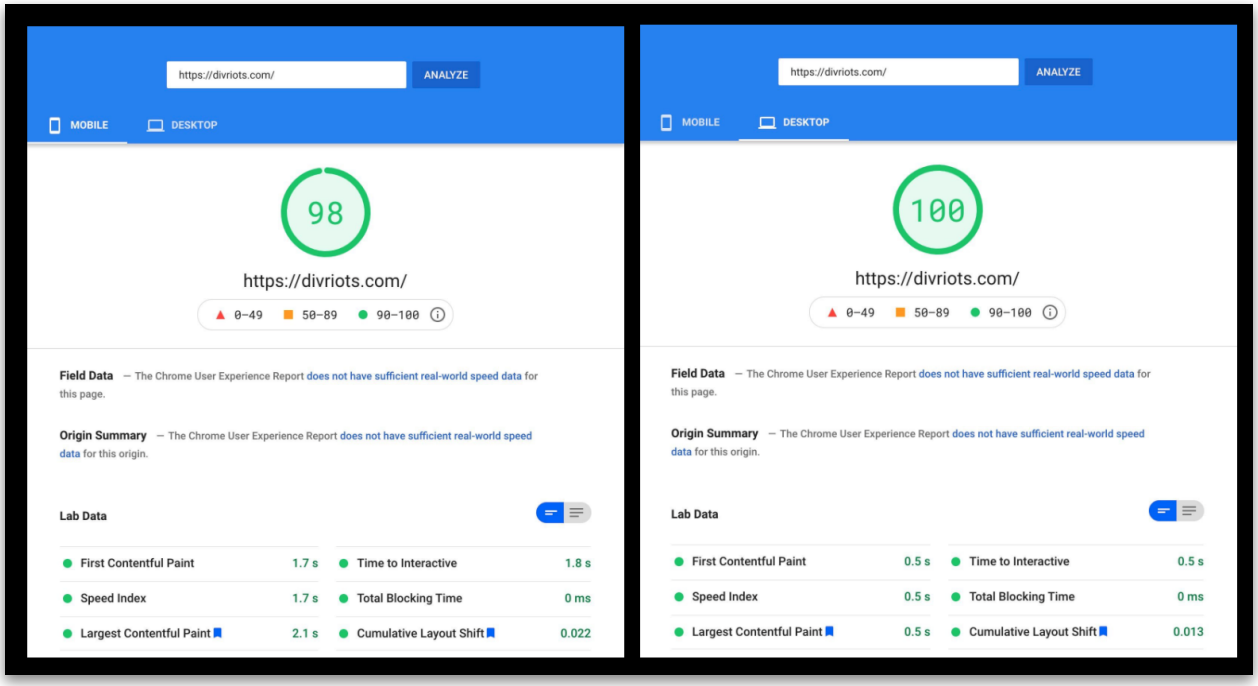


Image Courtesy: https://divriots.com/blog/our-experience-with-astro/

- **SEO:** Since all of the static content is rendered on the server; pages are SEO friendly.

- **Prioritizes important content:** Key content (especially for blogs, news articles, and product pages) is available almost immediately to the user. Secondary functionality for interactivity is usually required after consuming the key content becomes available gradually.

- **Accessibility:** The use of standard static HTML links to access other pages helps to improve the accessibility of the website.

- **Component-based:** The architecture offers all advantages of component-based architecture, such as reusability and maintainability.

Despite the advantages, the concept is still in a nascent stage. The limited support results in some disadvantages.

- The only options available to developers to implement Islands are to use one of the few frameworks available or develop the architecture yourself. Migrating existing sites to Astro or Marko would require additional efforts.

- Besides Jason's initial post, there is little discussion available on the idea.

- New frameworks claim to support the Islands architecture making it difficult to filter the ones which will work for you.

- The architecture is not suitable for highly interactive pages like social media apps which would probably require thousands of islands.

The Islands architecture concept is relatively new but likely to gain speed due to its performance advantages. It underscores the use of SSR for rendering static content while supporting interactivity through dynamic components with minimal impact on the page's performance. We hope to see many more players in this space in the future and have a wider choice of implementation options available.

PERFORMANCE

Optimize your loading sequence

Learn how to optimize your loading sequence to improve how quickly your app is usable

Note: This article is heavily influenced by insights from the Aurora team in Chrome, in particular Shubhie Panicker who has been researching the optimal loading sequence.

In every successful web page load, some critical components and resources become available at just the right time to give you a smooth loading experience. This ensures users perceive the performance of the application to be excellent. This excellent user experience should generally also translate to passing the Core Web Vitals.

Key metrics such as First Content Paint, Largest Contentful Paint, First Input Delay, etc used to measure performance are directly dependent on the loading sequence of critical resources. For example, the page cannot have its LCP if a critical resource like the hero image is not loaded. This article talks about the relationship between the loading sequence of resources and web vitals. Our objective is to provide clear guidance on how to optimize the loading sequence for better web vitals.

Before we establish an ideal loading sequence, let us first try to understand why it is so difficult to get the loading sequence right.

Why is optimal loading difficult to achieve?

We have had the unique opportunity to work on performance analysis for many of our partner's websites. We identified multiple similar issues that plagued the efficient loading of pages across different partner sites.

There is often a critical gap between developers' expectations and how the browser prioritizes resources on the page. This often results in sub-optimal performance scores. We analyzed further to discover what caused this gap and the following points summarize the essence of our analysis.

Sub-optimal sequencing
Web Vitals optimization requires not only a good understanding of what each metrics stands for but also the order in which they occur and how they relate to different critical resources. FCP occurs before LCP which occurs before FID. As such, resources required for achieving FCP should be prioritized over those required by LCP followed by those required by FID.

Resources are often not sequenced and pipelined in the correct order. This may be because developers are not aware of the dependency of metrics on resource loads. As a result, relevant resources are sometimes not available at the right time for the corresponding metric to trigger.

Examples:
a) By the time FCP fires, the hero image should be available for firing LCP.
b) By the time LCP fires, the JavaScript (JS) should be downloaded, parsed and ready (or already executing) to unblock interaction (FID).

Network/CPU Utilization

Resources are also not pipelined appropriately to ensure full CPU and Network utilization. This results in "Dead Time" on the CPU when the process is network bound and vice versa.

A great example of this is scripts that may be downloaded concurrently or sequentially. As the bandwidth gets divided during concurrent download, the total time for downloading all scripts is the same for both sequential and concurrent downloads. If you download scripts concurrently, the CPU is underutilized during the download. However, if you download the scripts sequentially, the CPU can start processing the first one as soon as it is downloaded. This results in better CPU and Network utilization.

Third-Party (3P) Products

3P libraries are often required to add common features and functionality to the website. Third parties include ads, analytics, social widgets, live chat, and other embeds that power a website. A third party library comes with its own JavaScript, images, fonts etc.

3P products don't usually have an incentive to optimize for and support the consumer site's loading performance. They could have a heavy JavaScript execution cost that delays interactivity, or gets in the way of other critical resources being downloaded.

Developers who include 3P products may tend to focus more on the value they add in terms of features rather than performance implications. As a result, 3P resources are sometimes added haphazardly, without full

consideration in terms of how it fits into the overall loading sequence. This makes them hard to control and schedule.

Platform Quirks

Browsers may differ in how they prioritize requests and implement hints. Optimization is easier if you have a deep knowledge of the platform and its quirks. Behavior particular to a specific browser makes it difficult to achieve the desired loading sequence consistently.

An example of this is the preload bug on the chromium platform.
The Preload (`<link rel=preload>`) instruction can be used to tell the browser to download key resources as soon as possible. It should only be used when you are sure that the resource will be used on the current page. The bug in Chromium causes it to behave such that requests issued via `<link rel=preload>` always start before other requests seen by the preload scanner even if those have higher priority. Issues such as these put a wrench in optimization plans.

HTTP2 Prioritization

The protocol itself does not provide many options or knobs for adjusting the order and priority of resources. Even if better prioritization primitives were to be made available, there are underlying problems with HTTP2 prioritization that make optimal sequencing difficult. Mainly, we cannot predict in what order servers or CDN's will prioritize requests for individual resources. Some CDN's re-prioritize requests while others implement partial, flawed, or no prioritization.

Resource level optimization

Effective sequencing needs that the resources that are being sequenced to be served optimally so that they will load quickly. Critical CSS should be inlined, Images should be sized correctly and JS should be code-split and delivered incrementally.

The framework itself is lacking constructs that allow code-splitting and serve JS and data incrementally. Users must rely on one of the following to split large chunks of 1P JS

1. Modern React (Suspense / Concurrent mode / Data Fetching) - This is still available for experimentation only
2. Lazy loading using dynamic imports - This is not intuitive and developers need to manually identify the boundaries along which to split the code.

When code-splitting, developers need to achieve just the right granularity of chunks because of a granularity vs performance trade-off.

Higher granularity is desirable because it

1. Minimizes JS needed for individual route and on subsequent user interactions
2. Allows for caching of common dependencies. This ensures that a change in the library doesn't require re-fetching of the entire bundle.

At the same time too much granularity when code-splitting can be bad because too many small chunks lower compression rates for individual chunks and affect browser performance.

Resource optimization also requires the elimination of dead or unused code. Unnecessary or obsolete JS may be often shipped to modern browsers which negatively affects performance. JS transpiled to ES5 and bundled with polyfills is unnecessary for modern browsers. Libraries and npm packages are often not published in ES module format. This makes it hard for bundlers to tree shake and optimize.

As you might have noticed, these issues are not limited to a particular set of resources or platforms. To work around these problems, one requires an understanding of the entire tech stack and how different resources can be coalesced to achieve optimal metrics. Before we define an overall optimization strategy, let us look at how individual resource requirements can defeat our purpose.

More on Resources - Relations, Constraints, and Priorities

In the previous section, we gave a few examples of how certain resources are required for a specific event like FCP or LCP to fire. Let us try to understand all such dependencies first before we discuss a way to work with them. Following is a resource-wise list of recommendations, constraints, and gotchas that need to be considered before we define an ideal sequence.

Critical CSS

Critical CSS refers to the minimum CSS required for FCP. It is better to inline such CSS within HTML rather than import it from another CSS file. Only the CSS required for the route should be downloaded at any given time and all critical CSS should be split accordingly.

If inlining is not possible, critical CSS should be preloaded and served from the same origin as the document. Avoid serving critical CSS from multiple domains or direct use of 3rd party critical CSS like Google Fonts. Your own server could serve as a proxy for 3rd party critical CSS instead.

Delay in fetching CSS or incorrect order of fetching CSS could impact FCP and LCP. To avoid this, non-inlined CSS should be prioritized and ordered above 1P JS and ABT images on the network.

Too much inlined CSS can cause HTML bloating and long style parsing times on the main thread. This can hurt the FCP. As such identifying what is critical and code-splitting are essential.

Inlined CSS cannot be cached. One workaround for this is to have a duplicate request for the CSS that can be cached. Note however, that this can result in multiple full-page layouts which could impact FID.

Fonts

Like critical CSS, the CSS for critical fonts should also be inlined. If inlining is not possible the script should be loaded with a preconnect specified. Delay in fetching fonts, e.g., google fonts or fonts from a different domain can affect

FCP. Preconnect tells the browser to set up connections to these resources earlier.

Inlining fonts can bloat the HTML significantly and delay initiating other critical resource fetches. Font fallback may be used to unblock FCP and make the text available. However, using font fallback can affect CLS due to jumping fonts. It can also affect FID due to a potentially large style and layout task on the main thread when the real font arrives.

Above the Fold (ABT) Images

This refers to images that are initially visible to the user on page load because they are within the viewport. A special case for ABT images is the hero image for the page. All ABT images should be sized. Unsized images hurt the CLS metric because of the layout shift that occurs when they are fully rendered. Placeholders for ABT images should be rendered by the server.

Delayed hero image or blank placeholders would result in a late LCP. Moreover, LCP will re-trigger, if the placeholder size does not match with the intrinsic size of the actual hero image and the image is not overlaid on replacement. Ideally, there should be no impact on FCP due to ABT images but in practice, an image can fire FCP.

Below the Fold (BTF) Images

These are images that are not immediately visible to the user on page load. As such they are ideal candidates for lazy loading. This ensures that they do not contend with 1P JS or important 3P needed on the page. If BTF images were to be loaded before 1P JS or important 3P resources, FID would get delayed.

1P JavaScript

1P JS impacts the interaction readiness of the application. It can get delayed on the network behind images & 3P JS and on the main thread behind 3P JS. As such it should start loading before ABT images on the network and execute before 3P JS on the main thread. 1P JS does not block FCP and LCP in pages that are rendered on the server-side.

3P JavaScript

3P sync script in HTML head could block CSS & font parsing and therefore FCP. Sync script in the head also blocks HTML body parsing. 3P script execution on the main thread can delay 1P script execution and push out hydration and FID. As such, better control is required for loading 3P scripts.

These recommendations and constraints would generally apply irrespective of the tech stack and browser. Note, how something that is a recommendation can also become a constraint. For example, inlining fonts and CSS is great, but too much of it can cause bloating. The trick is to find a balance between ‘Too little Too late' and ‘Too much Too soon’.

The following chart gives us an understanding of Chrome's priorities for loading different resources. Combining the information on priorities and the discussion on resource types will help to better understand the loading sequence that is proposed in the next section.

	Layout-blocking	Load in layout-blocking phase	Load one-at-a-time in layout-blocking phase		
Net Priority	**Highest**	**Medium**	**Low**	**Lowest**	**Idle**
Blink Priority	**VeryHigh**	**High**	**Medium**	**Low**	**VeryLow**
DevTools Priority	**Highest**	**High**	**Medium**	**Low**	**Lowest**
	Main Resource				
	CSS (match)				CSS (mismatch)
		Script (early** or not from preload scanner)	Script (late**)	Script (async)	
	Font	Font (preload)			
		Import			
		Image (in viewport)		Image	
				Media	
				SVG Document	
					Prefetch
		Preload*			
		XSL			
	XHR (sync)	XHR/fetch* (async)			
			Favicon		

Following are the key takeaways from this table.

- CSS and Fonts are loaded with the highest priority. This should help us prioritize critical CSS and fonts.
- Scripts get different priorities based on where they are in the document and whether they are async, defer, or blocking. Blocking scripts requested before the first image (or an image early in the document) are given higher priority over blocking scripts requested after the first image is fetched.

  Async/defer/injected scripts, regardless of where they are in the document, have the lowest priority. Thus we can prioritize different scripts by using the appropriate attributes for async and defer.

- Images that are visible and in the viewport have a higher priority (Net: Medium) than those that are not in the viewport (Net: Lowest). This helps us prioritize ABT images over BTF images.

Let us now see how all of the above details can be put together to define an optimal loading sequence.

What is the Ideal Loading Sequence

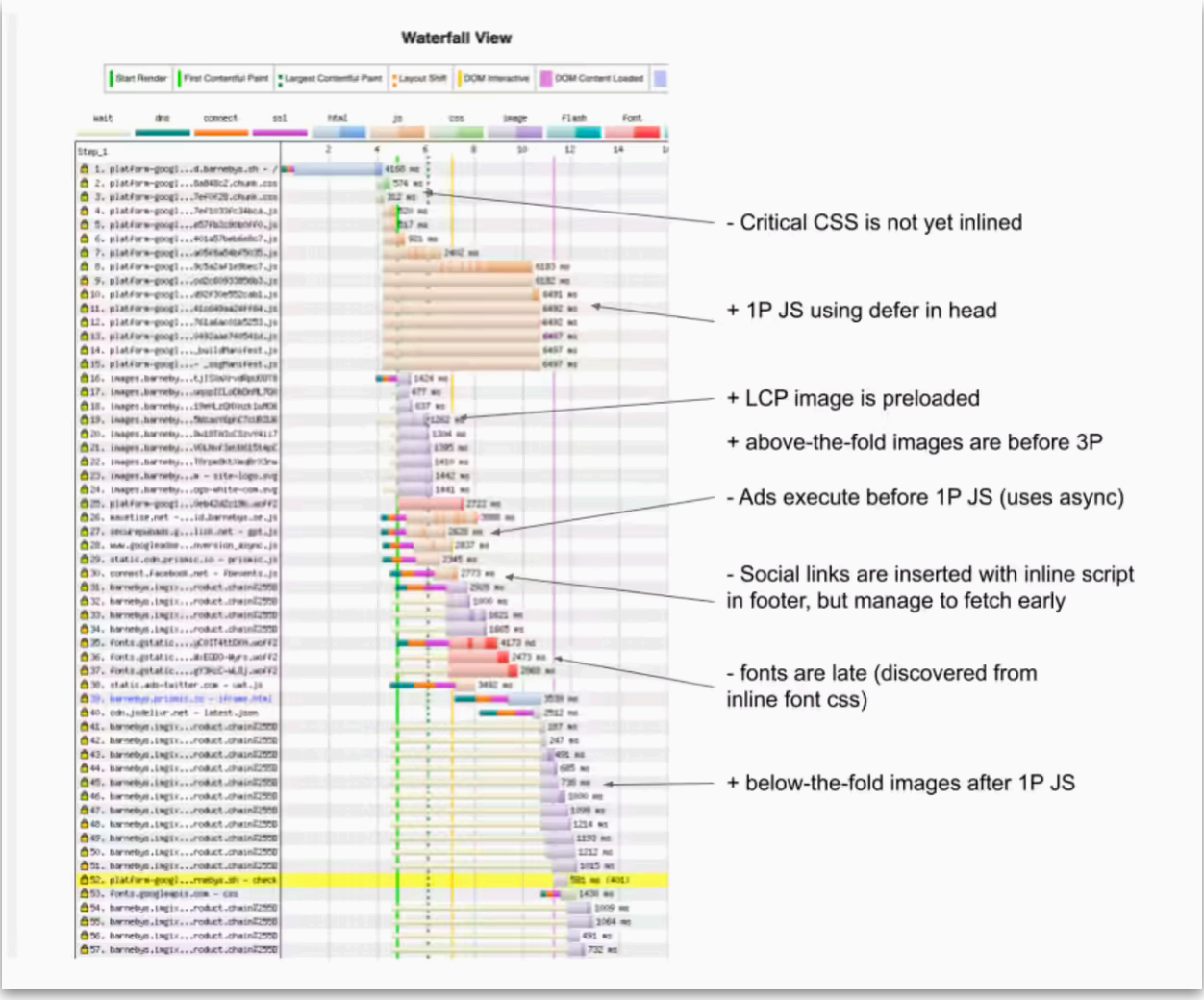
With that background, we can now propose a loading sequence that should optimize the loading of both 1P and 3P resources. The proposed sequence uses Next.js Server Side Rendering (SSR) as a reference for optimization.

Current State

Based on our experience, the following is the typical loading sequence we have observed for a Next.js SSR application before optimization.

CSS	CSS is preloaded before JS but is not inlined
JavaScript	1P JS is preloaded 3P JS is not managed and can still be render-blocking anywhere in the document.
Fonts	Fonts are neither in-lined nor do they use preconnect Fonts are loaded via external stylesheets which delays the loading Fonts may or may not be display blocking.
Images	Hero images are not prioritized Both ABT and BTF images are not optimized

Following is an example of one such sequence from one of our partner sites. The positives and negatives about the loading sequence are included as annotations.



Proposed Sequence without 3P

Following is a loading sequence that takes into account all of the constraints discussed previously. Let us first tackle a sequence without 3P. We will then see how 3P resources can be interleaved in this sequence. Note that, we have considered Google Fonts as 1P here.

Sequence of events on the main browser thread		**Sequence of requests on the network.**	
1	Parse the HTML	Small inline 1P scripts.	1
2	Execute small inline 1P scripts	Inlined critical CSS (Preload if external)	2
		Inlined critical Fonts (Preconnect if external)	3
3	Parse FCP resources (critical CSS, font)	LCP Image (Preconnect if external)	4
First Contentful Paint (FCP)		Fonts (triggered from inline font-css (Preconnect)	5
4	Render LCP resources (Hero image, text)	Non-critical (async) CSS	6
		First-party JS for interactivity	7
		Above the fold images (preconnect)	8
Largest Contentful Paint (LCP)		Below the fold images	9
5	Render important ABT images		
Visually Complete			

6	Parse Non-critical (async) CSS		
7	Execute 1P JS and hydrate	Lazy-loaded JS chunks	1(
First Input Delay (FID)			

While some parts of this sequence may be intuitive, the following points will help to justify it further.

1. We recommend avoiding preload as much as possible because it forces manual preload on every preceding resource and also causes manual curation of ordering. Preload should be especially avoided on fonts, as it is tricky to detect critical fonts.
2. Font-CSS should be ideally inlined. Fonts from another origin should be fetched using preconnect.
3. Preconnect is recommended for all resources from another origin. This will ensure that a connection is established in advance for downloading these resources.
4. Non-critical CSS should be fetched before user interaction begins (FID). This would avoid styling problems due to subsequent rendering of such CSS.
5. Start fetching first-party JS before ABT images on the network. It will take some time to download and parse the JS.
6. Parsing of the HTML on the main thread and download of ABT images can continue in parallel while 1P JS is parsed.

Proposed Sequence with 3P

Finally, we have reached the stage where we can propose a sequence for all key resources that are commonly loaded in a modern web application. Following is what the sequence for events on the main browser thread and network fetch requests will look like with 3P resources in the picture.

Sequence of events on the main browser thread		**Sequence of requests on the network.**	
1	Parse the HTML	FCP blocking 3P resources	1
		Small inline 1P scripts.	2
2	Execute small inline 1P scripts	Inlined critical CSS (Preload if external)	3
3	Parse FCP blocking 3P resources	Inlined critical Fonts (Preconnect if external)	4
4	Parse FCP resources (critical CSS, font)	3P personalized ABT image required for LCP	5
First Contentful Paint (FCP)		LCP Image (Preconnect if external)	6

5	Render 3P personalized ABT image required for LCP	Fonts (triggered from inline font-css (Preconnect)	7
		Non-critical (async) CSS	8
6	Render LCP resources (Hero image, text)	3P that must execute before first user interaction	9
		First-party JS for interactivity	1(
Largest Contentful Paint (LCP)		Above the fold images (preconnect)	1(
7	Render important ABT images	Default 3P JS	12
8	Parse Non-critical (async) CSS		
9	Execute 3P required for first user interaction	Below the fold images	13
10	Execute 1P JS and hydrate	Lazy-loaded JS chunks	14
First Input Delay (FID)		Less important 3P JS	15

The main concern here is how do you ensure that 3P scripts are downloaded optimally and in the required sequence.
Since the script request goes to another domain, preconnect is recommended for the following 3P requests. This helps to optimize the download.

#1 - FCP blocking 3P resources
#5 - 3P personalized ABT image required for LCP
#9 - 3P that must execute before first user interaction
#12 - Default 3P JS

To achieve the desired sequence, we recommend using the ScriptLoader component for Next. This component is designed to "optimize the critical rendering path and ensure external scripts don't become a bottleneck to optimal page load." The feature most relevant to our discussion is Loading Priorities. This allows us to schedule the scripts at different milestones to support different use cases. Following are the loading priority values available **After-Interactive**: Loads the specific 3P script after the next hydration. This can be used to load Tag-managers, Ads, or Analytics scripts that we want to execute as early as possible but after 1P scripts.

Before-Interactive: Loads the specific 3P script before hydration. It can be used in cases where we want the 3P script to execute before the 1P script. Eg., polyfill.io, bot detection, security and authentication, user consent management (GDPR), etc.

Lazy-Onload: Prioritize all other resources over the specified 3P script and lazy load the script. It can be used for CRM components like Google Feedback or Social Network specific scripts like those used for share buttons, comments, etc.

Thus, preconnect, script attributes and ScriptLoader for Next.js together can help us get the desired sequence for all our scripts.

Conclusion

The responsibility of optimizing apps falls on the shoulders of the creators of the platforms used as well as the developers who use it. Common issues need to be addressed. We aim to make sequencing easier from the inside out.

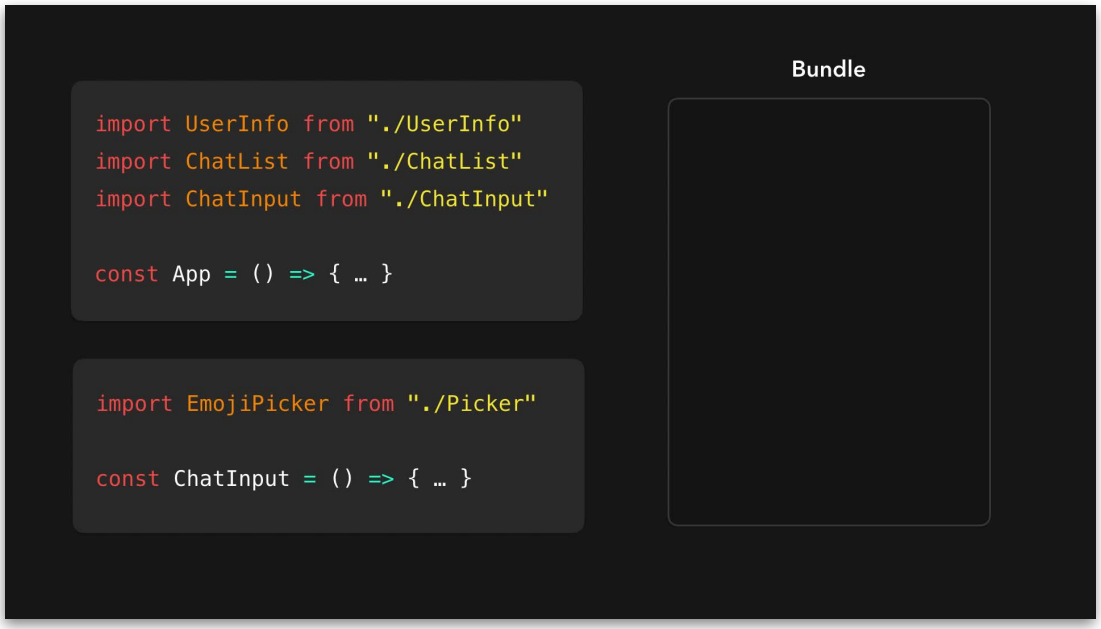
A tried and tested set of recommendations for different use cases and initiatives like the Script Loader help to achieve this for the React-Next.js stack. The next step would be to ensure that new apps conform to the recommendations above.

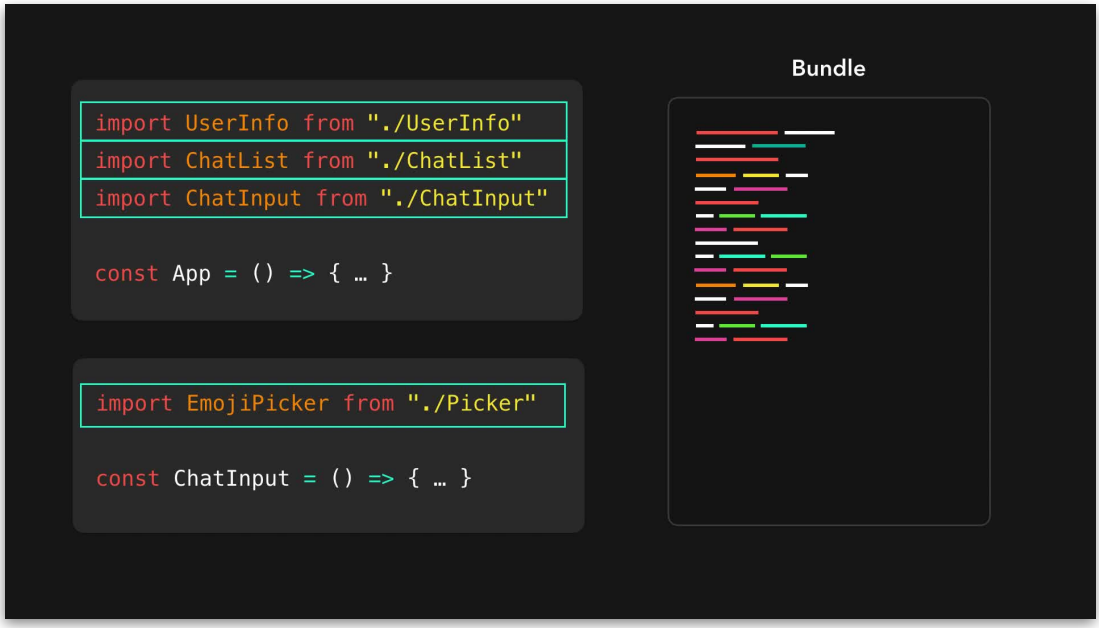
With special thanks to Leena Sohoni (Technical Analyst/Writer), for all her contributions to this write-up.

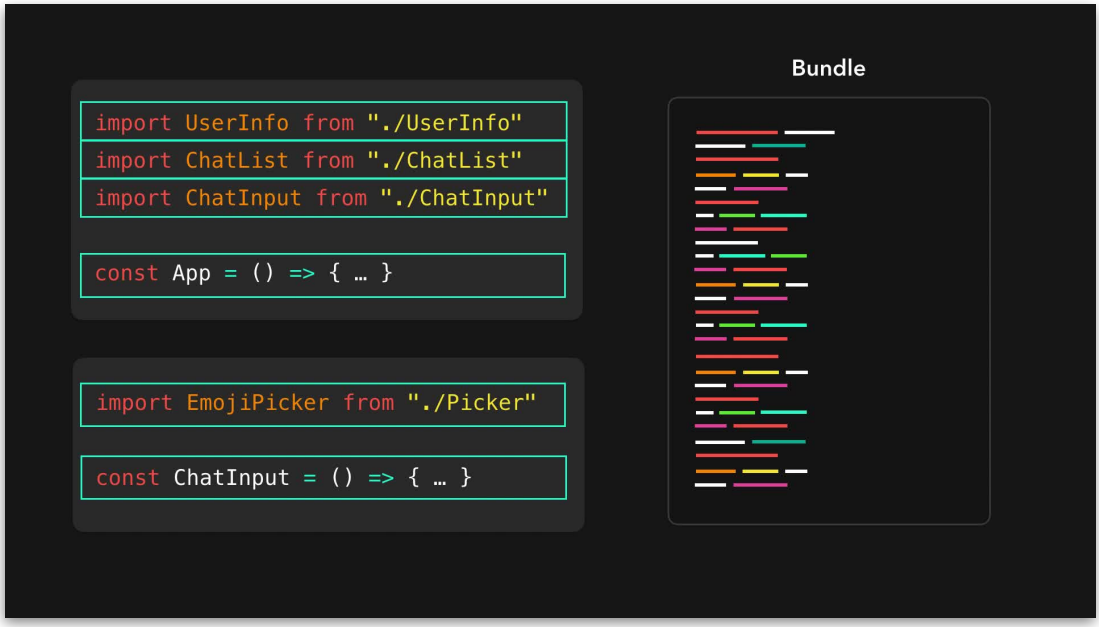
Static Import

Import code that has been exported by another module

The import keyword allows us to import code that has been exported by another module. By default, all modules we're statically importing get added to the initial bundle. A module that is imported by using the default ES2015 import syntax, import module from 'module', is statically imported.




import UserInfo from "./UserInfo"
import ChatList from "./ChatList"
import ChatInput from "./ChatInput"
const App = () => { … }
import EmojiPicker from "./Picker"
const ChatInput = () => { … }
Bundle


import UserInfo from "./UserInfo"
import ChatList from "./ChatList"
import ChatInput from "./ChatInput"
const App = () => { … }
import EmojiPicker from "./Picker"
const ChatInput = () => { … }
Bundle

Let's look at an example! A simple chat app contains a `Chat` component, in which we're statically importing and rendering three components: `UserProfile`, a `ChatList`, and a `ChatInput` to type and send messages! Within the `ChatInput` module, we're statically importing an `EmojiPicker` component to show be able to show the user the emoji picker when the user toggles the emoji.

```
import React from "react";

// Statically import Chatlist, ChatInput and UserInfo
import UserInfo from "./components/UserInfo";
import ChatList from "./components/ChatList";
import ChatInput from "./components/ChatInput";

import "./styles.css";

console.log("App loading", Date.now());

const App = () => (
  <div className="App">
    <UserInfo />
    <ChatList />
    <ChatInput />
  </div>
);
```

The modules get executed as soon as the engine reaches the line on which we import them. When you open the console, you can see the order in which the modules have been loaded!

Since the components were statically imported, Webpack bundled the modules into the initial bundle. We can see the bundle that Webpack creates after building the application.

```
Asset           Size      Chunks             Chunk Names
main.bundle.js  1.5 MiB    main  [emitted]  main
```

Our chat application's source code gets bundled into one bundle: `main.bundle.js`. A large bundle size can affect the loading time of our application significantly depending on the user's device and network connection. Before the `App` component is able to render its contents to the user's screen, it first has to load and parse all modules.

Luckily, there are many ways to speed up the loading time! We don't always have to import all modules at once: maybe there are some modules that should only get rendered based on user interaction, like the `EmojiPicker` in this case, or rendered further down the page. Instead of importing all component statically, we can dynamically import the modules after the `App` component has rendered its contents and the user is able to interact with our application.

Dynamic Import

Import parts of your code on demand

In our chat application, we have four key components: `UserInfo`, `ChatList`, `ChatInput` and `EmojiPicker`. However, only three of these components are used instantly on the initial page load: `UserInfo`, `ChatList` and `ChatInput`

The `EmojiPicker` isn't directly visible, and may not even be rendered at all if the user won't even click on the emoji in order to toggle the `EmojiPicker`. This would mean that we unnecessarily added the `EmojiPicker` module to our initial bundle, which potentially increased the loading time!

In order to solve this, we can dynamically import the `EmojiPicker` component. Instead of statically importing it, we'll only import it when we want to show the `EmojiPicker`.

An easy way to dynamically import components in React is by using React Suspense. The `React.Suspense` component receives the component that should be dynamically loaded, which makes it possible for the `App` component can render its contents faster by suspending the import of the `EmojiPicker` module!

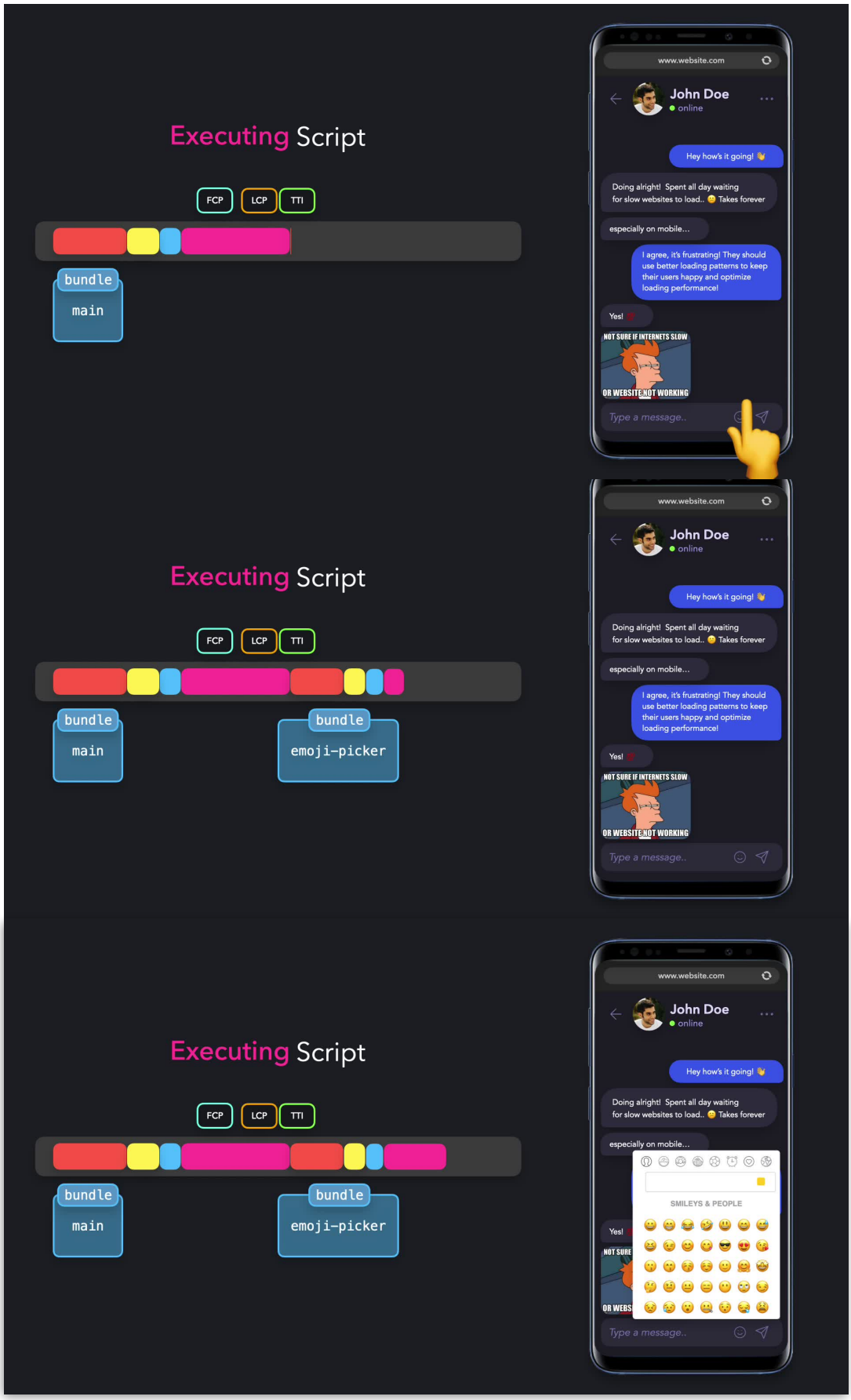
When the user clicks on the emoji, the `EmojiPicker` component gets rendered for the first time. The `EmojiPicker` component renders a `Suspense` component, which receives the lazily imported module:

the `EmojiPicker` in this case. The `Suspense` component accepts a fallback prop, which receives the component that should get rendered while the suspended component is still loading!

Instead of unnecessarily adding `EmojiPicker` to the initial bundle, we can split it up into its own bundle and reduce the size of the initial bundle! A smaller initial bundle size means a faster initial load: the user doesn't have to stare at a blank loading screen for as long. The fallback component lets the user know that our application hasn't frozen: they simply need to wait a little while for the module to be processed and executed.

```
Asset                          Size        Chunks            Chunk Names
emoji-picker.bundle.js         1.48 KiB    1    [emitted]    emoji-picker
main.bundle.js                 1.33 MiB    main [emitted]    main
vendors~emoji-picker.bundle.js 171 KiB     2    [emitted]    vendors~emoji-picker
```

Whereas previously the initial bundle was `1.5 MiB`, we've been able to reduce it to `1.33 MiB` by suspending the import of the `EmojiPicker`!

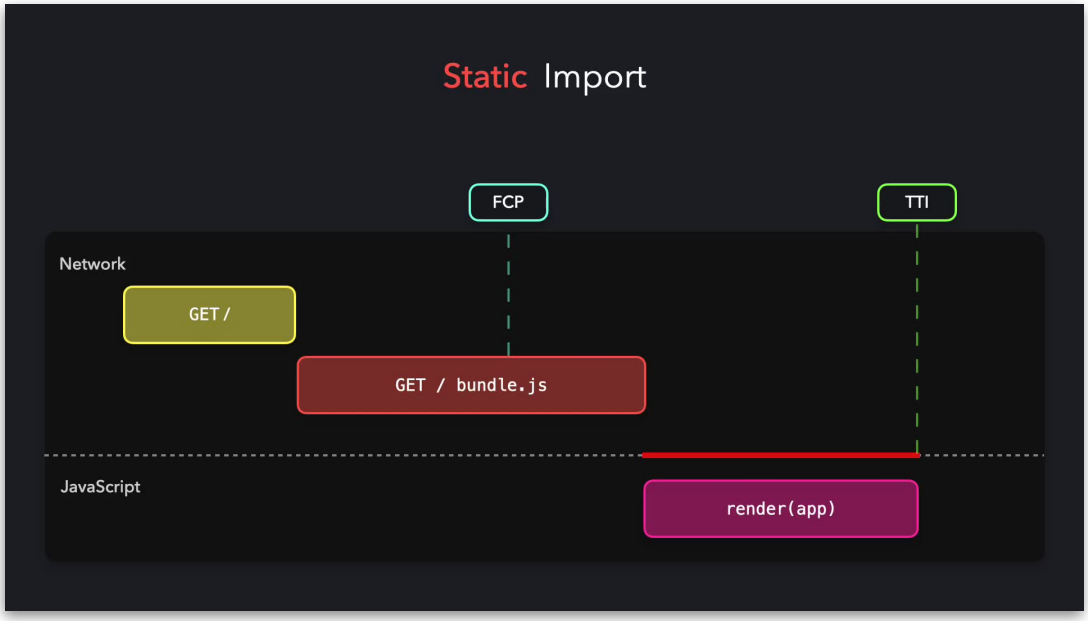

Executing Script
FCP
LCP
TTI
bundle
main
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..
Executing Script
FCP
LCP
TTI
bundle
main
bundle
emoji-picker
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..
Executing Script
FCP
LCP
TTI
bundle
main
bundle
emoji-picker
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
SMILEYS & PEOPLE
Yes!
Type a message..

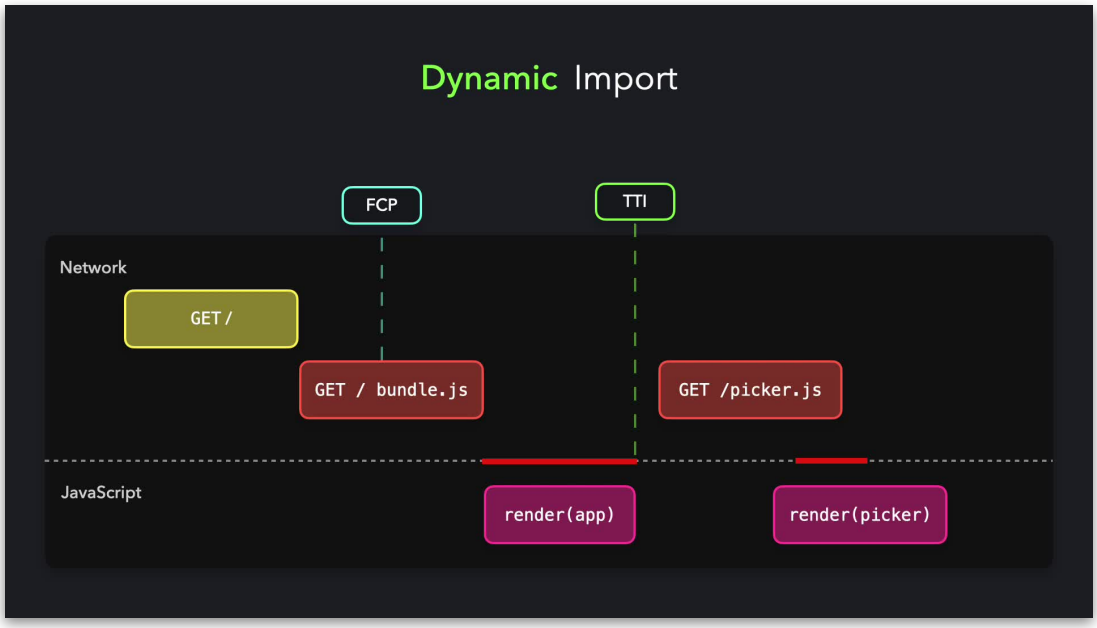
In the console, you can see that the `EmojiPicker` doesn't get executed until we've toggled the `EmojiPicker`!

```
import React, { Suspense, lazy } from "react";
  // import Send from "./icons/Send";
  // import Emoji from "./icons/Emoji";
  const Send = lazy(() =>
    import(/*webpackChunkName: "send-icon" */ "./icons/Send")
  );
  const Emoji = lazy(() =>
    import(/*webpackChunkName: "emoji-icon" */ "./icons/Emoji")
  );
  // Lazy load EmojiPicker  when <EmojiPicker /> renders
  const Picker = lazy(() =>
    import(/*webpackChunkName: "emoji-picker" */ "./EmojiPicker")
  );

  const ChatInput = () => {
    const [pickerOpen, togglePicker] = React.useReducer(
      state => !state, false
    );

    return (
      <Suspense fallback={<p id="loading">Loading...</p>}>
        <div className="chat-input-container">
          <input type="text" placeholder="Type a message..." />
          <Emoji onClick={togglePicker} />
          {pickerOpen && <Picker />}
          <Send />
        </div>
      </Suspense>
    );
  };

  console.log("ChatInput loaded", Date.now());

  export default ChatInput;
```

When building the application, we can see the different bundles that Webpack created.

By dynamically importing the `EmojiPicker` component, we managed to reduce the initial bundle size from `1.5MiB` to `1.33MiB`! Although the user may still have to wait a while until the `EmojiPicker` has been fully loaded, we have improved the user experience by making sure the application is rendered and interactive while the user waits for the component to load.





Loadable Components

Server-side rendering doesn't support React Suspense (yet). A good alternative to React Suspense is the loadable-components library, which can be used in SSR applications.

```
import React from "react";
import loadable from "@loadable/component";

import Send from "./icons/Send";
import Emoji from "./icons/Emoji";

const EmojiPicker = loadable(() => import("./EmojiPicker"), {
  fallback: <div id="loading">Loading...</div>
});

const ChatInput = () => {
  const [pickerOpen, togglePicker] = React.useReducer(state => !state, false);

  return (
    <div className="chat-input-container">
      <input type="text" placeholder="Type a message..." />
      <Emoji onClick={togglePicker} />
      {pickerOpen && <EmojiPicker />}
      <Send />
    </div>
  );
};

export default ChatInput;
```

Similar to React Suspense, we can pass the lazily imported module to the loadable, which will only import the module once the `EmojiPicker` module is being requested! While the module is being loaded, we can render a fallback component.

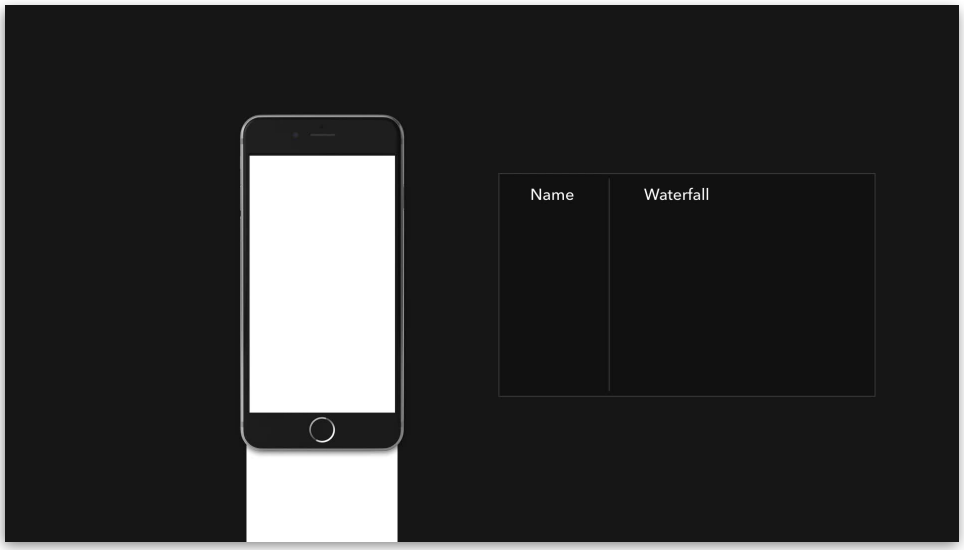
```
import React from "react";
import Send from "./icons/Send";
import Emoji from "./icons/Emoji";
import loadable from "@loadable/component";

const EmojiPicker = loadable(() => import("./components/EmojiPicker"), {
  fallback: <p id="loading">Loading...</p>
});

const ChatInput = () => {
  const [pickerOpen, togglePicker] = React.useReducer(state => !state, false);

  return (
    <div className="chat-input-container">
      <input type="text" placeholder="Type a message..." />
      <Emoji onClick={togglePicker} />
      {pickerOpen && <EmojiPicker />}
      <Send />
    </div>
  );
};

console.log("ChatInput loaded", Date.now());

export default ChatInput;
```

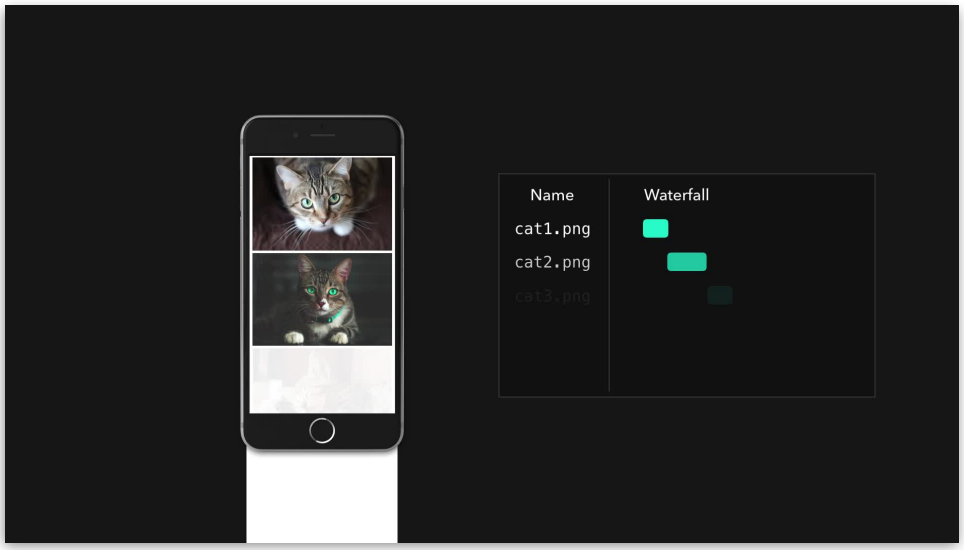
Although loadable components are a great alternative to React Suspense for SSR applications, they're also useful in CSR applications in order to suspend the import of modules.

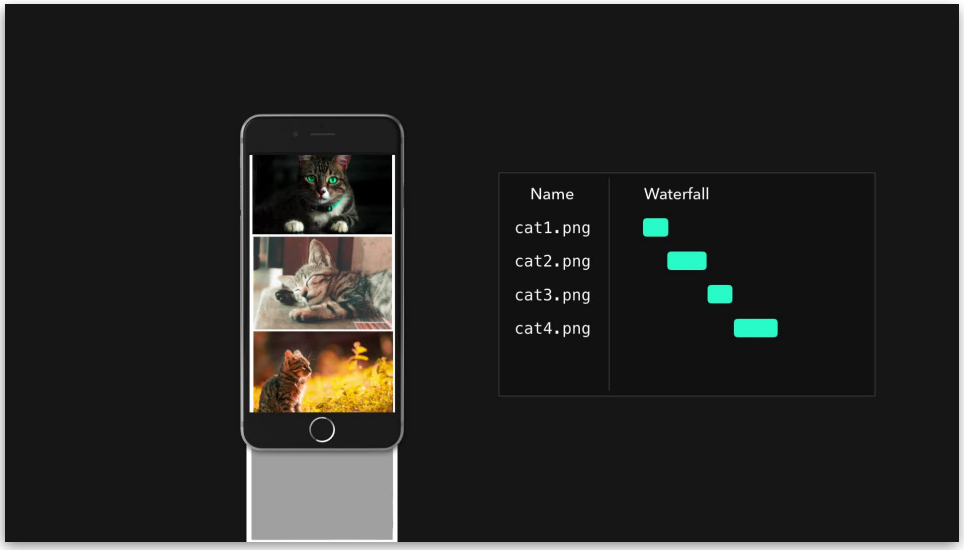
Import on Visibility

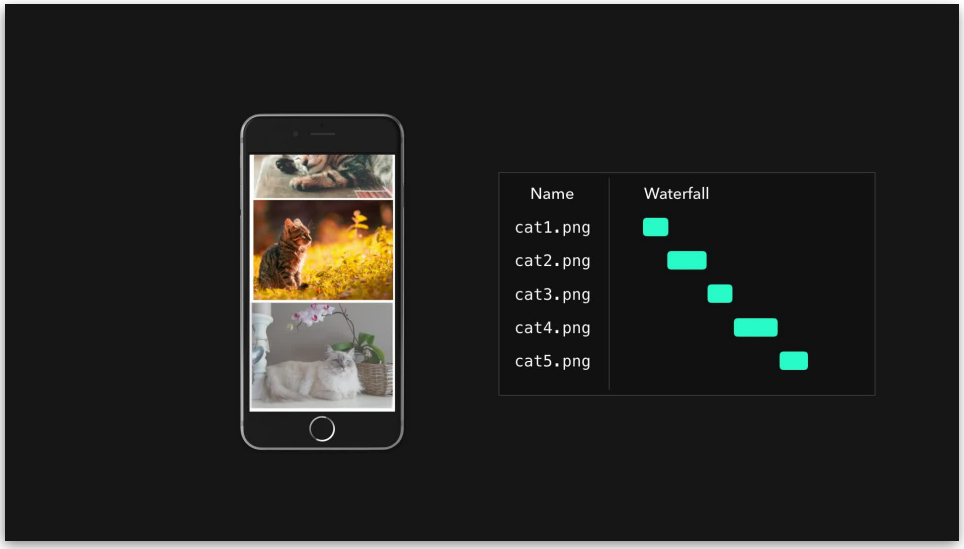
Load non-critical components when they are visible in the viewport

Besides user interaction, we often have components that aren't visible on the initial page. A good example of this is lazy loading images that aren't directly visible in the viewport, but only get loaded once the user scrolls.





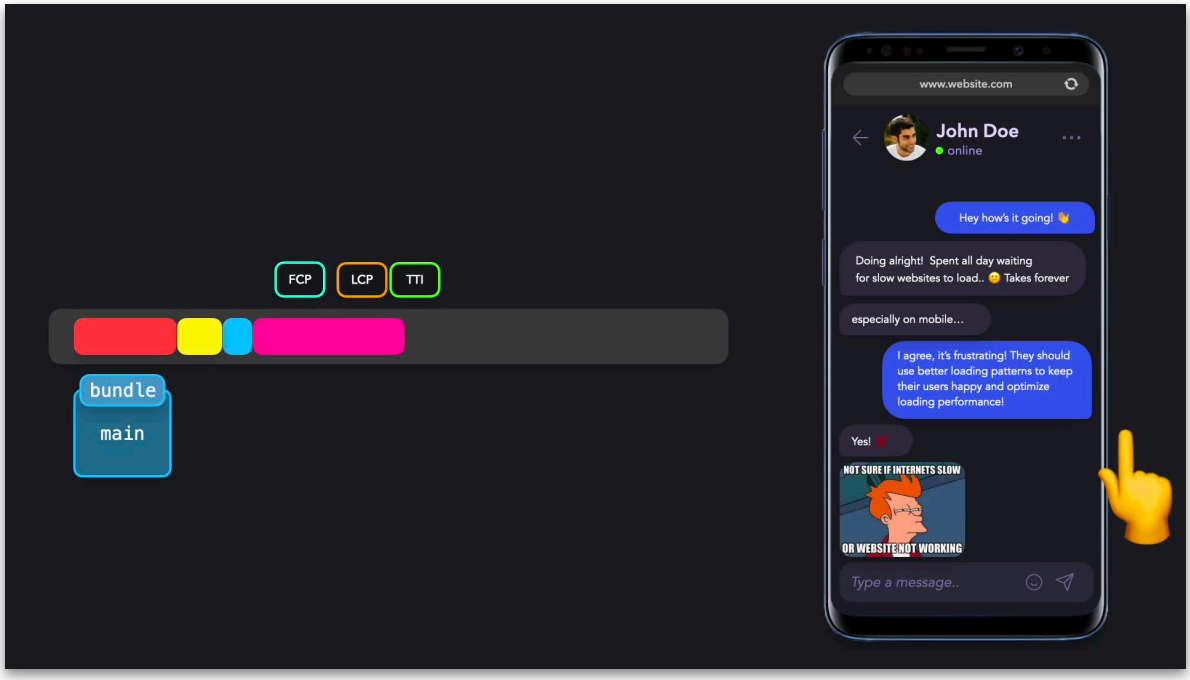


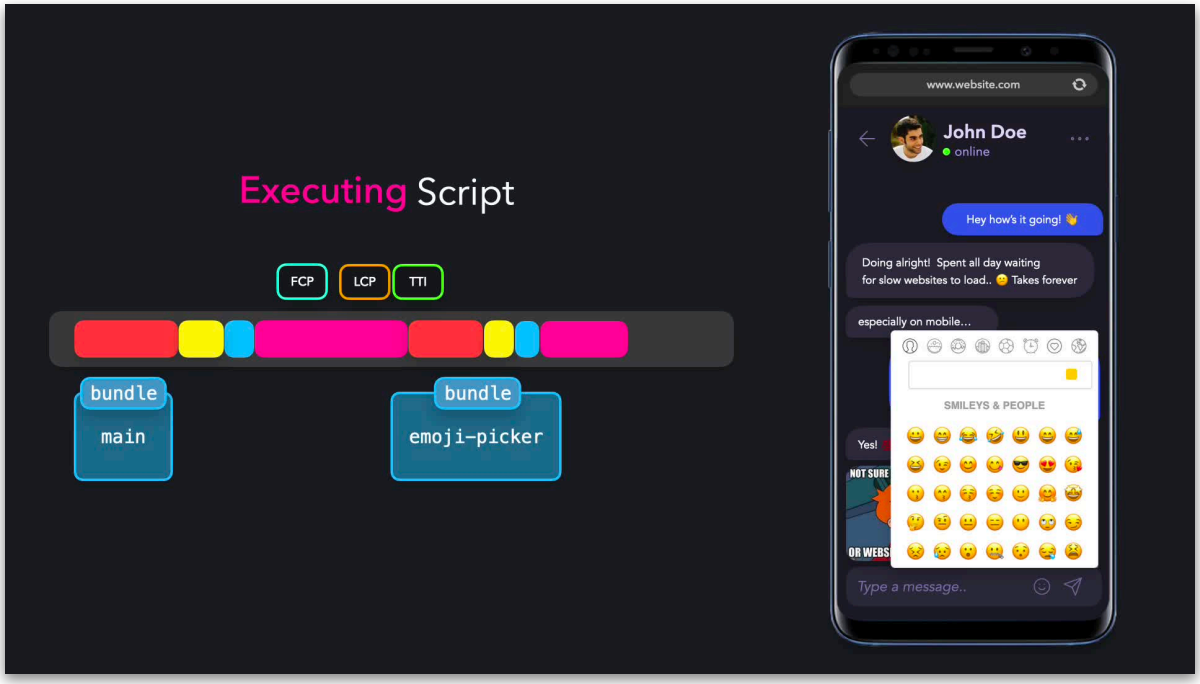


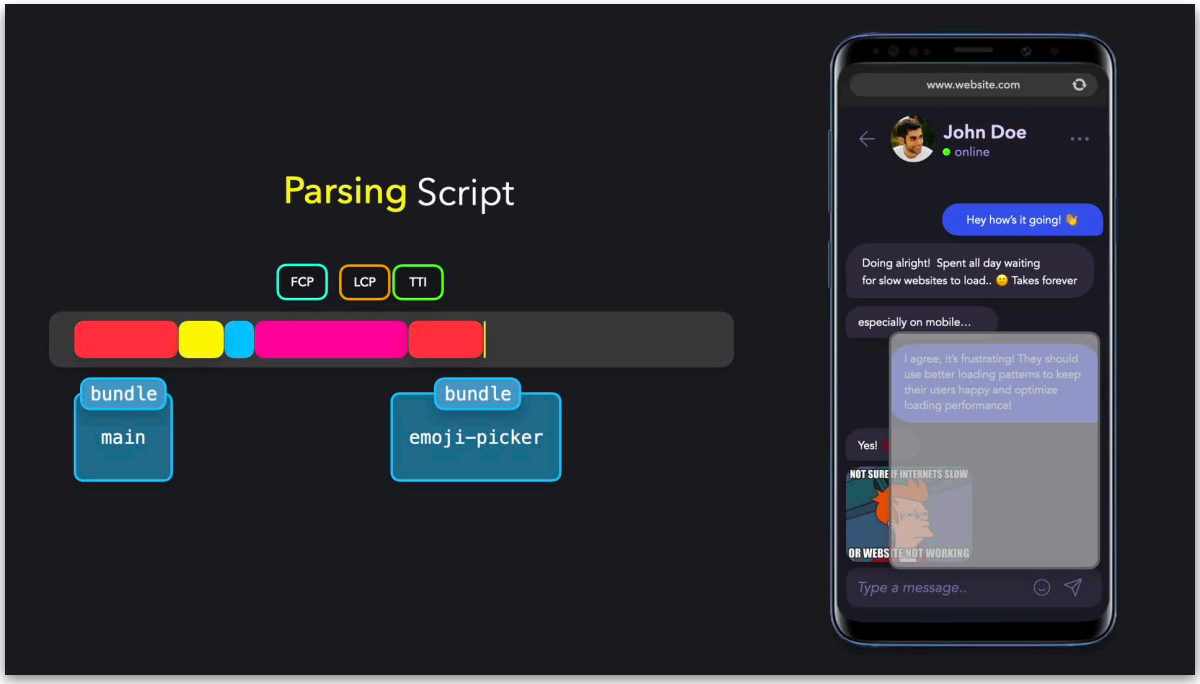
As we're not requesting all images instantly, we can reduce the initial loading time. We can do the same with components! In order to know whether components are currently in our viewport, we can use the `IntersectionObserver API`, or use libraries such as `react-lazyload` or `react-loadable-visibility` to quickly add import on visibility to our application.

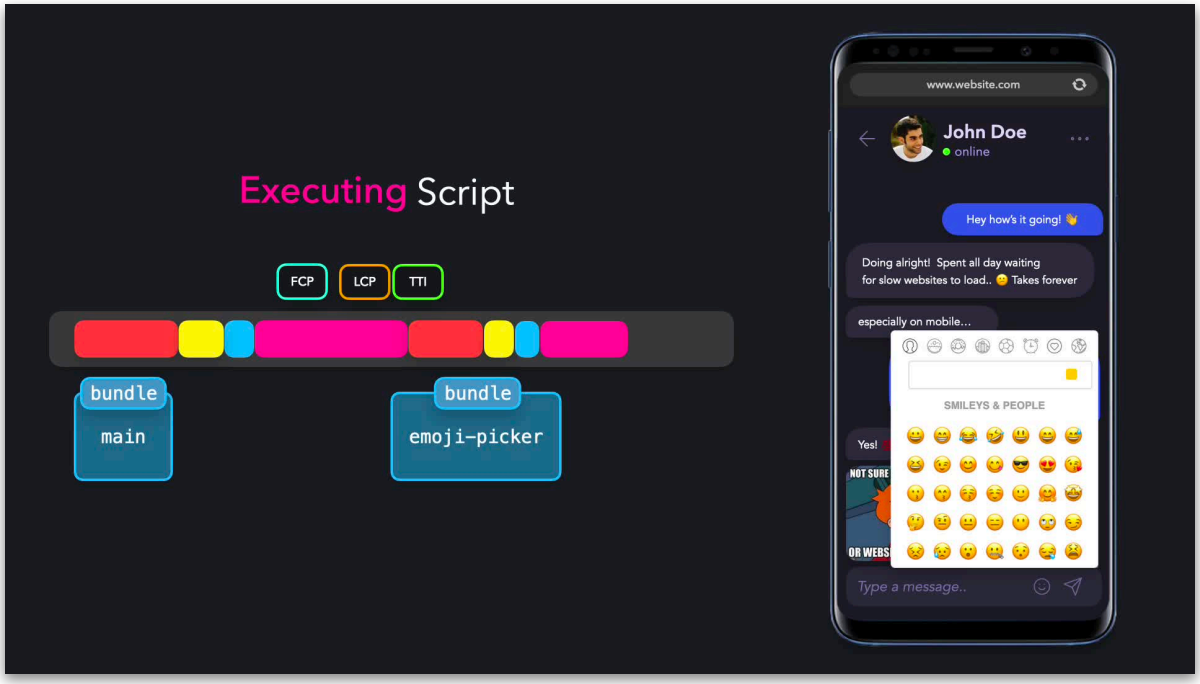
```
import React from "react";
import Send from "./icons/Send";
import Emoji from "./icons/Emoji";
import LoadableVisibility from "react-loadable-visibility/react-loadable";

const EmojiPicker = LoadableVisibility({
  loader: () => import("./EmojiPicker"),
  loading: <p id="loading">Loading</p>
});

const ChatInput = () => {
  const [pickerOpen, togglePicker] = React.useReducer(state => !state, false);

  return (
    <div className="chat-input-container">
      <input type="text" placeholder="Type a message..." />
      <Emoji onClick={togglePicker} />
      {pickerOpen && <EmojiPicker />}
      <Send />
    </div>
  );
};

console.log("ChatInput loading", Date.now());

export default ChatInput;
```

Whenever the `EmojiPicker` is rendered to the screen, after the user clicks on the Gif button, `react-loadable-visibility` detects that the `EmojiPicker` element should be visible on the screen. Only then, it will start importing the module while the user sees a loading component being rendered.


FCP
LCP
TTI
bundle
main
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..


Executing Script
FCP
LCP
TTI
bundle
main
bundle
emoji-picker
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
SMILEYS & PEOPLE
Yes!
Type a message..





The fallback component lets the user know that our application hasn't frozen: they simply need to wait a short while for the module to be loaded, parsed, compiled, and executed!

Import on Interaction

Load non-critical resources when a user interacts with UI requiring it

Your page may contain code or data for a component or resource that isn’t immediately necessary. For example, part of the user-interface a user doesn't see unless they click or scroll on parts of the page. This can apply to many kinds of first-party code you author, but this also applies to third-party widgets such as video players or chat widgets where you typically need to click a button to display the main interface.

Loading these resources eagerly (right away) can block the main thread if they are costly, pushing out how soon a user can interact with more critical parts of a page. This can impact interaction readiness metrics like **First Input Delay**, **Total Blocking Time** and **Time to Interactive.** Instead of loading these resources immediately, you can load them at a more opportune moment, such as:

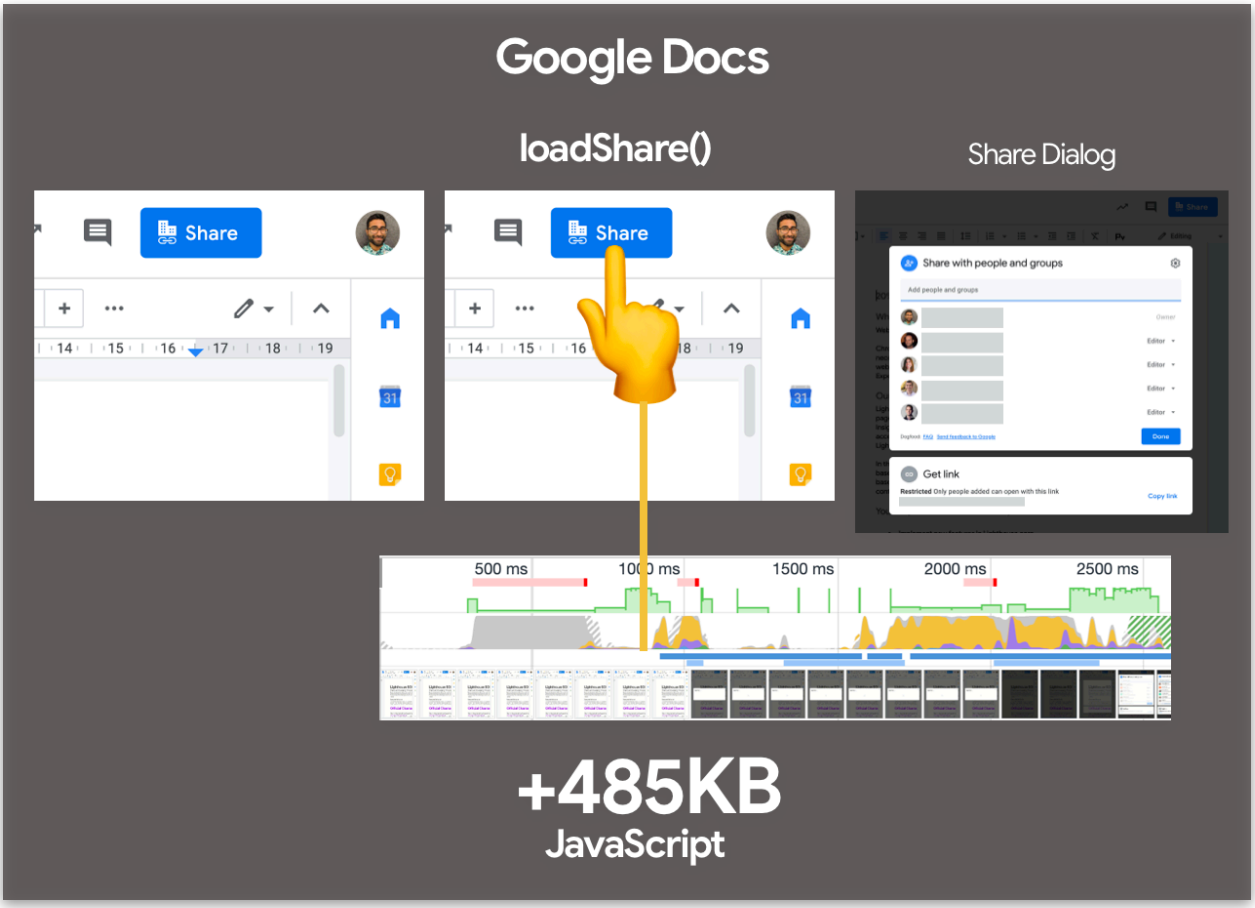
- When the user clicks to interact with that component for the first time
- The component scrolls into view
- Deferring load of that component until the browser is idle (via requestIdleCallback).

The different ways to load resources are, at a high-level:

- **Eager:** load resource right away (the normal way of loading scripts)
- **Lazy (Route-based):** load when a user navigates to a route or component
- **Lazy (On interaction):** load when the user clicks UI (e.g Show Chat)
- **Lazy (In viewport):** load when the user scrolls towards the component
- **Prefetch:** load prior to needed, but after critical resources are loaded
- **Preload:** eagerly, with a greater level of urgency

Import on interaction for first-party code should only be done if you're unable to prefetch resources prior to interaction. The pattern is however very relevant for third-party code, where you generally want to defer it if non-critical to a later point in time. This can be achieved in many ways (defer until interaction, until the browser is idle or using other heuristics).

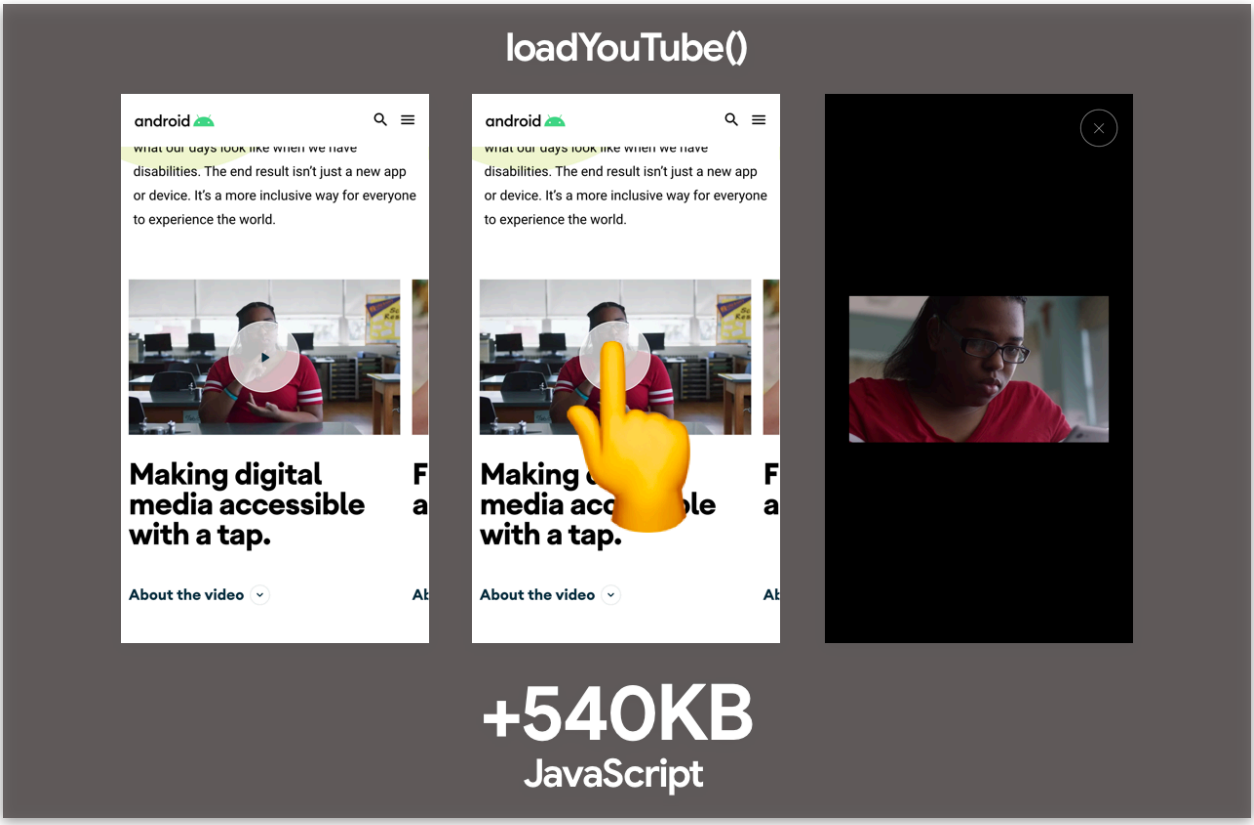
Lazily importing feature code on interaction is a pattern used in many contexts we will cover in this post. One place you may have used it before is Google Docs, where they save loading 500KB of script for the share feature by deferring its load until user-interaction.



Another place where import-on-interaction can be a good fit is loading third-party widgets.

"Fake" loading third-party UI with a facade

You might be importing a third-party script and have less control over what it renders or when it loads code. One option for implementing load-on-interaction is straight-forward: use a facade. A facade is a simple "preview" or "placeholder" for a more costly component where you simulate the basic experience, such as with an image or screenshot. It's terminology we've been using for this idea on the Lighthouse team.

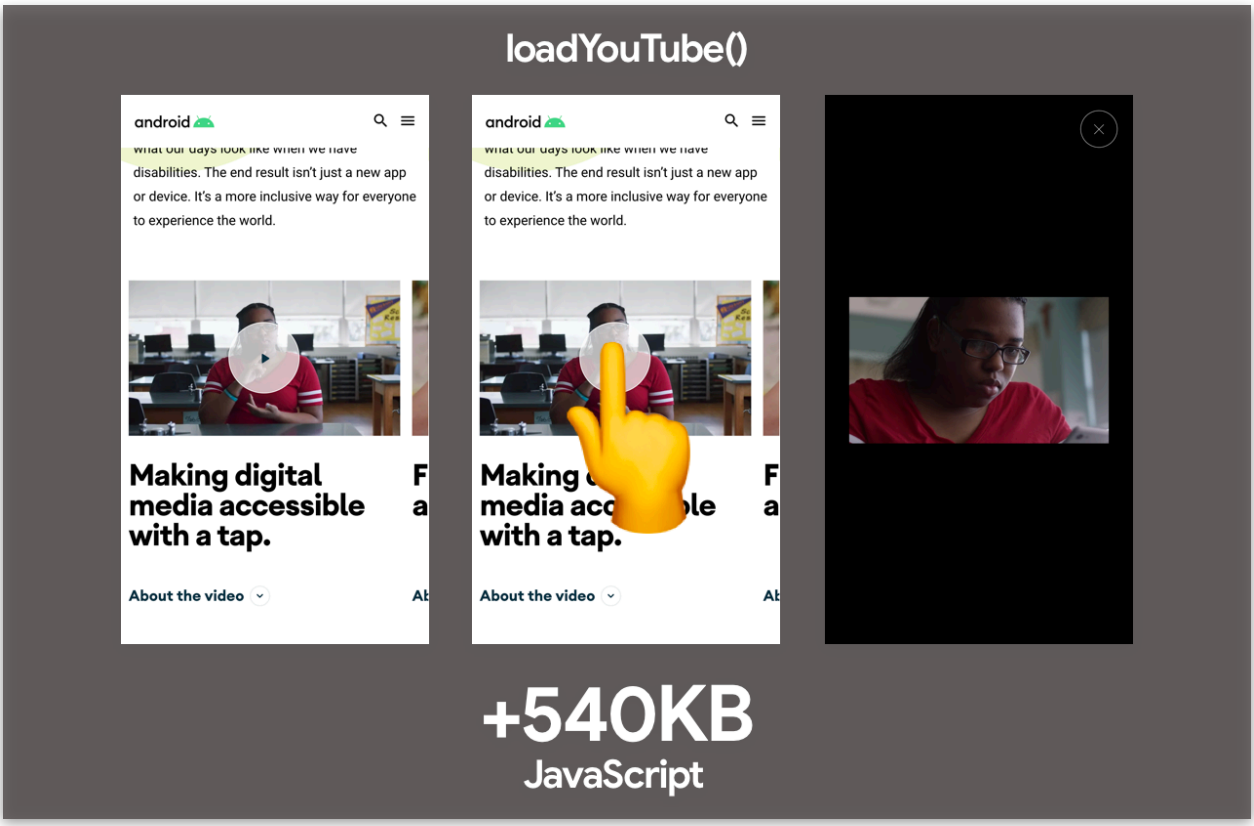


When a user clicks on the "preview" (the facade), the code for the resource is loaded. This limits users needing to pay the experience cost for a feature if they're not going to use it. Similarly, facades can preconnect to necessary resources on hover.

Third-party resources are often added to pages without full consideration for how they fit into the overall loading of a site. Synchronously-loaded third-party scripts block the browser parser and can delay hydration. If possible, 3P script should be loaded with async/defer (or other approaches) to ensure 1P scripts aren't starved of network bandwidth. Unless they are critical, they can be a good candidate for shifting to deferred late-loading using patterns like import-on-interaction.

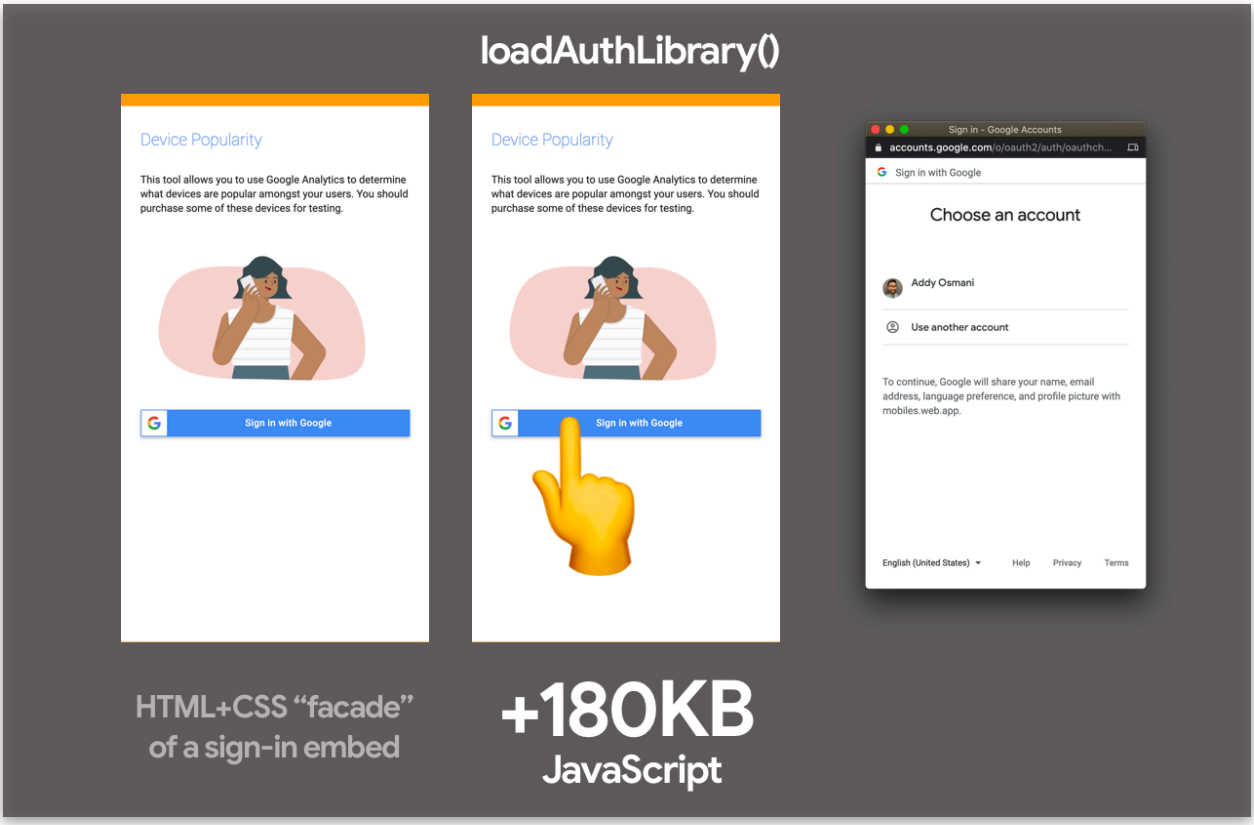
Video Player Embeds

A good example of a "facade" is the YouTube Lite Embed by Paul Irish. This provides a Custom Element which takes a YouTube Video ID and presents a minimal thumbnail and play button. Clicking the element dynamically loads the full YouTube embed code, meaning users who never click play don’t pay the cost of fetching and processing it.



Authentication

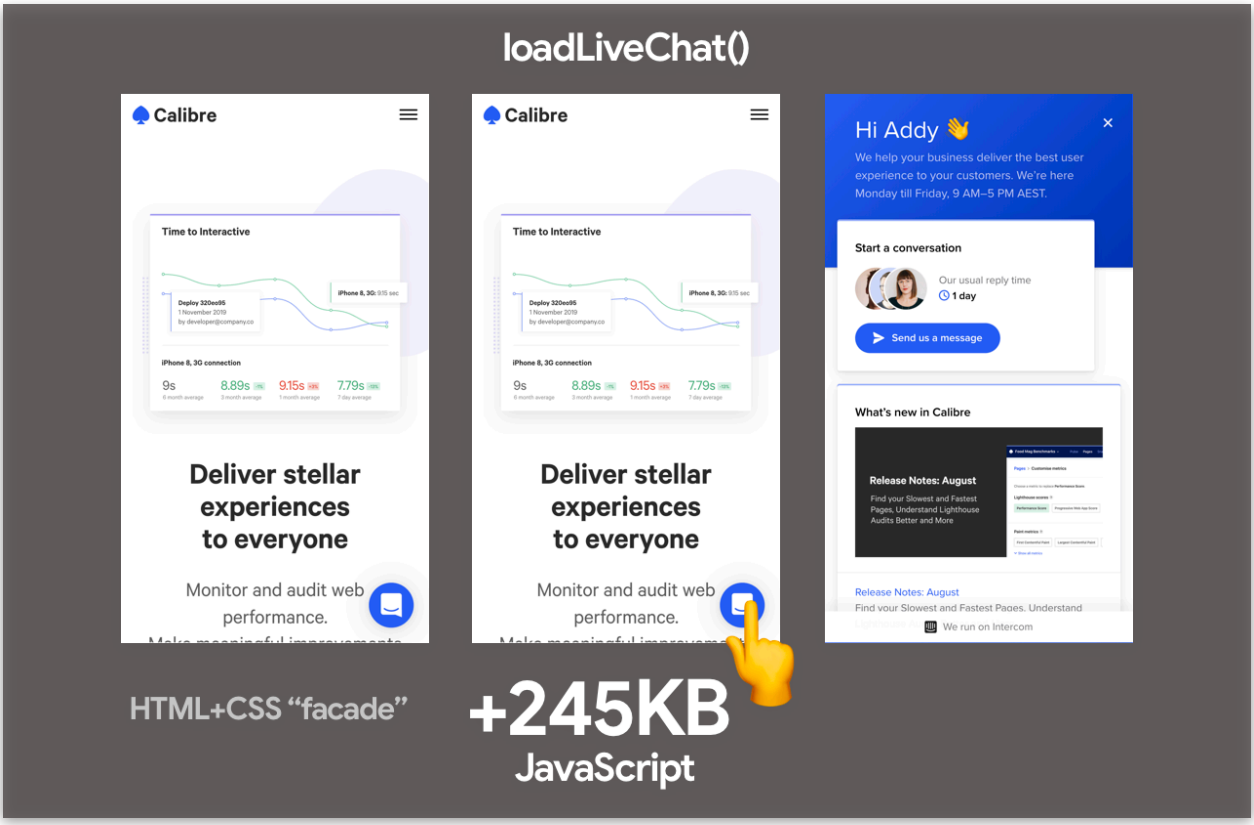
Apps may need to support authentication with a service via a client-side JavaScript SDK. These can occasionally be large with heavy JS execution costs and one might rather not eagerly load them up front if a user isn't going to login. Instead, dynamically import authentication libraries when a user clicks on a "Login" button, keeping the main thread more free during initial load.



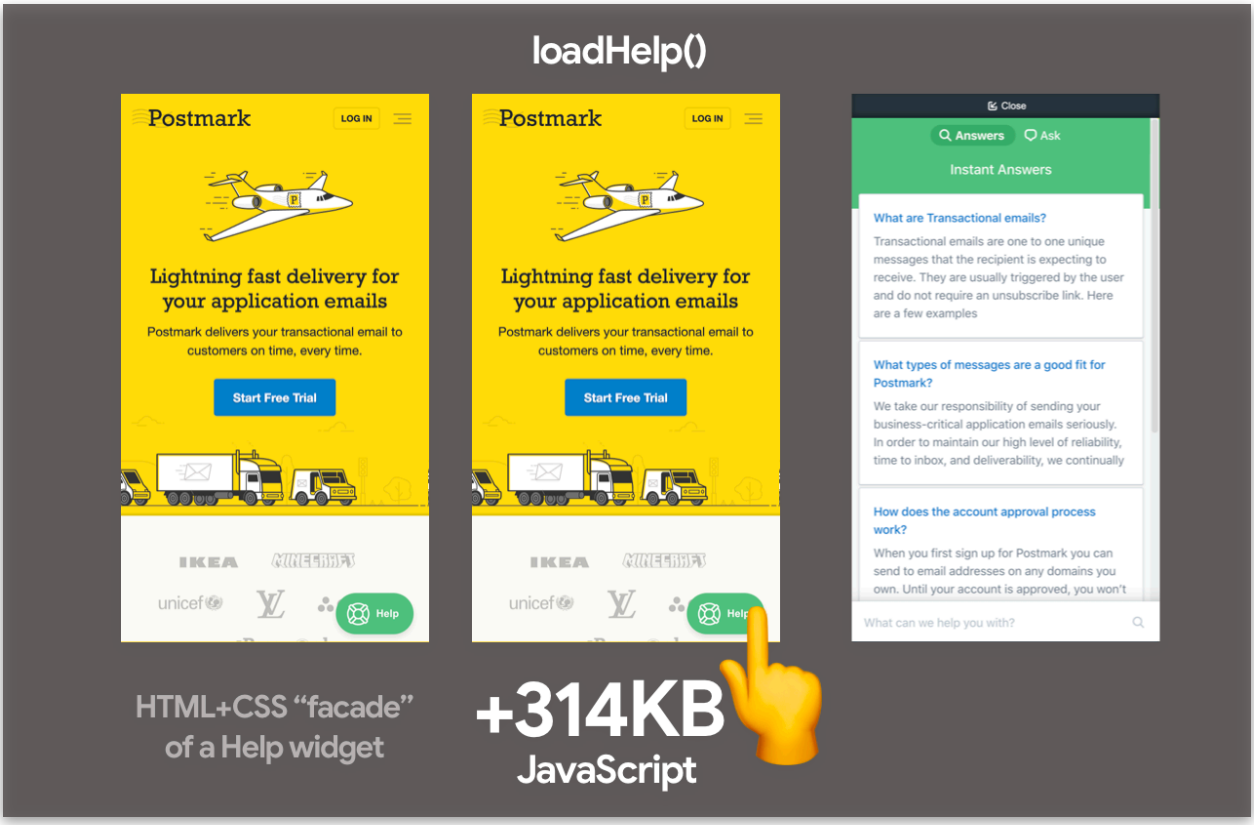
Chat Widgets

Calibre app improved performance of their Intercom-based live chat by 30% through usage of a similar facade approach. They implemented a "fake"

fast loading live chat button using just CSS and HTML, which when clicked would load their Intercom bundles.



Postmark noted that their Help chat widget was always eagerly loaded, even though it was only occasionally used by customers. The widget would pull in 314KB of script, more than their whole home page. To improve user-experience, they replaced the widget with a fake replica using HTML and CSS, loading the real-thing on click. This change reduced Time to Interactive from 7.7s to 3.7s.



Vanilla JavaScript

In JavaScript, dynamic import() enables lazy-loading modules and returns a promise and can be quite powerful when applied correctly. Below is an example where dynamic import is used in a button event listener to import the lodash.sortby module and then use it.

```
const btn = document.querySelector('button');

btn.addEventListener('click', e => {
  e.preventDefault();
  import('lodash.sortby')
    .then(module => module.default)
    .then(sortInput()) // use the imported dependency
    .catch(err => { console.log(err) });
});
```

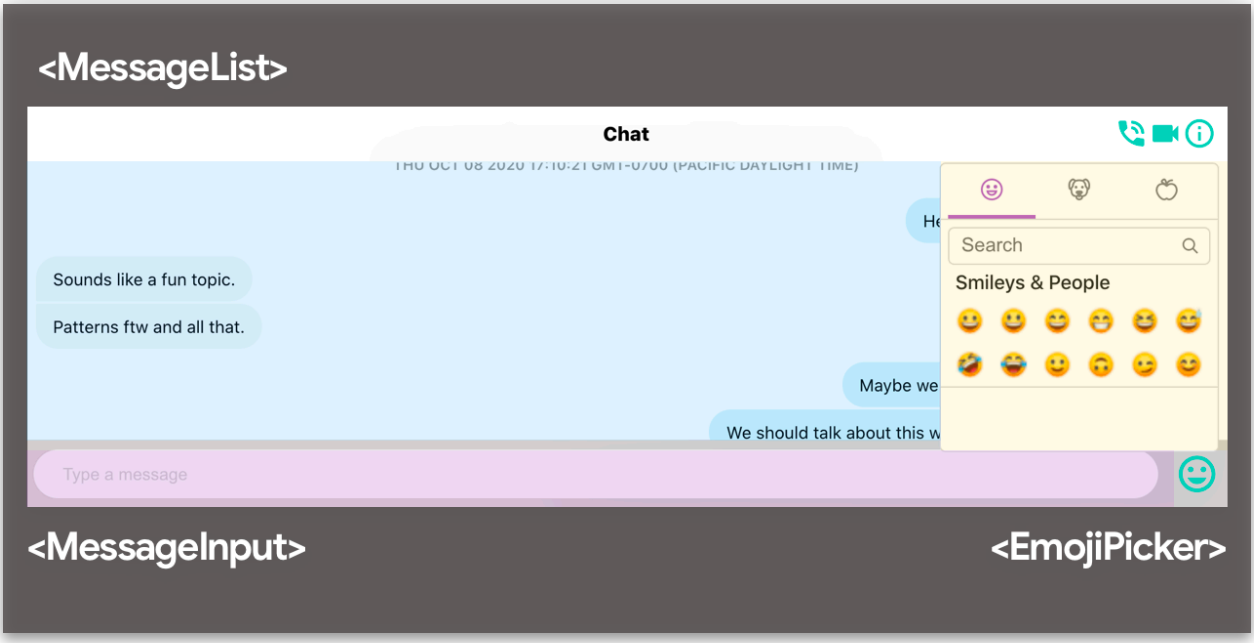
Prior to dynamic import or for use-cases it doesn't fit as well, dynamically injecting scripts into the page using a Promise-based script loader was also an option.

```
const loginBtn = document.querySelector('#login');

loginBtn.addEventListener('click', () => {
  const loader = new scriptLoader();
  loader.load([
      '//apis.google.com/js/client:platform.js?onload=showLoginScreen'
  ]).then(({length}) => {
      console.log(`${length} scripts loaded!`);
  });
});
```

React

Let's imagine we have a Chat application which has a `MessageList`, `MessageInput` and an `EmojiPicker` component (powered by `emoji-mart`, which is 98KB minified and gzipped). It can be common to eagerly load all of these components on initial page-load.

```
import MessageList from './MessageList';
import MessageInput from './MessageInput';
import EmojiPicker from './EmojiPicker';

const Channel = () => {
  ...
  return (
    <div>
      <MessageList />
      <MessageInput />
      {emojiPickerOpen && <EmojiPicker />}
    </div>
  );
};
```



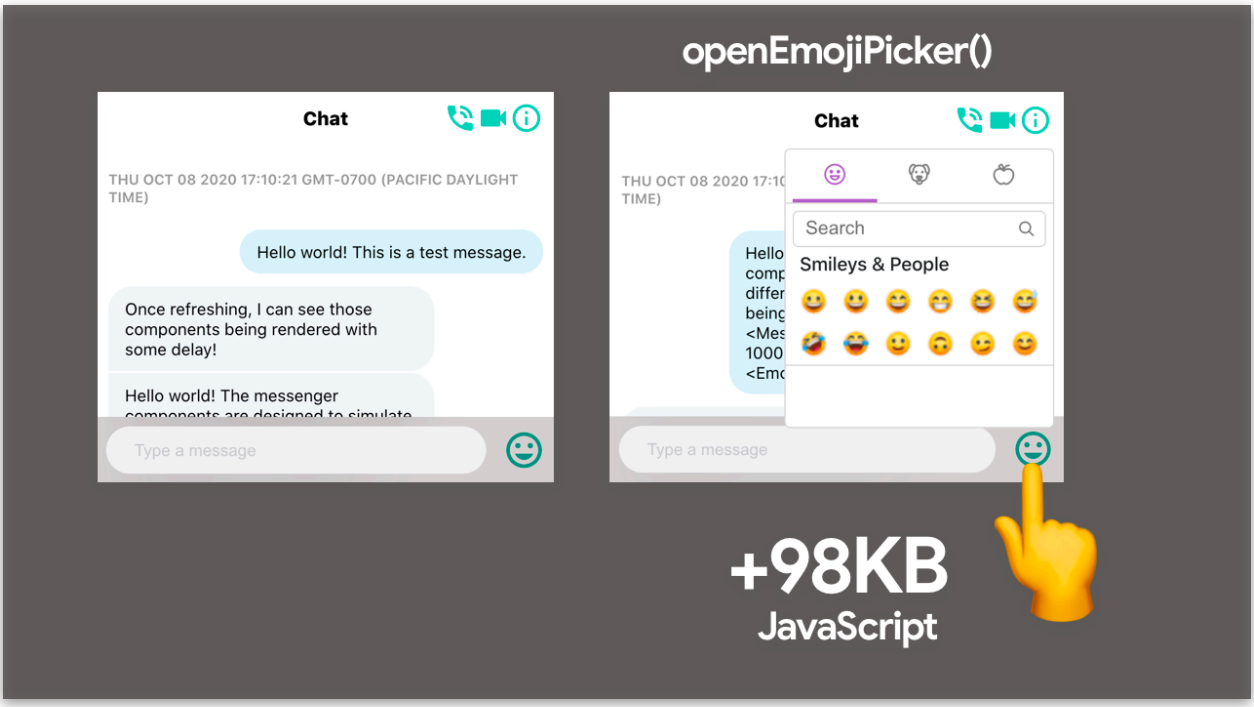
Breaking the loading of this work up is relatively straight-forward with code-splitting. The `React.lazy` method makes it easy to code-split a React application on a component level using dynamic imports.

The `React.lazy` function provides a built-in way to separate components in an application into separate chunks of JavaScript with very little legwork. You can then take care of loading states when you couple it with the `Suspense` component.

```
import React, { lazy, Suspense } from 'react';
import MessageList from './MessageList';
import MessageInput from './MessageInput';

const EmojiPicker = lazy(
  () => import('./EmojiPicker')
);

const Channel = () => {
  ...
  return (
    <div>
      <MessageList />
      <MessageInput />
      {emojiPickerOpen && (
        <Suspense fallback={<div>Loading...</div>}>
          <EmojiPicker />
        </Suspense>
      )}
    </div>
  );
};
```

We can extend this idea to only import code for the `EmojiPicker` component when the Emoji icon is clicked in a `MessageInput`, rather than eagerly when the application initially loads:

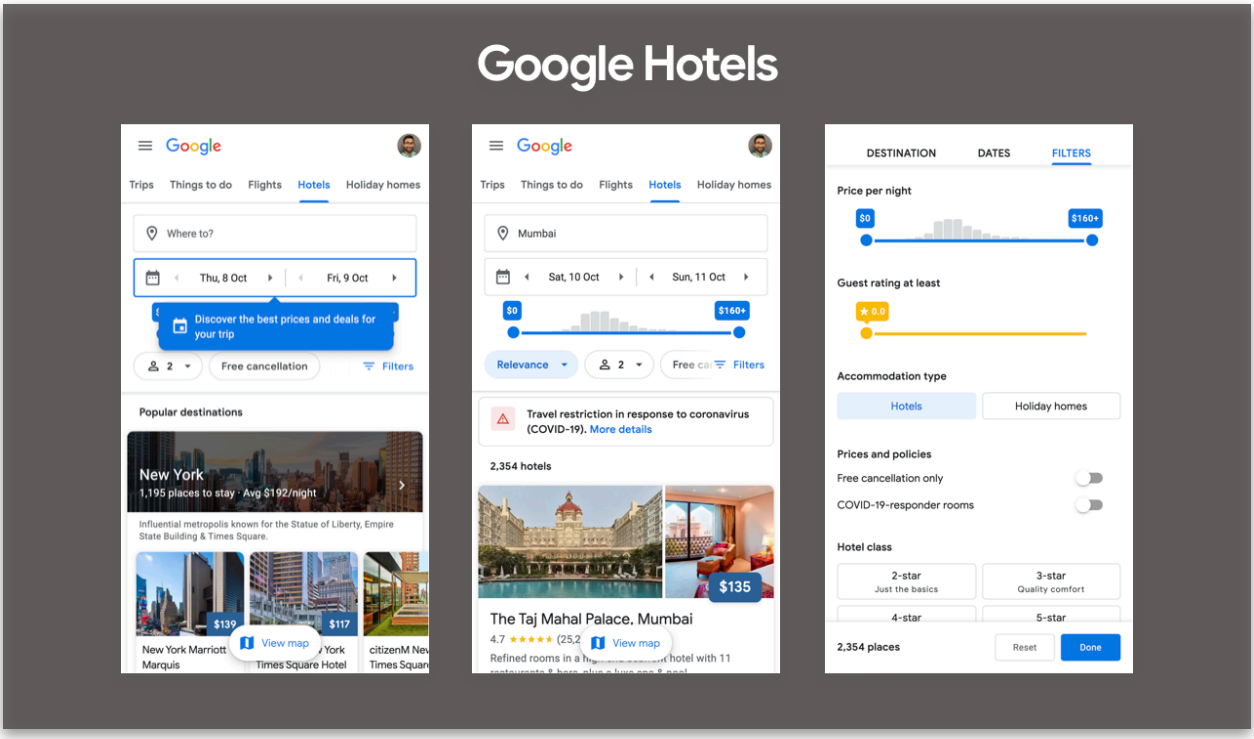
```
import React, { useState, createElement } from 'react';
import MessageList from './MessageList';
import MessageInput from './MessageInput';
import ErrorBoundary from './ErrorBoundary';

const Channel = () => {
  const [emojiPickerEl, setEmojiPickerEl] = useState(null);

  const openEmojiPicker = () => {
    import(/* webpackChunkName: "emoji-picker" */ './EmojiPicker')
      .then(module => module.default)
      .then(emojiPicker => {
        setEmojiPickerEl(createElement(emojiPicker));
      });
  };

  const closeEmojiPickerHandler = () => {
    setEmojiPickerEl(null);
  };

  return (
    <ErrorBoundary>
      <div>
        <MessageList />
        <MessageInput onClick={openEmojiPicker} />
        {emojiPickerEl}
      </div>
    </ErrorBoundary>
  );
};
```



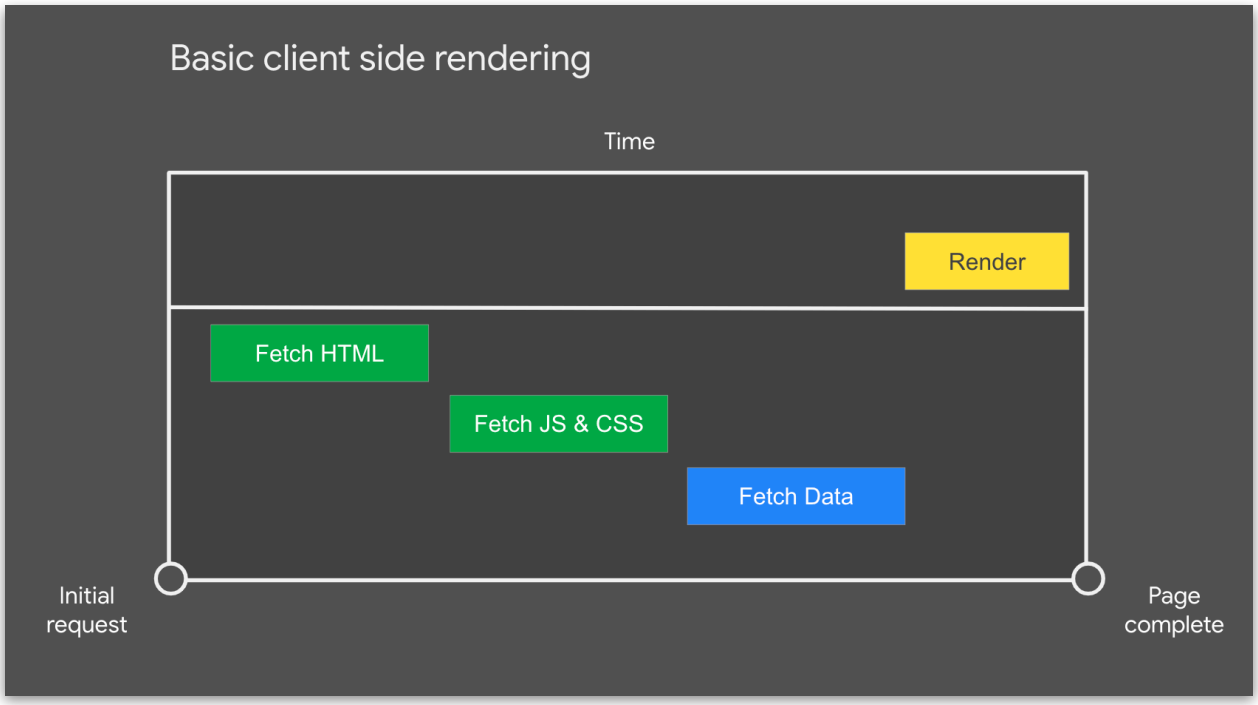
Import-on-interaction for first-party code as part of progressive loading

Loading code on interaction also happens to be a key part of how Google handles progressive loading in large applications like Flights and Photos. To illustrate this, let's take a look at an example previously presented by Shubhie Panicker.

Imagine a user is planning a trip to Mumbai, India and they visit Google Hotels to look at prices. All of the resources needed for this interaction could be loaded eagerly upfront, but if a user hasn't selected any destination, the HTML/CSS/JS required for the map would be unnecessary.



In the simplest download scenario, imagine Google Hotels is using naive client-side rendering (CSR). All the code would be downloaded and processed upfront: HTML, followed by JS, CSS and then fetching the data, only to render once we have everything. However, this leaves the user waiting a long time with nothing displayed on-screen. A big chunk of the JavaScript and CSS may be unnecessary.



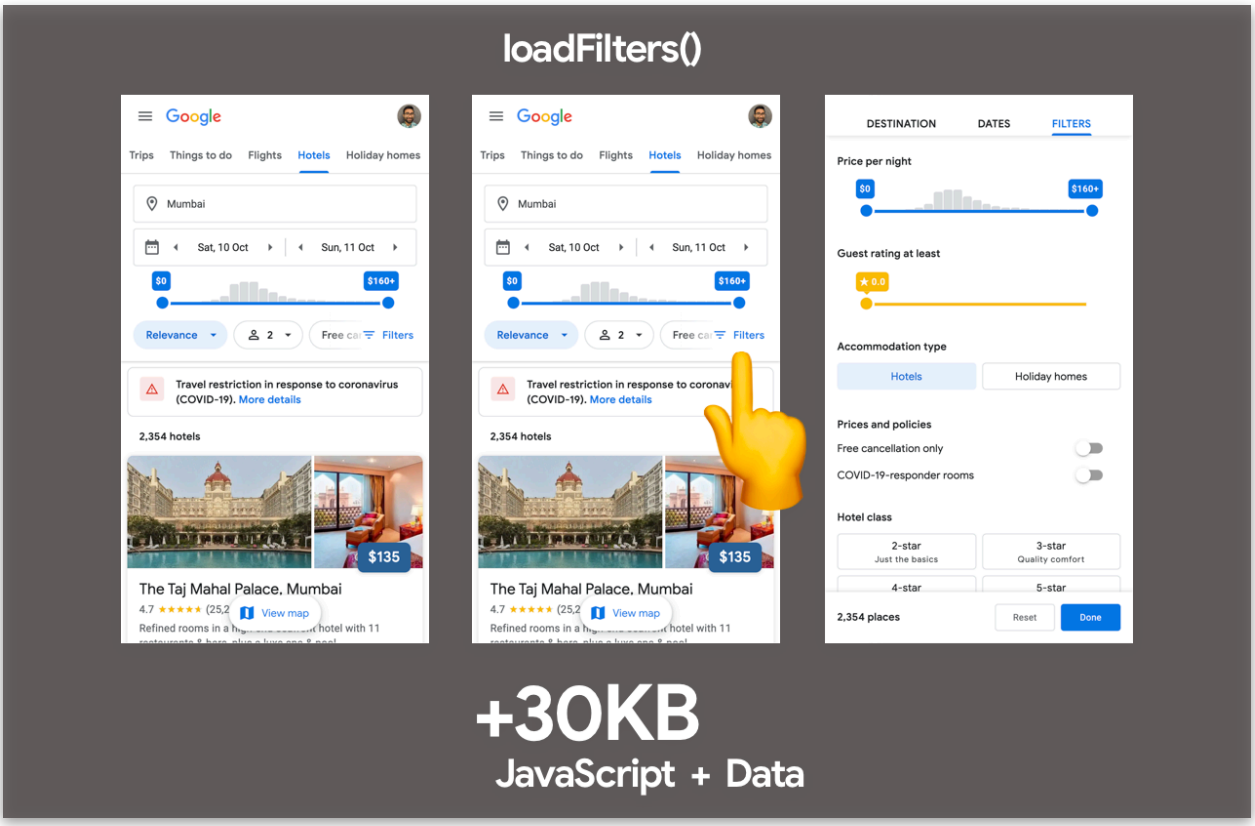
Next, imagine this experience moved to server-side rendering (SSR). We would allow the user to get a visually complete page sooner, which is great, however it wouldn't be interactive until the data is fetched from the server and the client framework completes hydration.

SSR can be an improvement, but the user may have an uncanny valley experience where the page looks ready, but they are unable to tap on anything. Sometimes this is referred to as rage clicks as users tend to click over and over again repeatedly in frustration.

Returning to the Google Hotels search example, if we zoom in to the UI a little we can see that when a user clicks on "more filters" to find exactly the right hotel, the code required for that component is downloaded.

Only very minimal code is downloaded initially and beyond this, user interaction dictates which code is sent down when.

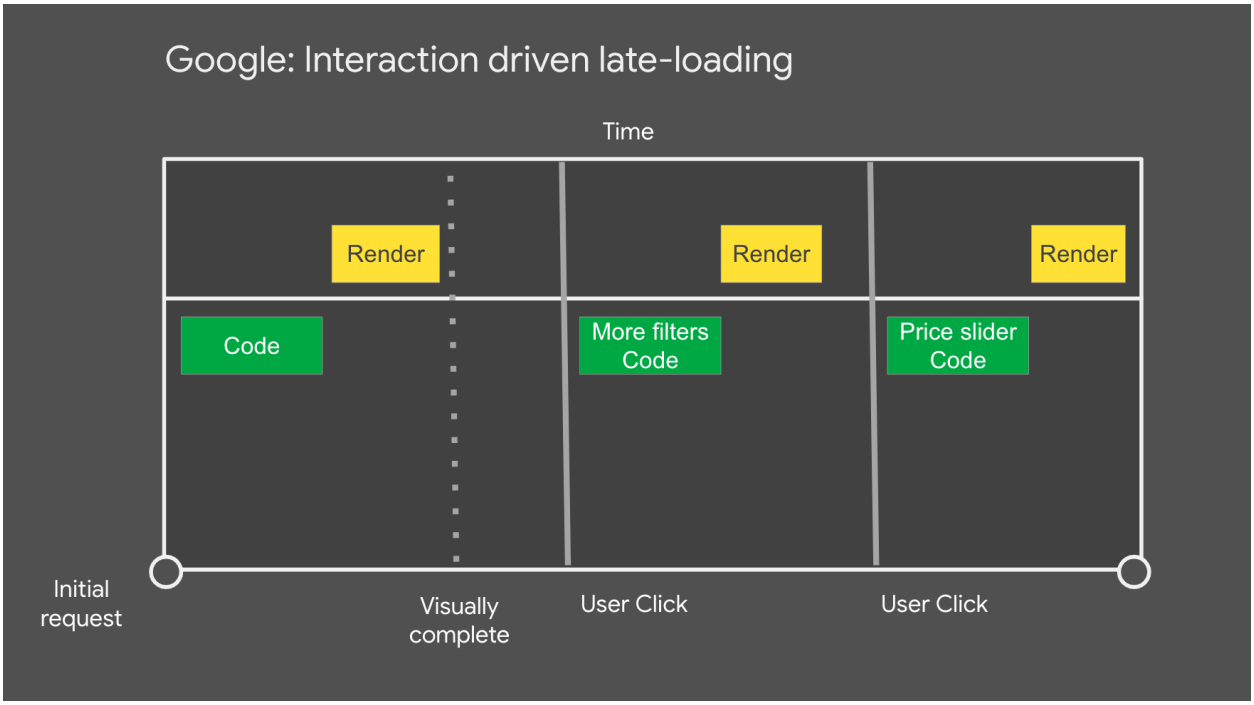
Let’s take a closer look at this loading scenario.



There are a number of important aspects to interaction-driven late-loading:

First, we download the minimal code initially so the page is visually complete quickly.

Next, as the user starts interacting with the page we use those interactions to determine which other code to load. For example loading the code for the "more filters" component. This means code for many features on the page are never sent down to the browser, as the user didn’t need to use them.



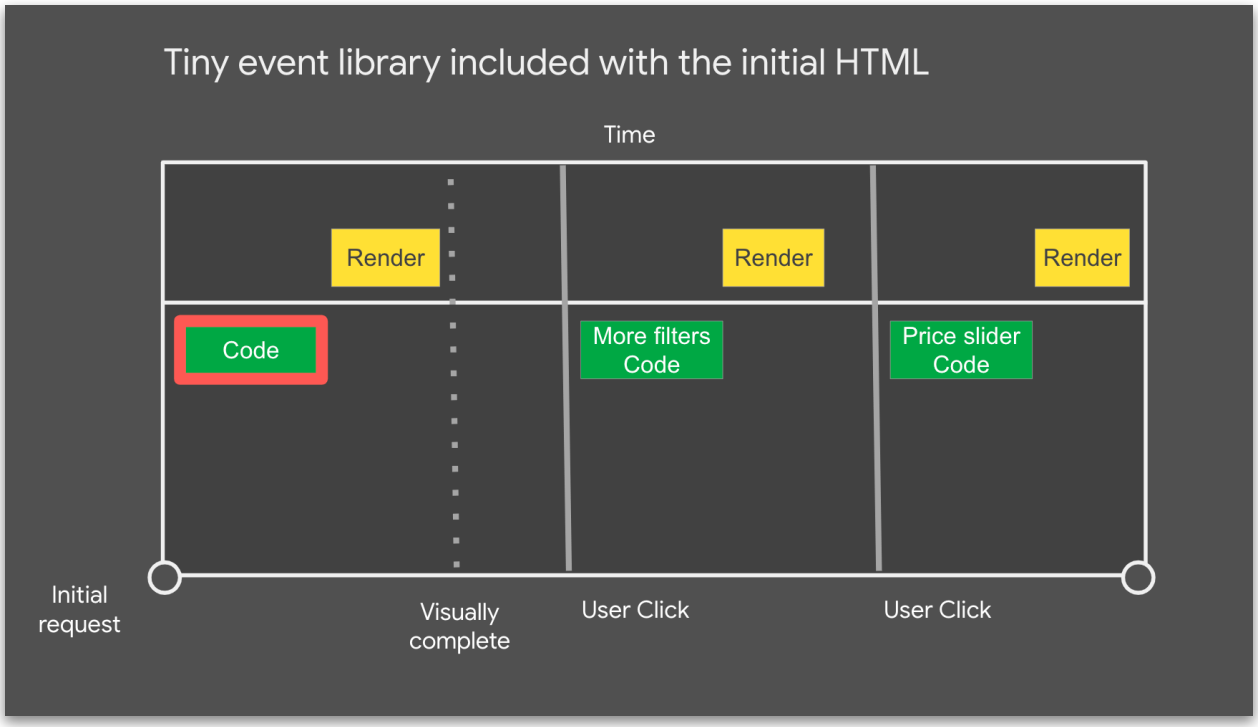
How do we avoid losing early clicks?

In the framework stack used by these Google teams, we can track clicks early because the first chunk of HTML includes a small event library (JSAction) which tracks all clicks before the framework is bootstrapped. The events are used for two things:

- Triggering download of component code based on user interactions
- Replaying user interactions when the framework finishes bootstrapping

Other potential heuristics one could use include, loading component code:

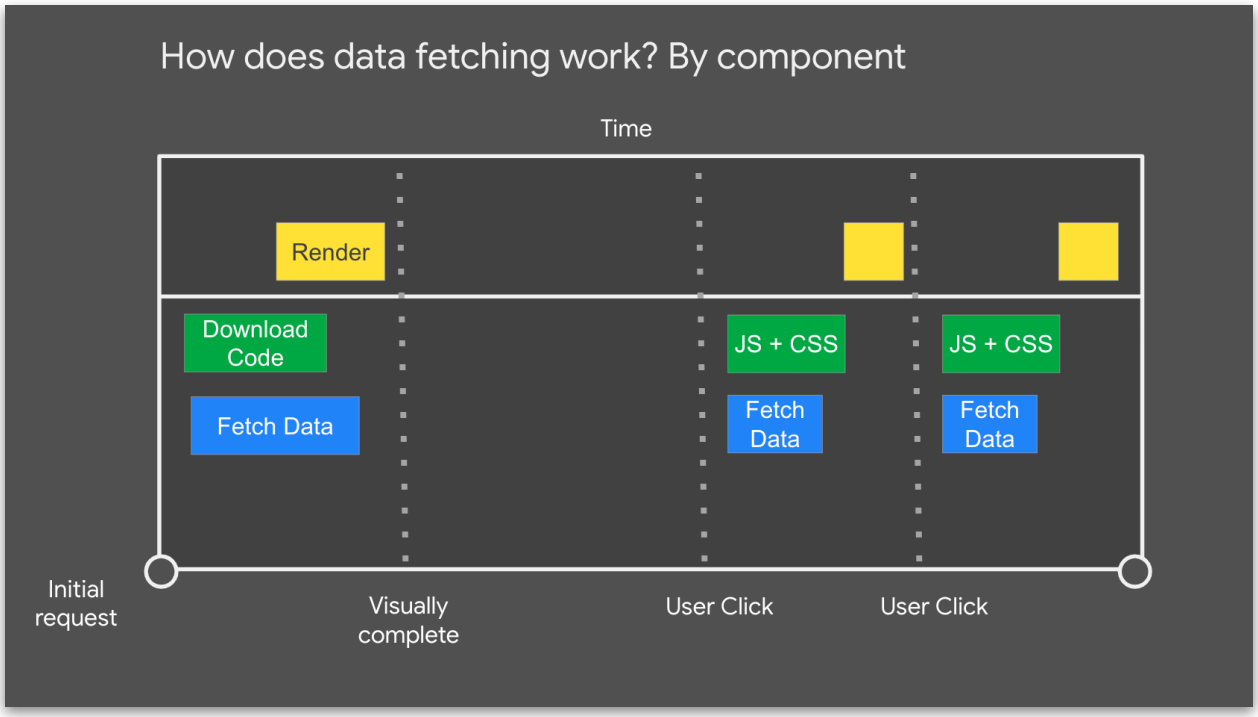
- A period after idle time
- On user mouse hover over the relevant UI/button/call to action
- Based on a sliding scale of eagerness based on browser signals (e.g network speed, Data Saver mode etc).



What about data?

The initial data which is used to render the page is included in the initial page's SSR HTML and streamed. Data that is late loaded is downloaded based on user interactions as we know what component it goes with.

This completes the import-on-interaction picture with data-fetching working similar to how CSS and JS function. As the component is aware of what code and data it needs, all of its resources are never more than a request away.



This functions as we create a graph of components and their dependencies during build time. The web application is able to refer to this graph at any point and quickly fetch the resources (code and data) needed for any component. It also means we code-split based on the component rather than the route.

For a walkthrough of the above example, see Elevating the Web Platform with the JavaScript Community.

Trade-offs

Shifting costly work closer to user-interaction can optimize how quickly pages initially load, however the technique is not without trade-offs.

What happens if it takes a long time to load a script after the user clicks?

In the Google Hotels example, small granular chunks minimize the chance a user is going to wait long for code and data to fetch and execute. In some of the other cases, a large dependency may indeed introduce this concern on slower networks.

One way to reduce the chance of this happening is to better break-up the loading of, or prefetch these resources after critical content in the page is done loading. I'd encourage measuring the impact of this to determine how much it's a real application in your apps.

What about lack of functionality before user interaction?

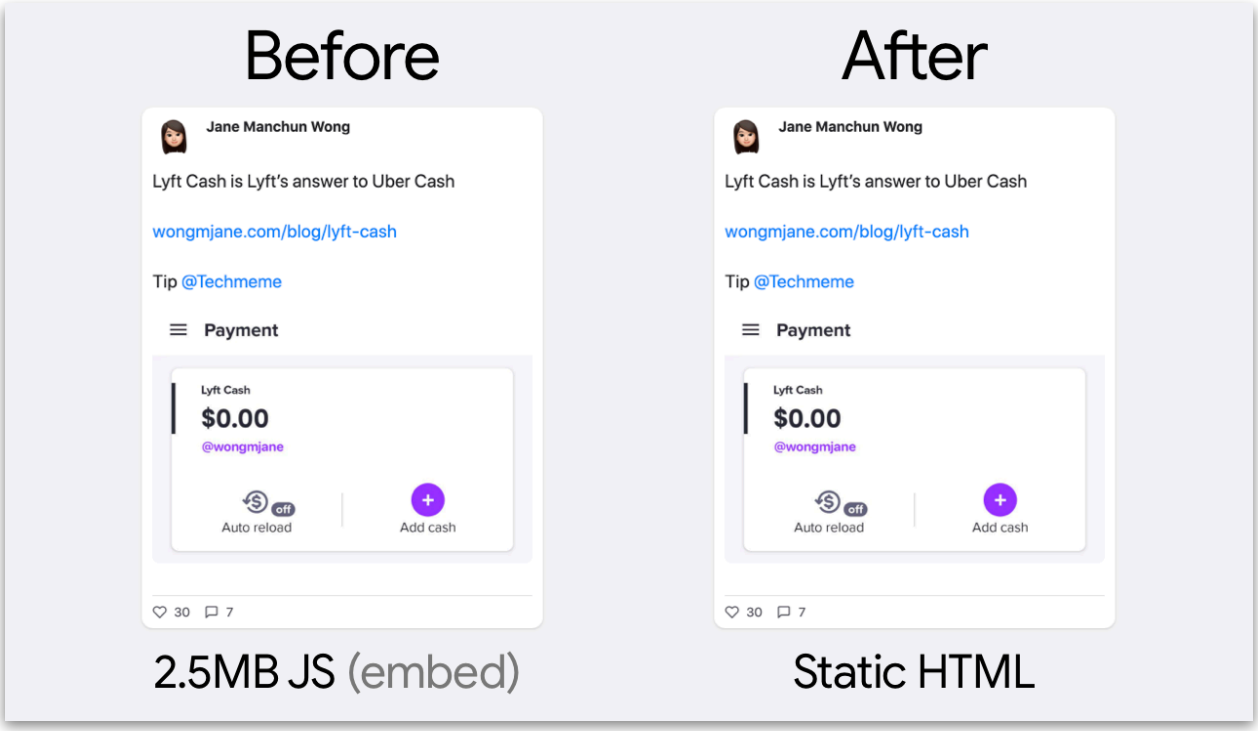
Another trade-off with facades is a lack of functionality prior to user interaction. An embedded video player for example will not be able to autoplay media. If such functionality is key, you might consider alternative approaches to loading the resources, such as lazy-loading these third-party iframes on the user scrolling them into view rather than deferring load until interaction.

Replacing interactive embeds with a static variant

We have discussed the import-on-interaction pattern and progressive loading, but what about going entirely static for the embeds use-case?.

The final rendered content from an embed may be needed immediately in some cases e.g a social media post that is visible in the initial viewport. This can also introduce its own challenges when the embed brings in 2-3MB of JavaScript. Because the embed content is needed right away, lazy-loading and facades may be less applicable.

If optimizing for performance, it's possible to entirely replace an embed with a static variant that looks similar, linking out to a more interactive version (e.g the original social media post). At build time, the data for the embed can be pulled in and transformed into a static HTML version.



This is the approach *@wongmjane* leveraged on their blog for one type of social media embed, improving both page load performance and removing the Cumulative Layout Shift experienced due to the embed code enhancing the fallback text, causing layout shifts.

While static replacements can be good for performance, they do often require doing something custom so keep this in mind when evaluating your options.

Conclusions

First-party JavaScript often impacts the interaction readiness of modern pages on the web, but it can often get delayed on the network behind non-critical JS from either first or third-party sources that keep the main thread busy.

In general, avoid synchronous third-party scripts in the document head and aim to load non-blocking third-party scripts after first-party JS has finished loading. Patterns like import-on-interaction give us a way to defer the loading of non-critical resources to a point when a user is much more likely to need the UI they power.

With special thanks to Shubhie Panicker, Connor Clark, Patrick Hulce, Anton Karlovskiy and Adam Raine for their input.

Route Based Splitting

Dynamically load components based on the current route

We can request resources that are only needed for specific routes, by adding route-based splitting. By combining React Suspense with libraries such as `react-router`, we can dynamically load components based on the current route.

```
import React, { lazy, Suspense } from "react";
import { render } from "react-dom";
import { Switch, Route, BrowserRouter as Router } from "react-router-dom";

const App = lazy(() => import(/* webpackChunkName: "home" */ "./App"));
const Overview = lazy(() =>
  import(/* webpackChunkName: "overview" */ "./Overview")
);
const Settings = lazy(() =>
  import(/* webpackChunkName: "settings" */ "./Settings")
);

render(
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/">
          <App />
        </Route>
        <Route path="/overview">
          <Overview />
        </Route>
        <Route path="/settings">
          <Settings />
        </Route>
      </Switch>
    </Suspense>
  </Router>,
  document.getElementById("root")
);
```

By lazily loading the components per route, we're only requesting the bundle that contains the code that's necessary for the current route. Since most people are used to the fact that there may be some loading time during a redirect, it's the perfect place to lazily load components!

Bundle Splitting

Split your code into small, reusable pieces

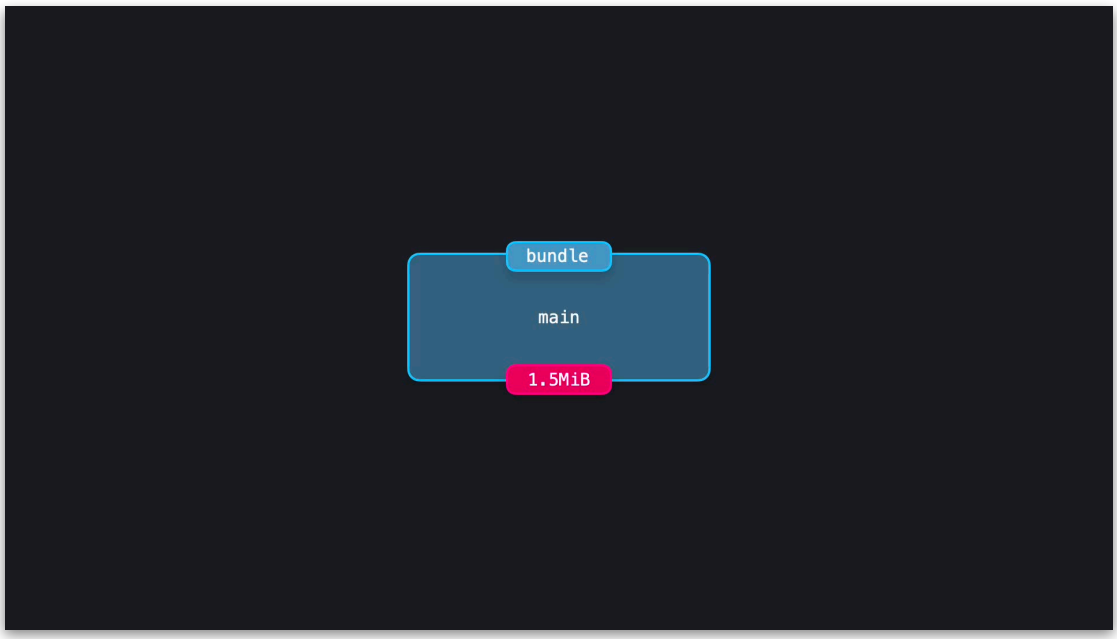
When building a modern web application, bundlers such as Webpack or Rollup take an application's source code, and bundle this together into one or more bundles. When a user visits a website, the bundle is requested and loaded in order to display the data to the user's screen.

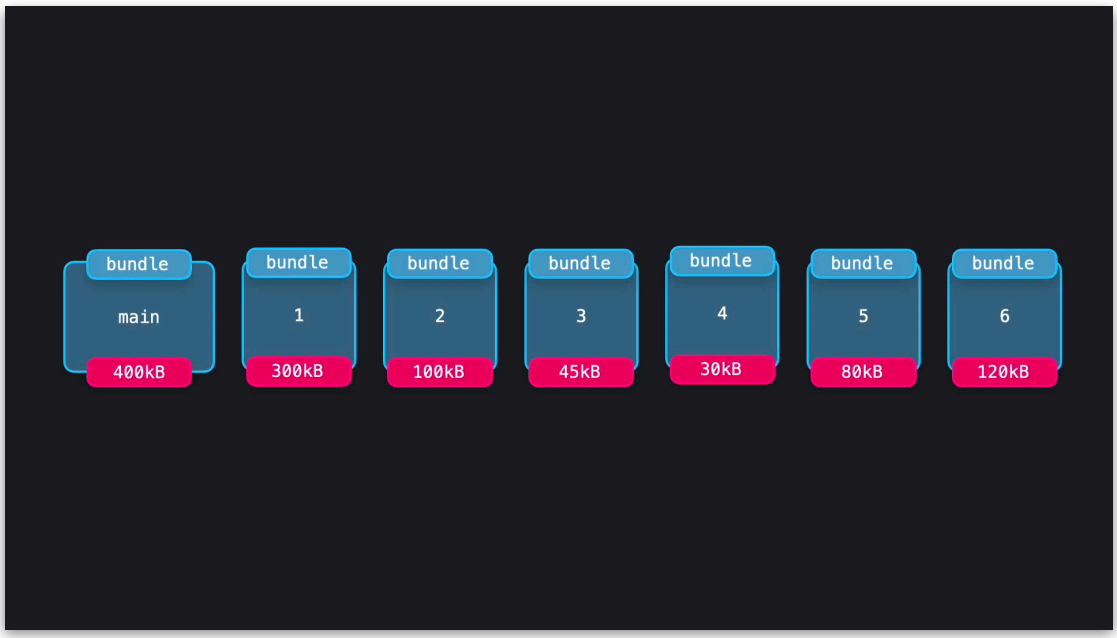
JavaScript engines such as V8 are able to parse and compile data that's been requested by the user as it's being loaded. Although modern browsers have evolved to parse and compile the code as quickly and performant as possible, the developer is still in charge of optimizing two steps in the process: the loading time and execution time of the requested data. We want to make sure we're keeping the execution time as short as possible to prevent blocking the main thread

Even though modern browsers are able to stream the bundle as it arrives, it can still take a significant time before the first pixel is painted on the user's device. The bigger the bundle the longer it can take before the engine reaches the line on which the first rendering call has been made. Until that time, the user has to stare at a blank screen for quite some time, which can be.. highly frustrating!

We want to display data to the user as quickly as possible. A larger bundle leads to an increased amount of loading time, processing time, and execution time. It would be great if we could reduce the size of this bundle, in order to speed things up.

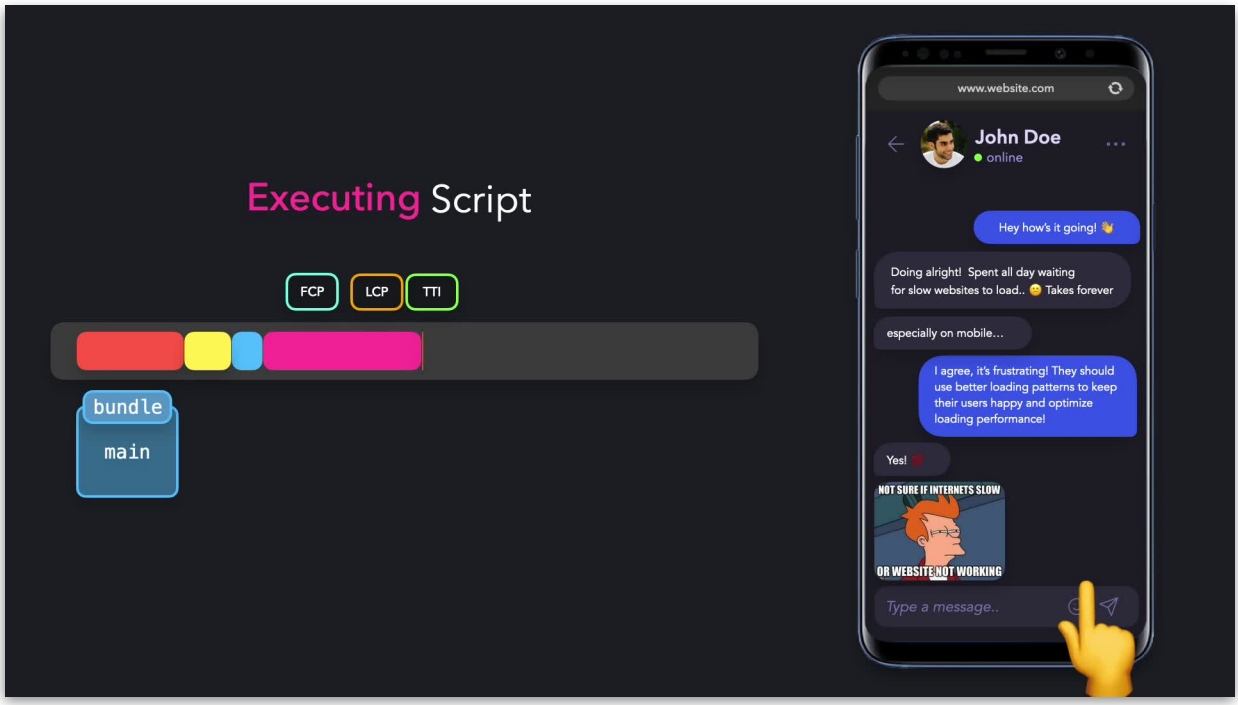
Instead of requesting one giant bundle that contains unnecessary code, we can split the bundle into multiple smaller bundles!

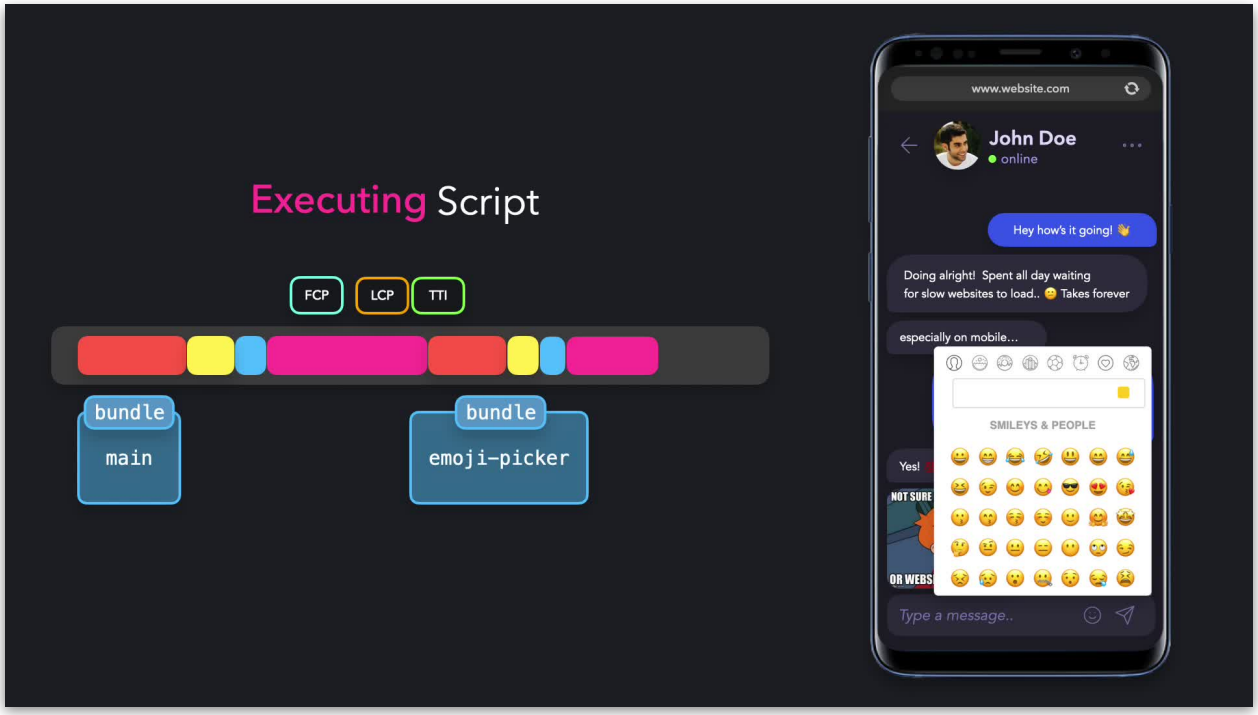




By bundle-splitting the application, we can reduce the time it takes to load, process and execute a bundle! By reducing the loading and execution time, we can reduce the time it takes before the first content has been painted on the user's screen, the First Contentful Paint, and the time it takes before the largest component has been rendered to the screen, the Largest Contentful Paint.

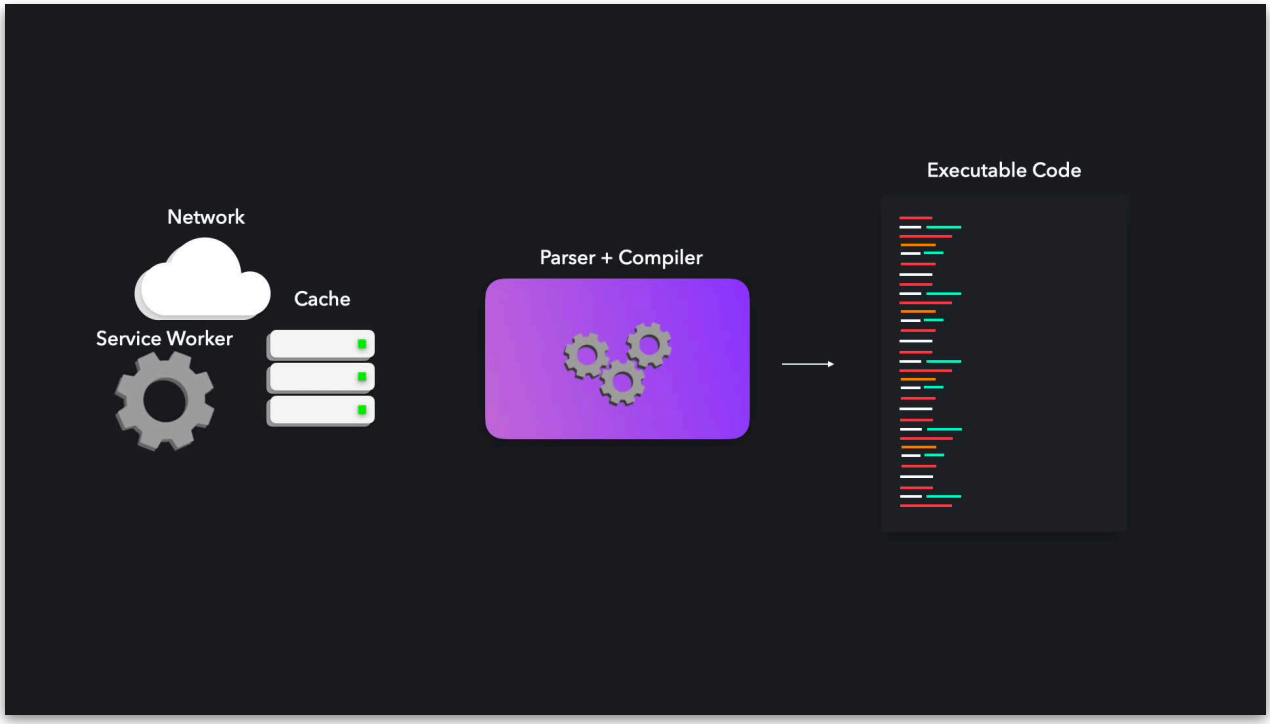
Although being able to see data on our screen is great, we don't just want to see the content. In order to have a fully functioning application, we want users to be able to interact with it as well! The UI only becomes interactive after the bundle has been loaded and executed. The time it takes before all content has been painted to the screen and has been made interactive, is called the Time To Interactive.

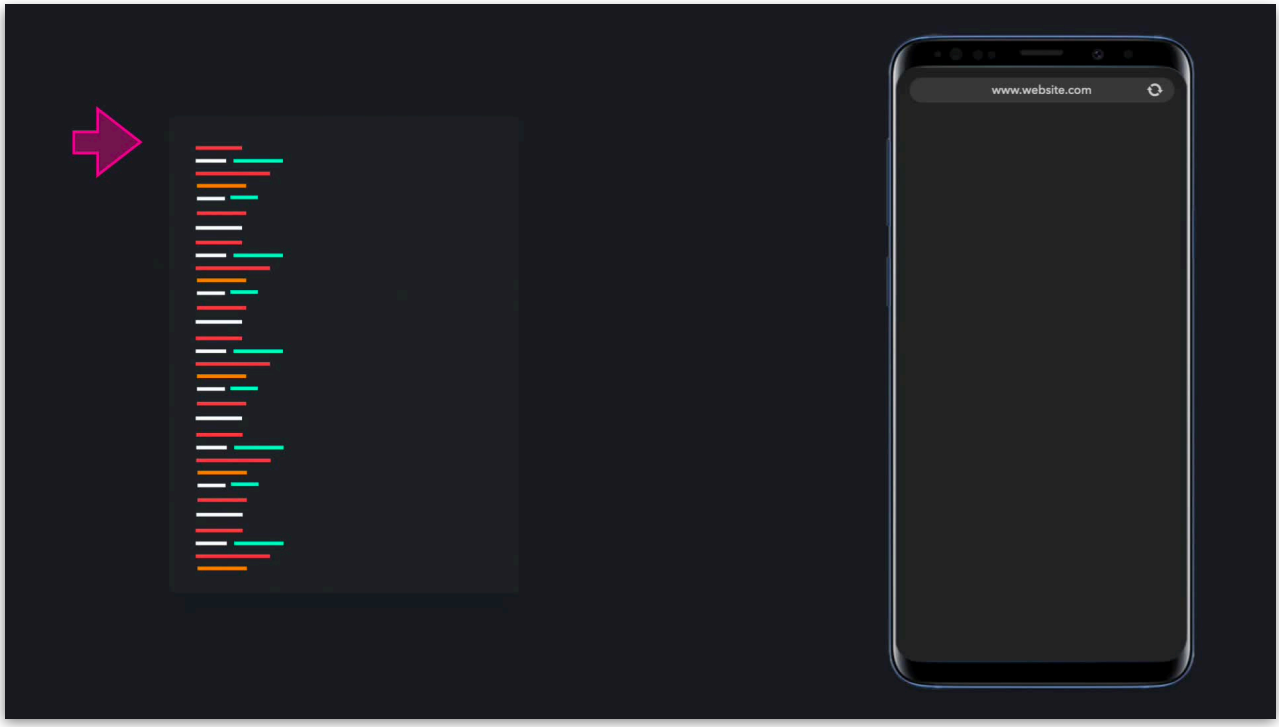


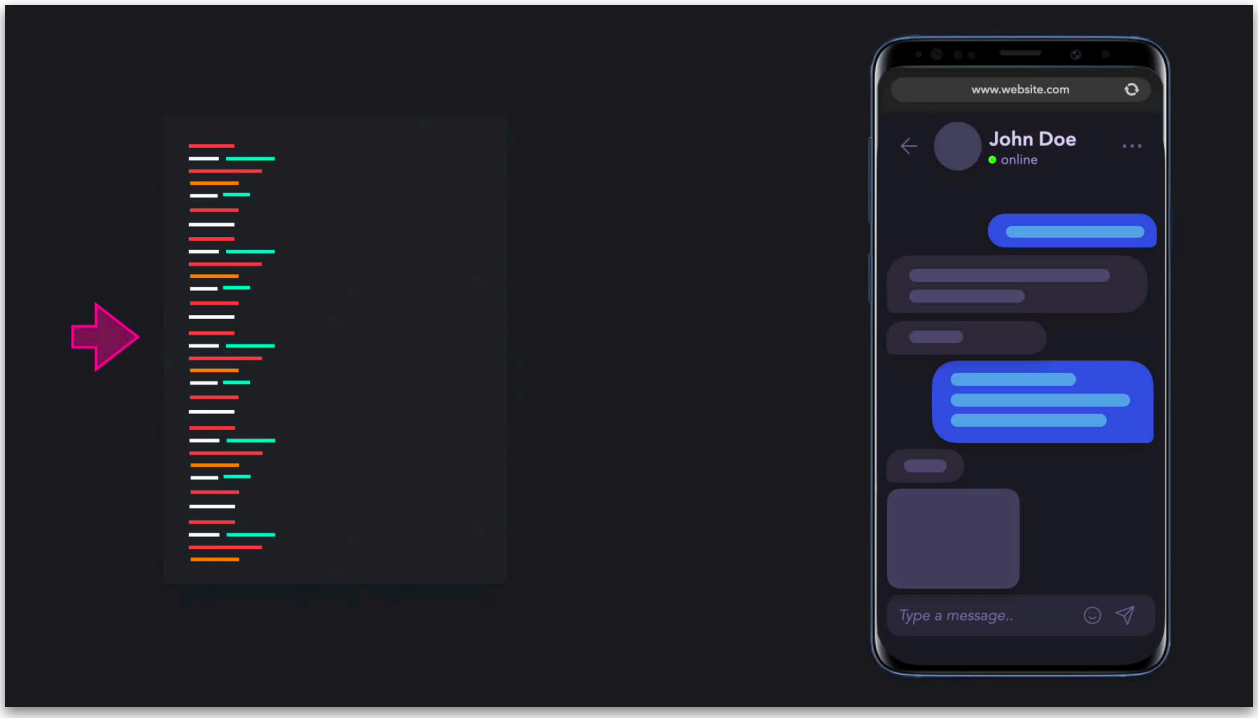


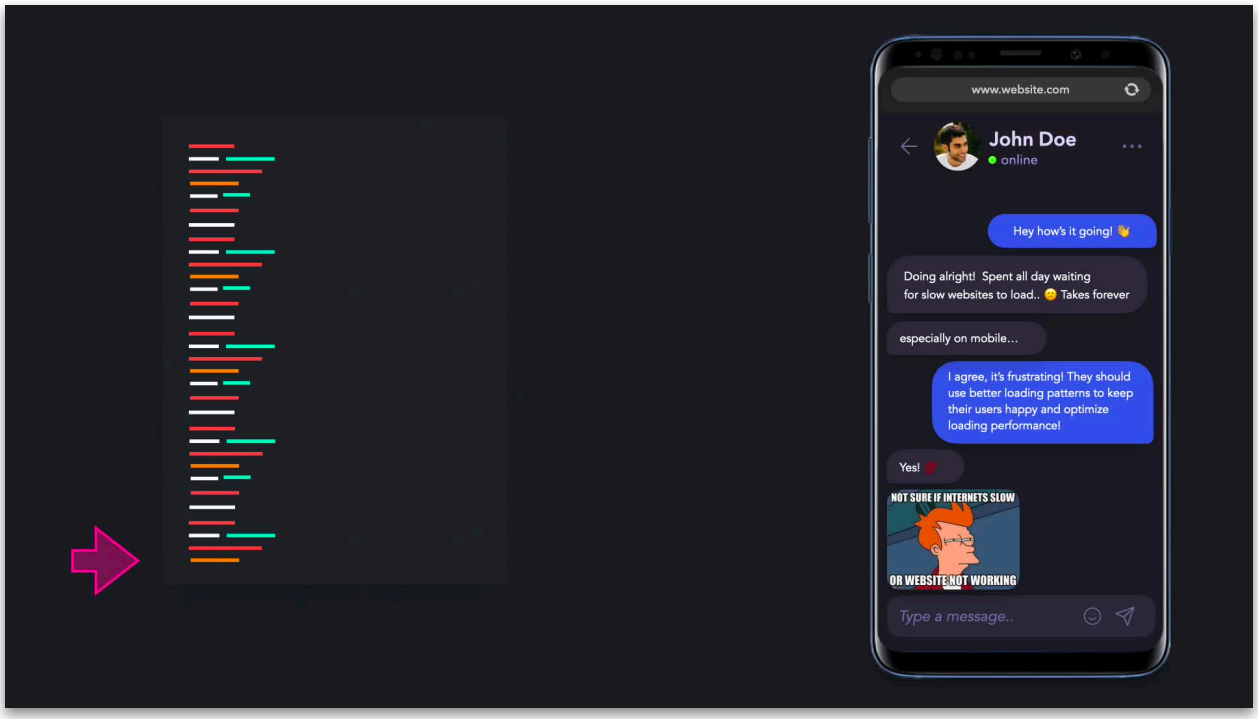
A bigger bundle doesn't necessarily mean a longer execution time. It could happen that we loaded a ton of code that the user won't even use! Maybe some parts of the bundle will only get executed on a certain user interaction, which the user may or may not do!

The engine still has to load, parse and compile code that's not even used on the initial render before the user is able to see anything on their screen. Although the parsing and compilation costs can be practically ignored due to the browser's performant way of handling these two steps, fetching a larger bundle than necessary can hurt the performance of your application. Users on low-end devices or slower networks will see a significant increase in loading time before the bundle has been fetched.

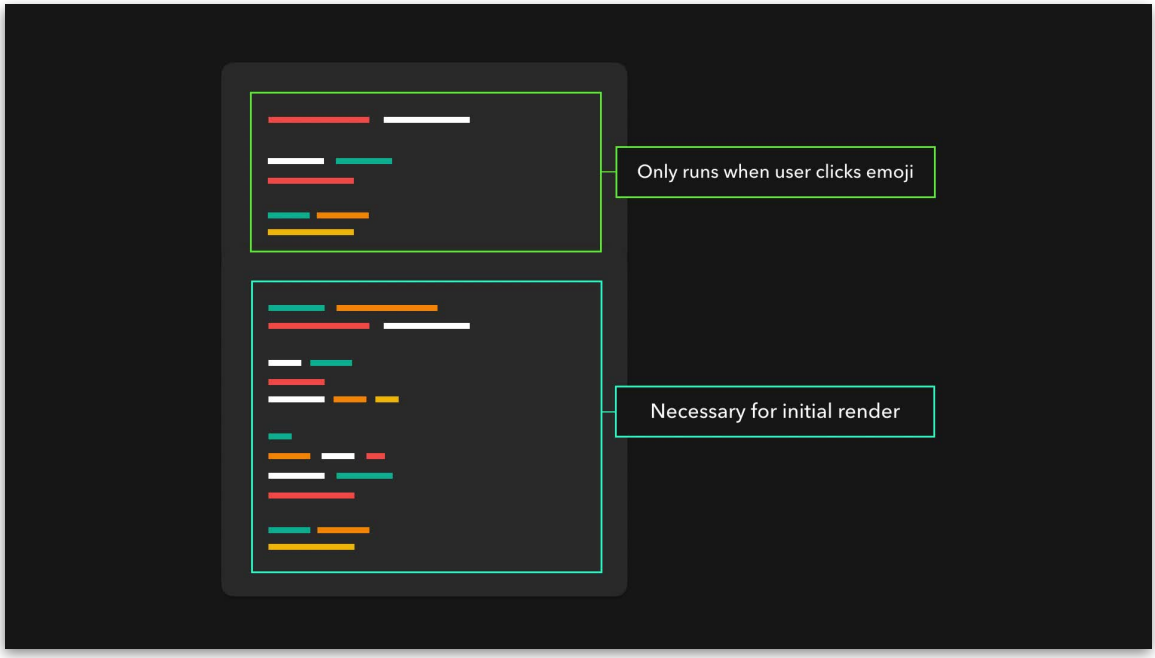

Network
Cache
Service Worker
Parser + Compiler
Executable Code

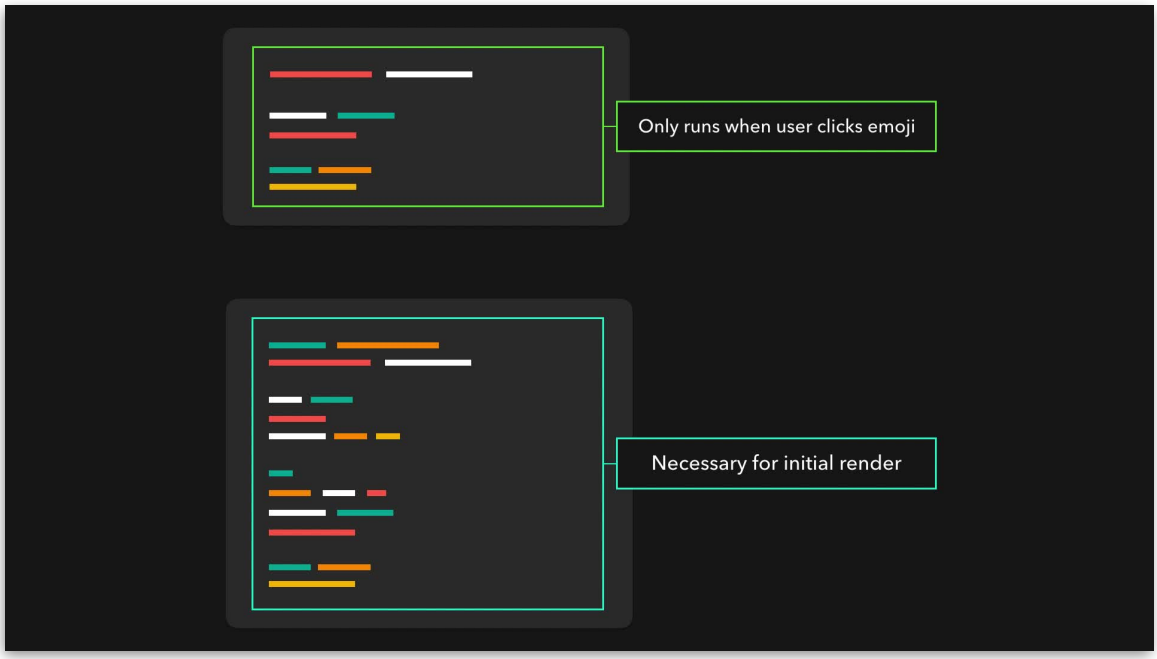

www.website.com


www.website.com
John Doe
online
Type a message..


www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..

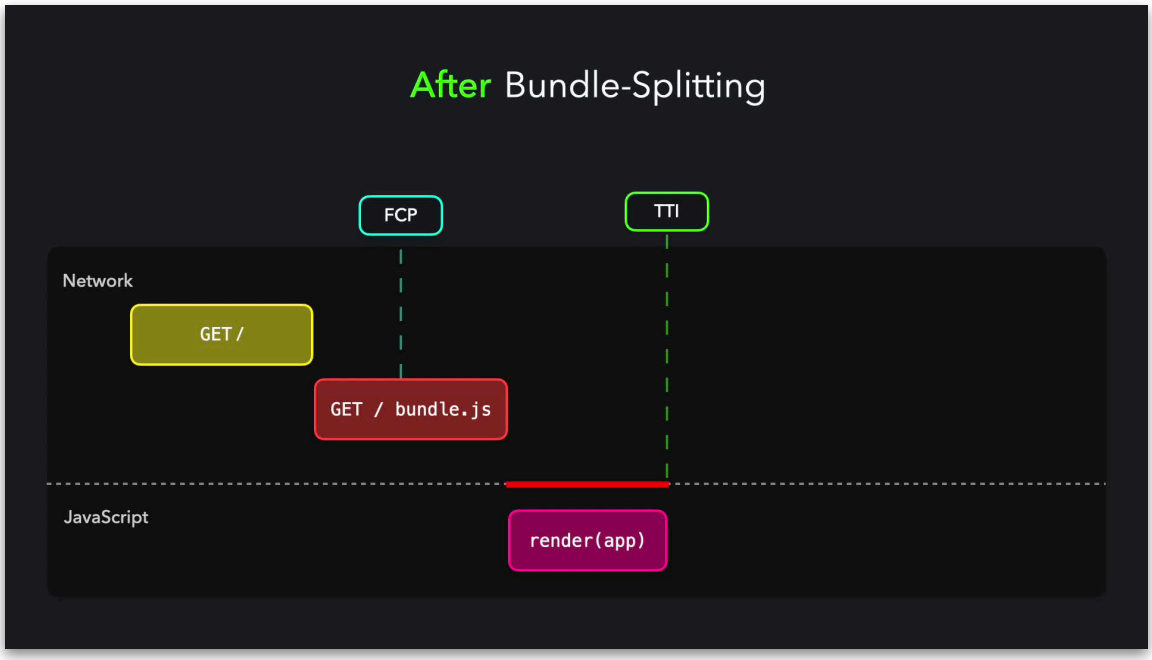
The first part still had to be loaded and processed, even though the engine only used the last part of the file in order to . Instead of intially requesting parts of the code that don't have a high priority in the current navigation, we can separate this code from the code that's needed in order to render the initial page.

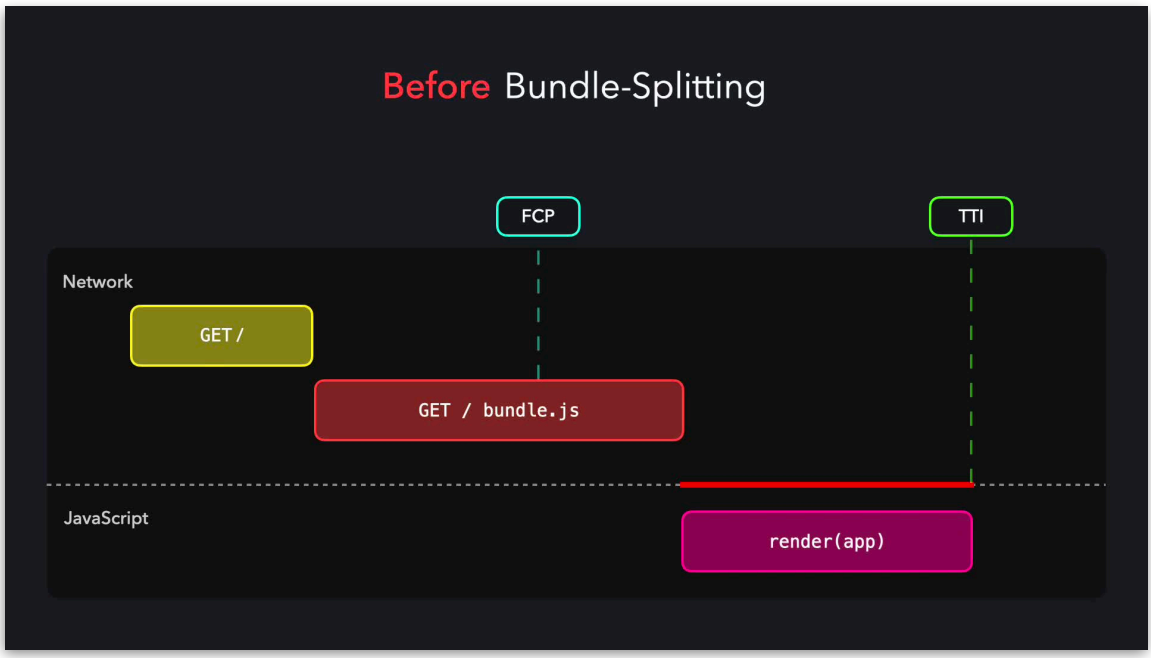




By bundle-splitting the large bundle into two smaller bundles, `main.bundle.js` and `emoji-picker.bundle.js`, we reduce the initial loading time by fetching a smaller amount of data.

In this project, we'll cover some methods that allow us to bundle-split our application into multiple smaller bundles, and load the resources in the most efficient and performant ways.





PRPL Pattern

Optimize initial load through precaching, lazy loading, and minimizing roundtrips

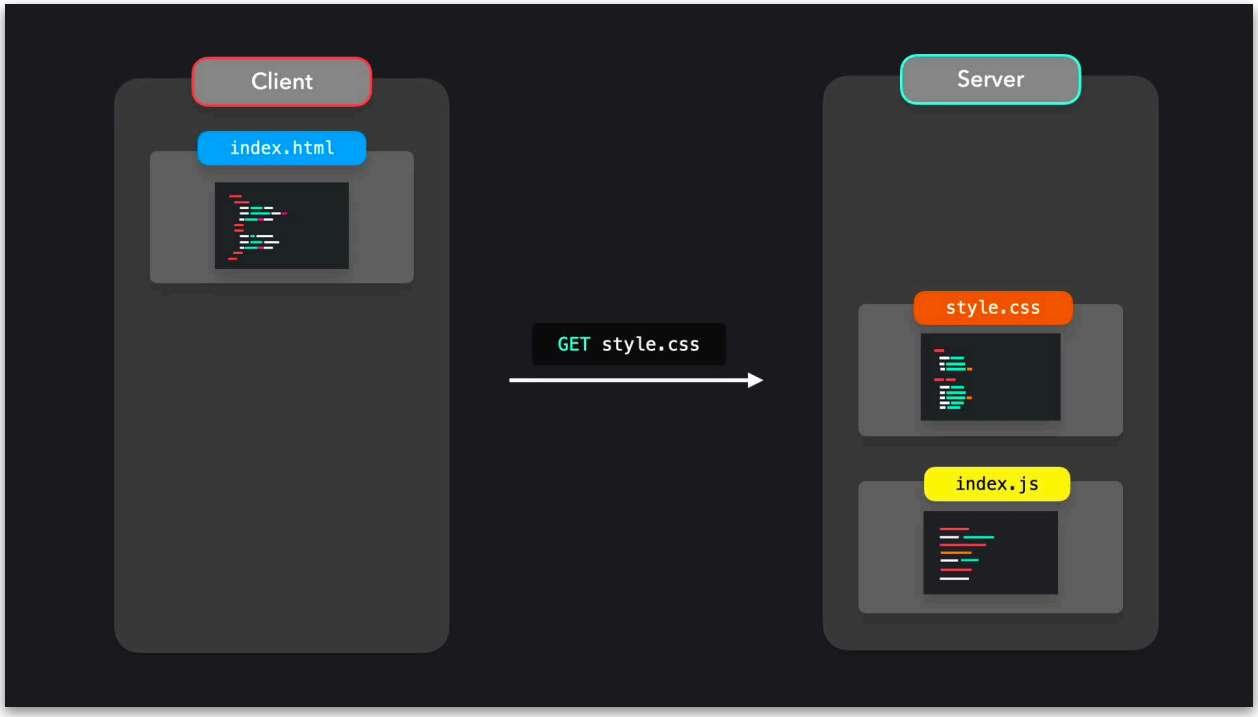
Making our applications globally accessible can be a challenge! We have to make sure the application is performant on low-end devices and in regions with a poor internet connectivity. In order to make sure our application can load as efficiently as possible in difficult conditions, we can use the PRPL pattern.

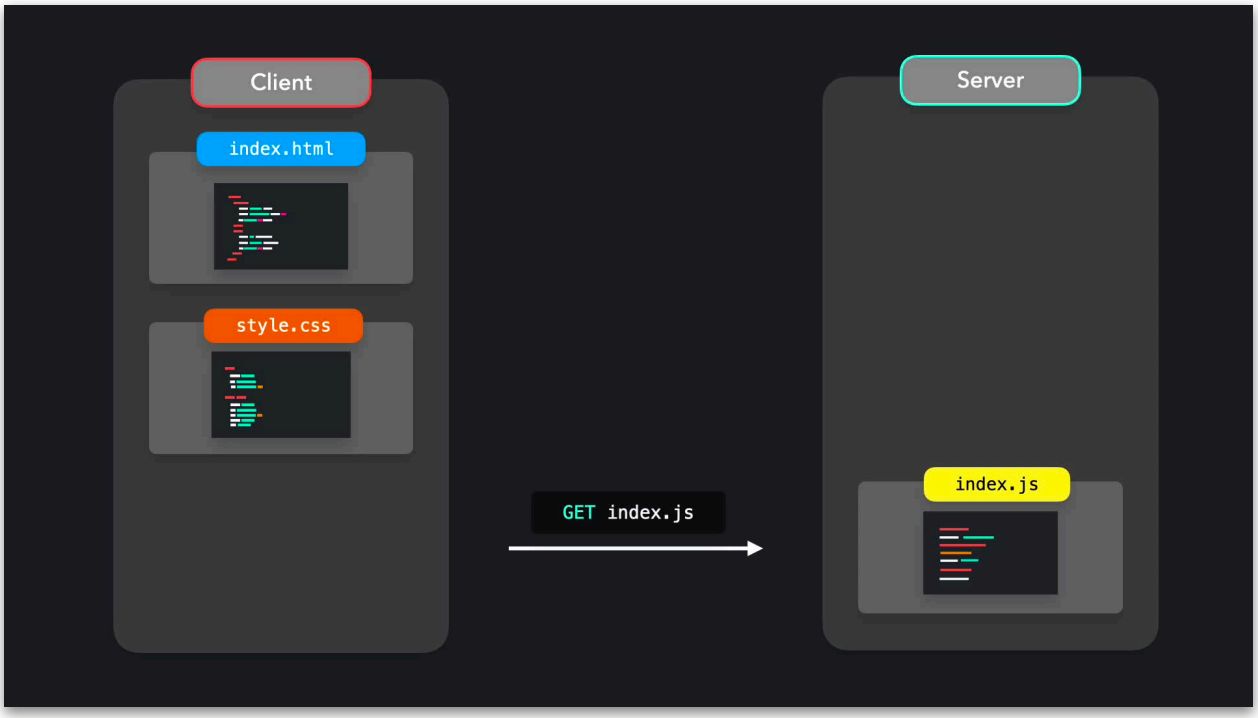
The PRPL pattern focuses on four main performance considerations:

- Pushing critical resources efficiently, which minimizes the amount of roundtrips to the server and reducing the loading time.
- Rendering the initial route soon as possible to improve the user experience
- Pre-caching assets in the background for frequently visited routes to minimize the amount of requests to the server and enable a better offline experience
- Lazily loading routes or assets that aren't requested as frequently

When we want to visit a website, we first have to make a request to the server in order to get those resources. The file that the entrypoint points to gets returned from the server, which is usually our application's initial HTML file!

The browser's HTML parser starts to parse this data as soon as it starts receiving it from the server. If the parser discovers that more resources are needed, such as stylesheets or scripts, another HTTP request is sent to the server in order to get those resources!





Having to repeatedly request the resources isn't optimal, as we're trying to minimize the amount of round trips between the client and the server!

For a long time, we used HTTP/1.1 in order to communicate between the client and the server. Although HTTP/1.1 introduced many improvement compared to HTTP/1.0, such as being able to keep the TCP connection between the client and the server alive before a new HTTP requests gets sent with the keep-alive header, there were still some issues that had to be solved!

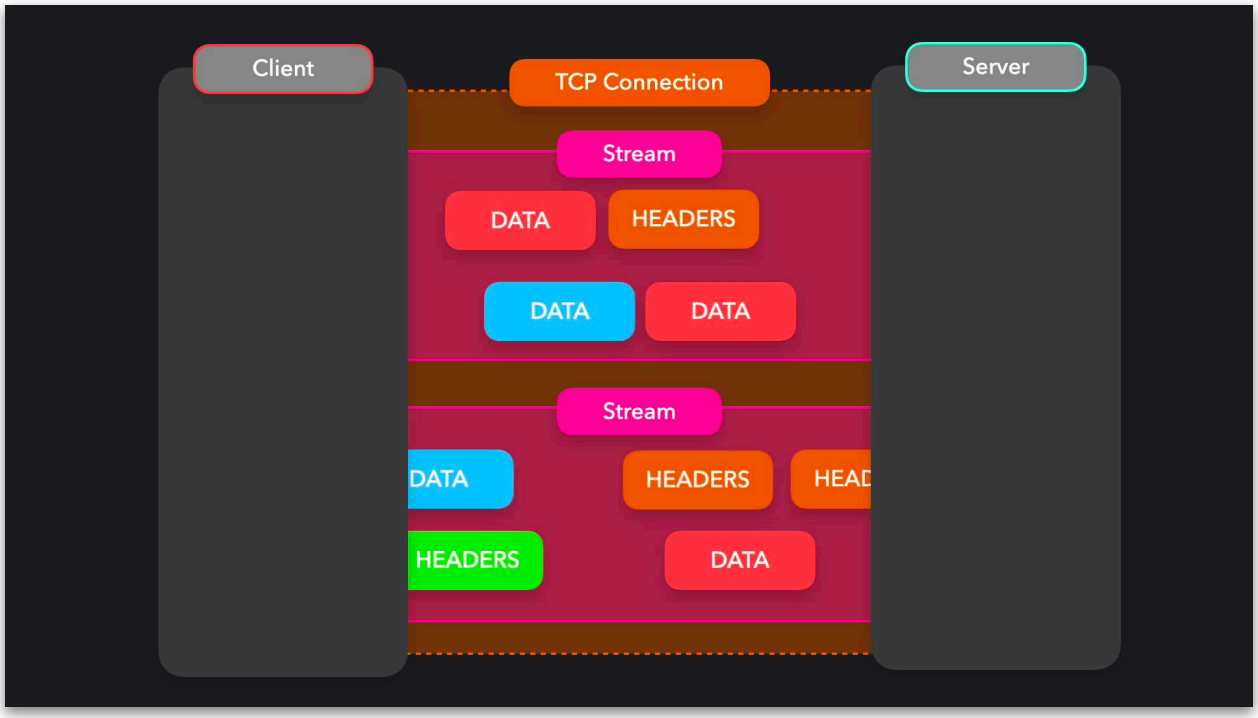
HTTP/2 introduced some significant changes compared to HTTP/1.1, which make it easier for us to optimize the message exchange between the client and the server.

Whereas HTTP/1.1 used a newline delimited plaintext protocol in the requests and responses, HTTP/2 splits the requests and responses up in smaller pieces called frames. An HTTP request that contains headers and a body field gets split into at least two frames: a headers frame, and a data frame!

HTTP/1.1 had a maximum amount of 6 TCP connections between the client and the server. Before a new request can get sent over the same TCP connection, the previous request has to be resolved! If the previous request is taking a long time to resolve, this request is blocking the other requests from being sent. This common issue is called head of line blocking, and can increase the loading time of certain resources!

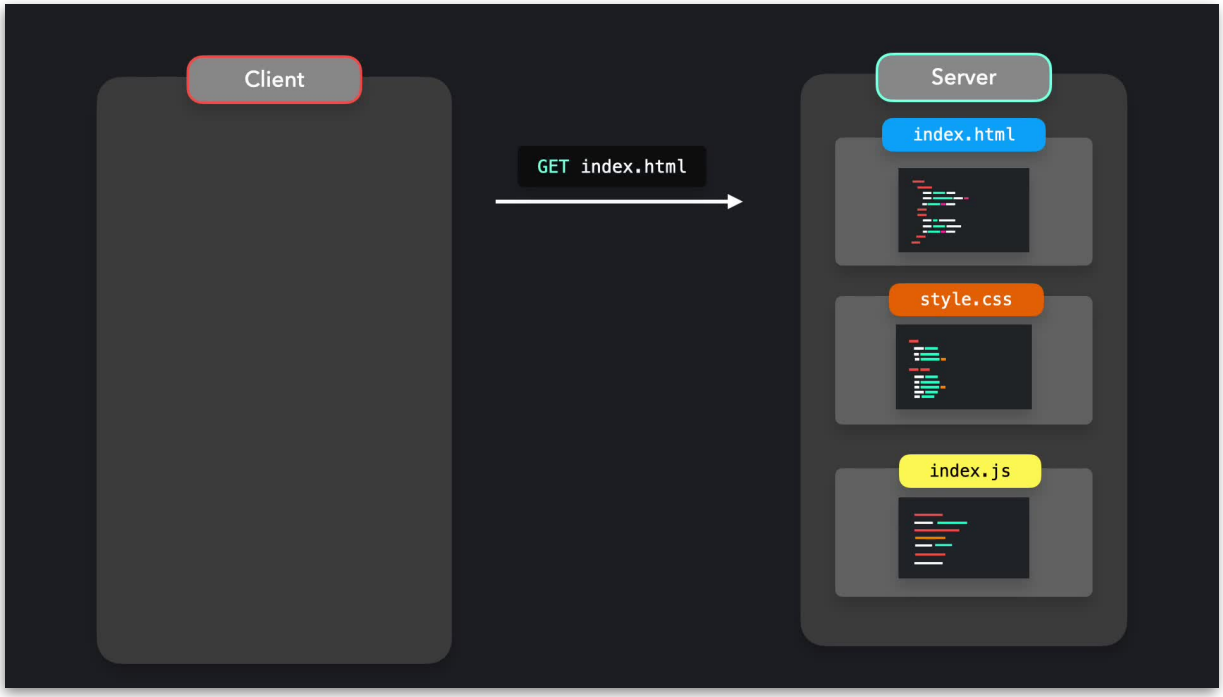
HTTP/2 makes use of bidirectional streams, which makes it possible to have one single TCP connection that includes multiple bidirectional streams, which can carry multiple request and response frames between the client and the server!

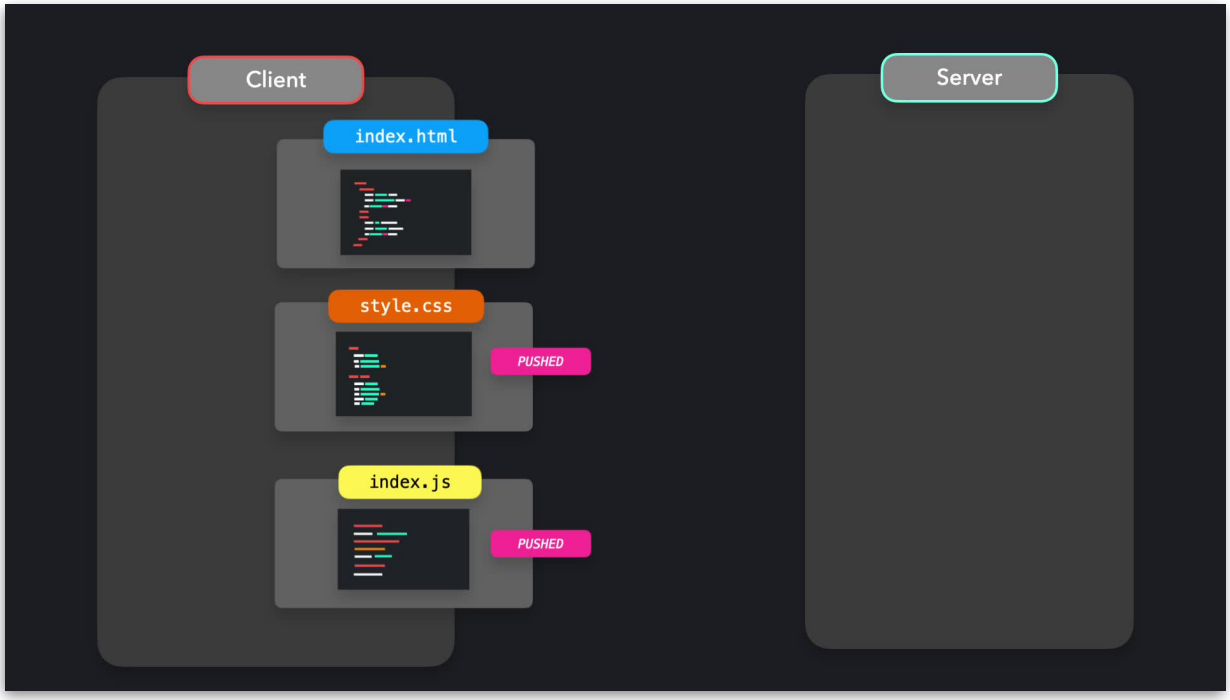
Once the server has received all request frames for that specific request, it reassembles them and generates response frames. These response frames are sent back to the client, which reassembles them. Since the stream is bidirectional, we can send both request and response frames over the same stream.

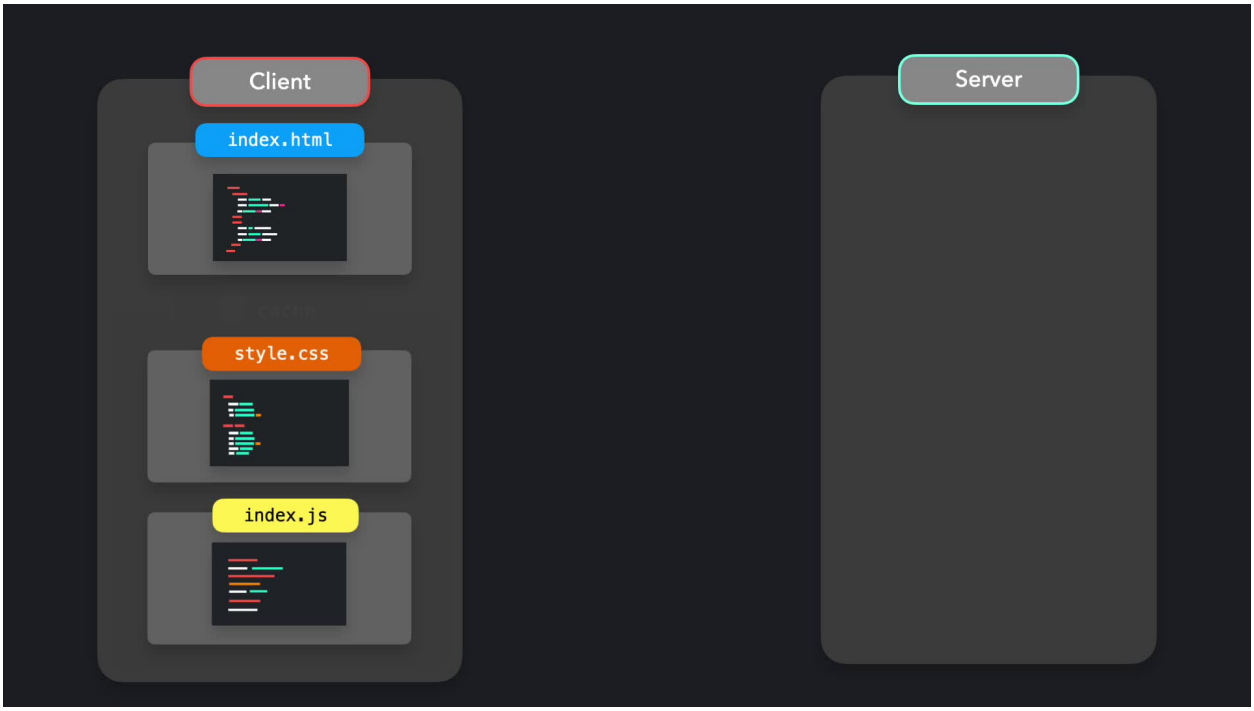


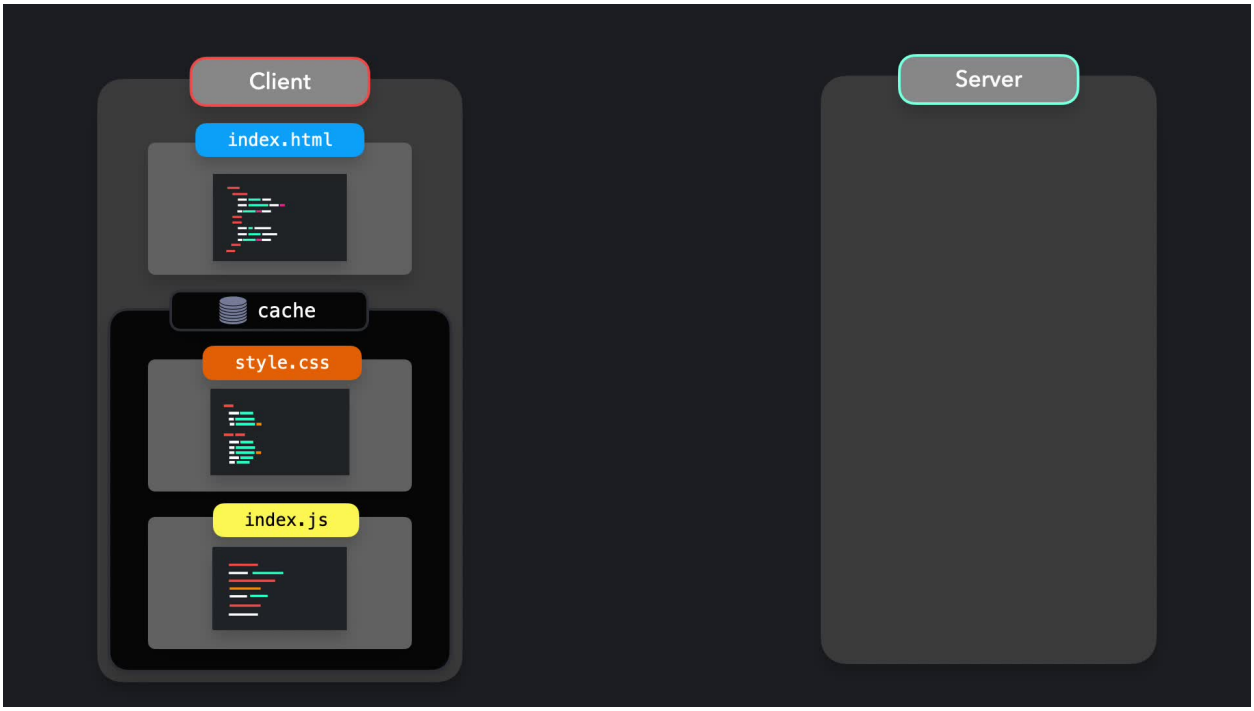
HTTP/2 solves head of line blocking by allowing multiple requests to get sent on the same TCP connection before the previous request resolves!

HTTP/2 also introduced a more optimized way of fetching data, called server push. Instead of having to explicitly ask for resources each time by sending an HTTP request, the server can send the additional resources automatically, by “pushing” these resources.









After the client has received the additional resources, the resources will get stored in browser cache. When the resources get discovered while parsing the entry file, the browser can quickly get the resources from cache instead of having to make an HTTP request to the server!

Although pushing resources reduces the amount of time to receive additional resources, server push is not HTTP cache aware! The pushed resources won't be available to us the next time we visit the website, and will have to be requested again. In order to solve this, the PRPL pattern uses service workers after the initial load to cache those resources in order to make sure the client isn't making unnecessary requests.

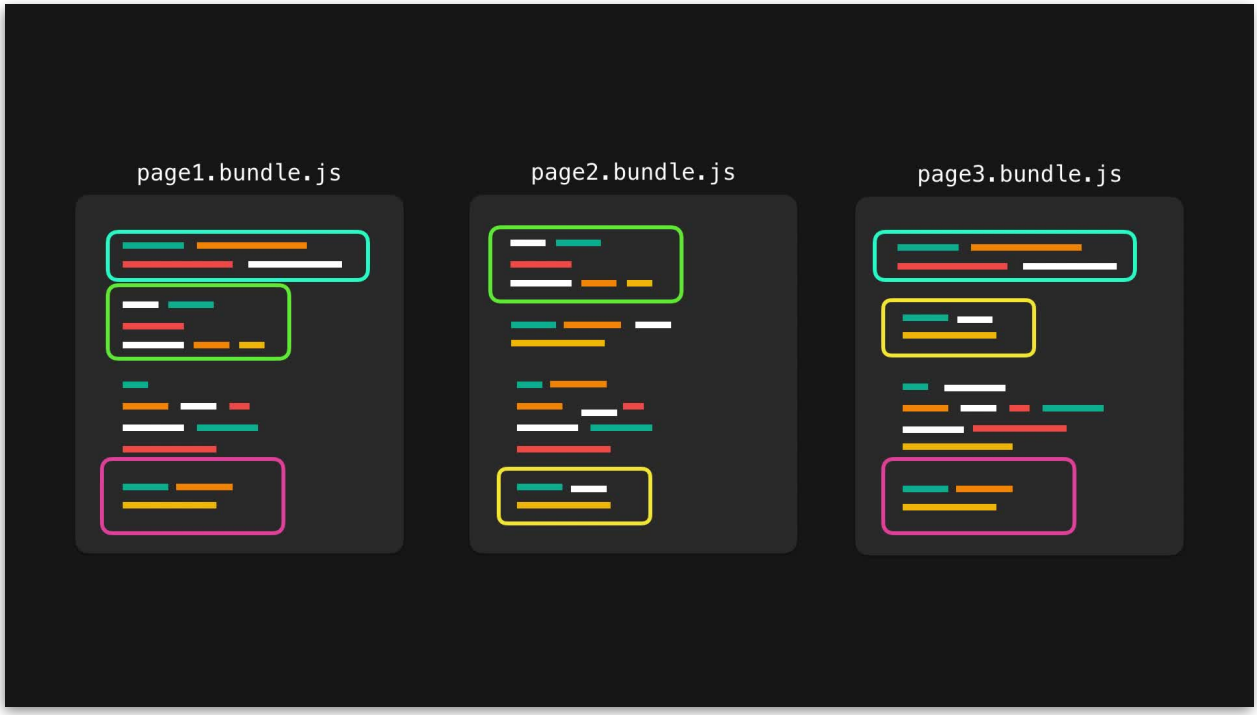
As the authors of a site, we usually know what resources are critical to fetch early on, while browsers do their best to guess this. Luckily, we can help the browser by adding a preload resource hint to the critical resources!

By telling the browser that you'd like to preload a certain resource, you're telling the browser that you would like to fetch it sooner than the browser would otherwise discover it! Preloading is a great way to optimize the time it takes to load resources that are critical for the current route.

Although preloading resources are a great way to reduce the amount of roundtrips and optimize loading time, pushing too many files can be harmful. The browser's cache is limited, and you may be unnecessarily using bandwidth by requesting resources that weren't actually needed by the client.

The PRPL pattern focuses on optimizing the initial load. No other resources get loaded before the initial route has loaded and rendered completely!

We can achieve this by code-splitting our application into small, performant bundles. Those bundles should make it possible for the users to only load the resources they need, when they need it, while also maximizing cachability! Caching larger bundles can be an issue. It can happen that multiple bundles share the same resources.

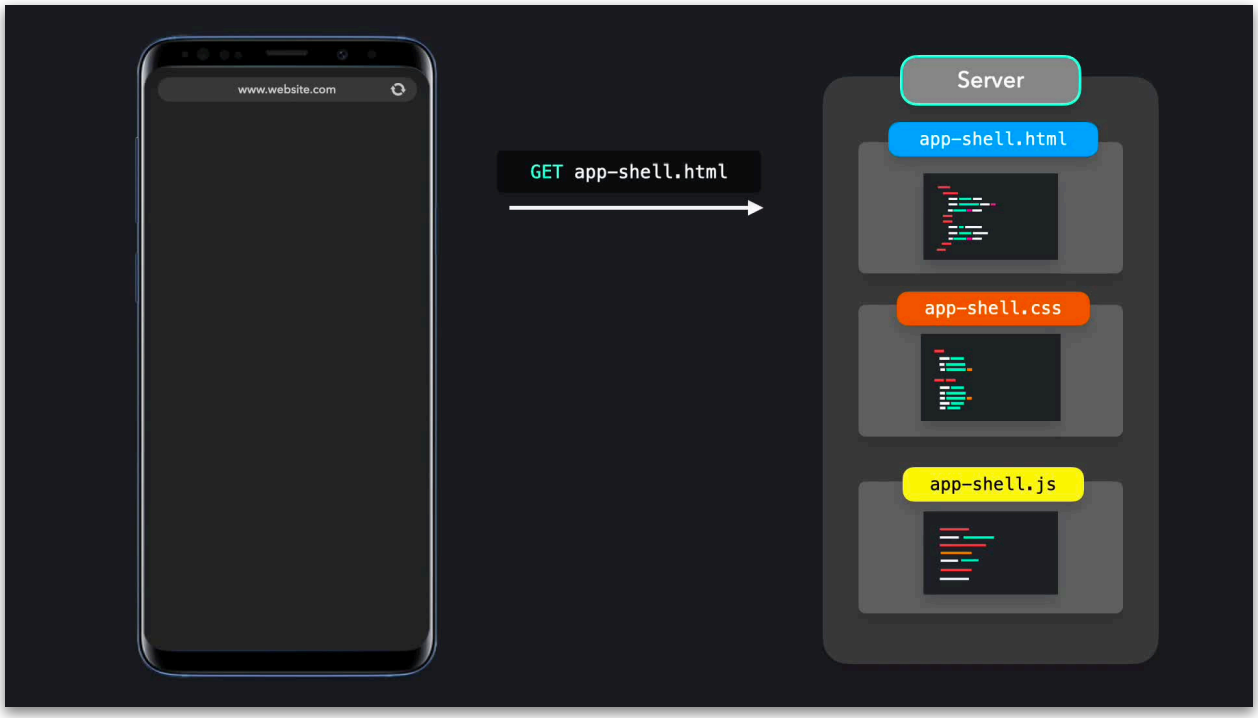


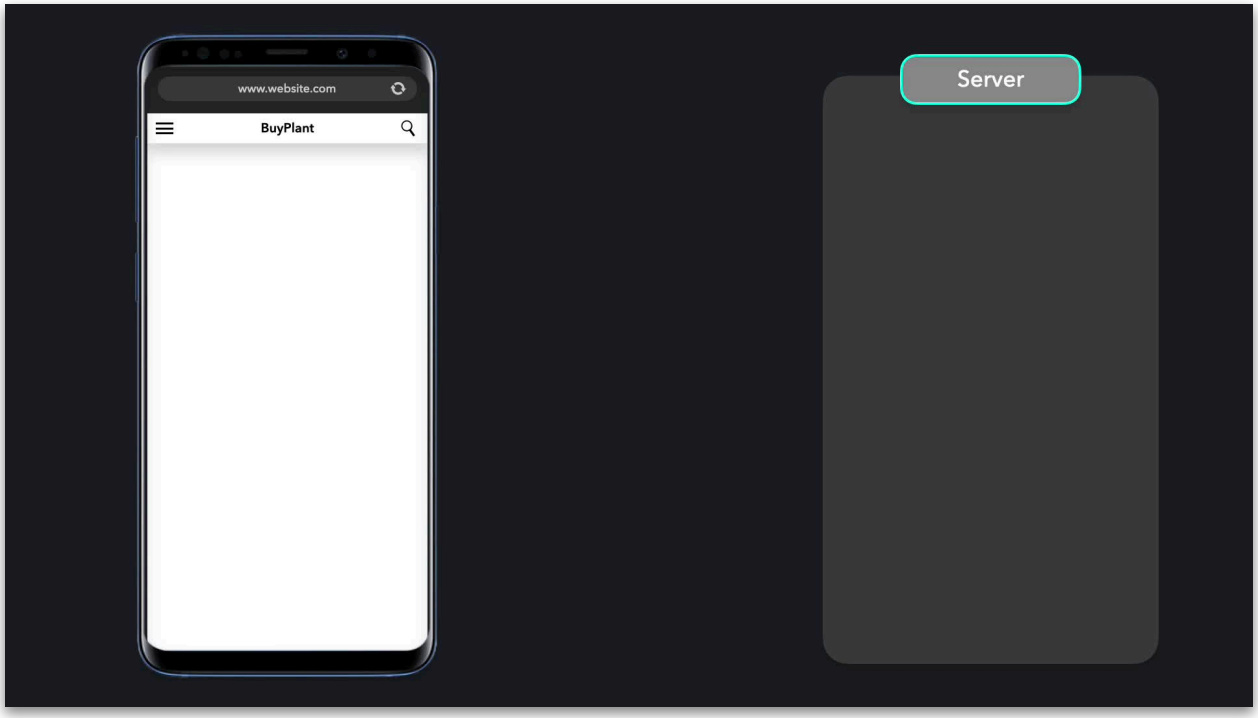
A browser has a hard time identifying which parts of the bundle are shared between multiple routes, and can therefore not cache these resources. Caching resources is important to reduce the number of roundtrips to the server, and to make our application offline-friendly!

When working with the PRPL pattern, we need to make sure that the bundles we're requesting contain the minimal amount of resources we need at that time, and are cachable by the browser. In some cases, this could mean that having no bundles at all would be more performant, and we could simply work with unbundled modules!

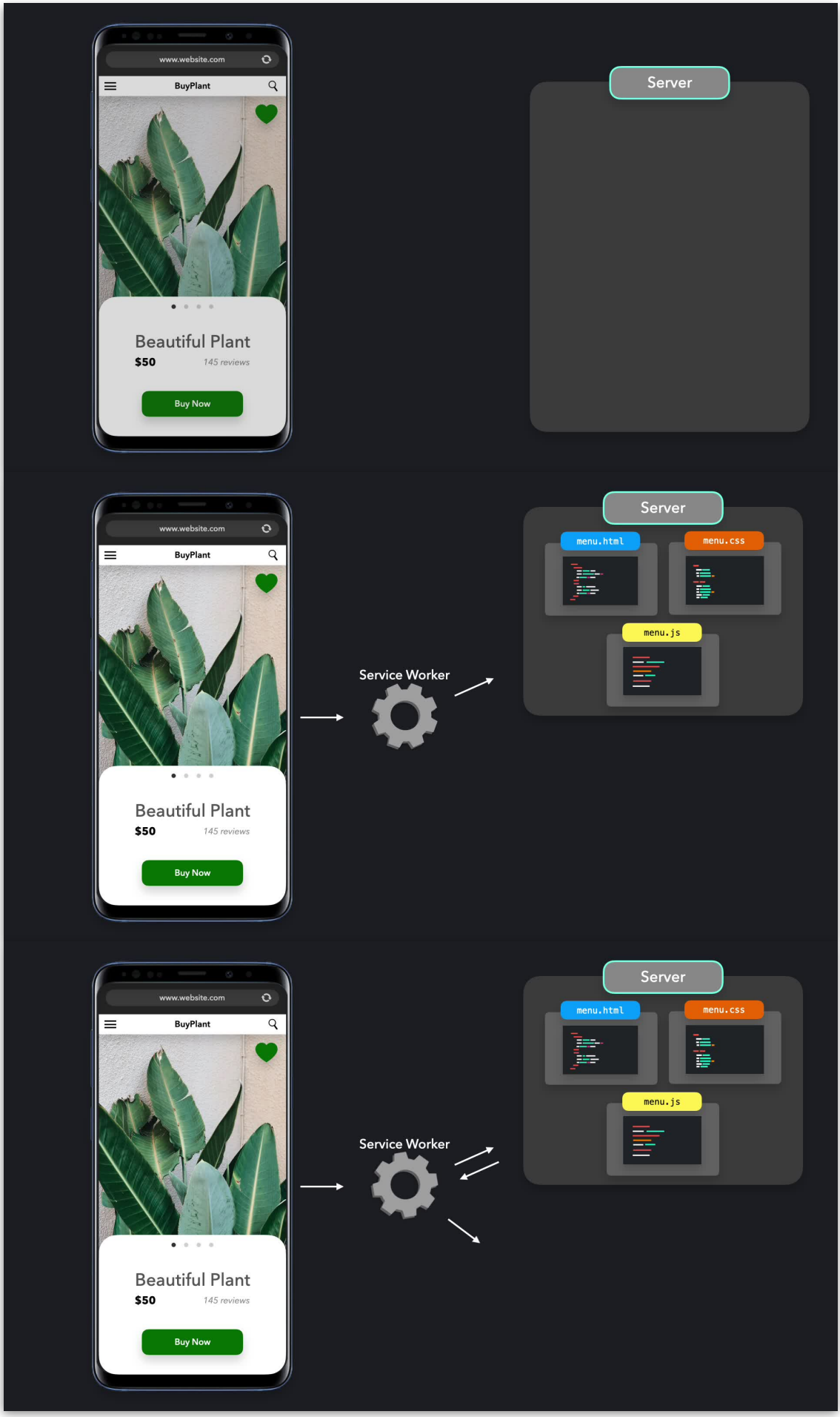
The benefit of being able to dynamically request minimal resources by bundling an application can easily be mocked by configuring the browser and server to support HTTP/2 push, and caching the resources efficiently. For browsers that don't support HTTP/2 server push, we can create a build that is optimized to minimize the amount of roundtrips. The client doesn't have to know whether it's receiving a bundled or unbundled resource: the server delivers the appropriate build for each browser.

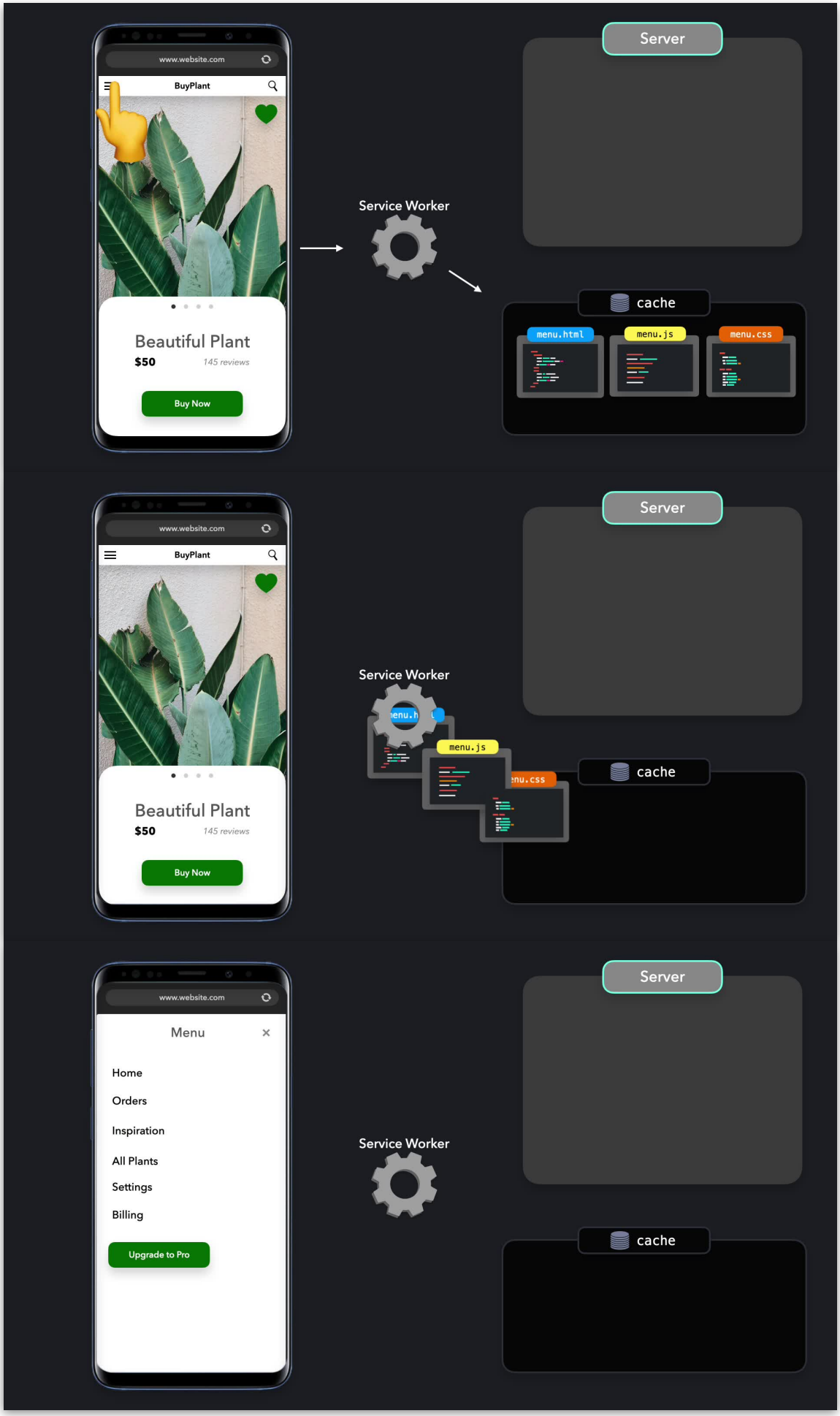
The PRPL pattern often uses an app shell as its main entry point, which is a minimal file that contains most of the application's logic and is shared between routes! It also contains the application's router, which can dynamically request the necessary resources.


www.website.com
GET app-shell.html
Server
app-shell.html
app-shell.css
app-shell.js


www.website.com
BuyPlant
Server

The PRPL pattern makes sure that no other resources get requested or rendered before the initial route is visible on the user's device. Once the initial route has been loaded successfully, a server worker can get installed in order to fetch the resources for the other frequently visited routes in the background!


www.website.com
BuyPlant
Beautiful Plant
$50
145 reviews
Buy Now
Server
Service Worker
menu.html
menu.css
menu.js


www.website.com
BuyPlant
Beautiful Plant
$50
145 reviews
Buy Now
Service Worker
Server
cache
menu.html
menu.js
menu.css
www.website.com
BuyPlant
Beautiful Plant
$50
145 reviews
Buy Now
Service Worker
Server
menu.js
cache
www.website.com
Menu
Home
Orders
Inspiration
All Plants
Settings
Billing
Upgrade to Pro
Service Worker
Server
cache

Since this data is being fetched in the background, the user won't experience any delays. If a user wants to navigate to a frequently visited route that's been cached by the service worker, the service worker can quickly get the required resources from cache instead of having to send a request to the server.

Resources for routes that aren't as frequently visited can be dynamically imported.

Tree Shaking

Reduce the bundle size by eliminating dead code

It can happen that we add code to our bundle that isn't used anywhere in our application. This piece of dead code can be eliminated in order to reduce the size of the bundle, and prevent unnecessarily loading more data! The process of eliminating dead code before adding it to our bundle, is called tree-shaking

Although tree-shaking works for simple modules like the math module, there are some cases in which tree-shaking can be tricky.

Concepts

Tree shaking is aimed at removing code that will never be used from a final JavaScript bundle. When done right, it can reduce the size of your JavaScript bundles and lower download, parse and (in some cases) execution time. For most modern JavaScript apps that use a module bundler (like webpack or Rollup), your bundler is what you would expect to automatically remove dead code.

Consider your application and its dependencies as an abstract syntax tree (we want to "shake" the syntax tree to optimize it). Each node in the tree is a dependency that gives your app functionality. In Tree shaking, input files are treated as a graph. Each node in the graph is a top level statement which is

called a "part" in the code. Tree shaking is a graph traversal which starts from the entry point and marks any traversed paths for inclusion.

Every component can declare symbols, reference symbols, and rely on other files. Even the "parts" are marked as having side effects or not. For example, the statement `let firstName = 'Jane'` has no side effects because the statement can be removed without any observed difference if nothing needs foo. But the statement `let firstName = getName()` has side effects, because the call to `getName()` can not be removed without changing the meaning of the code, even if nothing needs `firstName`.

Imports

Only modules defined with the ES2015 module syntax (import and export) can be tree-shaken. The way you import modules specifies whether the module can be tree-shaken or not.

Tree shaking starts by visiting all parts of the entry point file with side effects, and proceeds to traverse the edges of the graph until new sections are reached. Once the traversal is completed, the JavaScript bundle includes only the parts that were reached during the traversal. The other pieces are left out. Let's say we define the following `utilities.js` file:

```
export function read(props) {
    return props.book
}

export function nap(props) {
    return props.winks
}
```

Then we have the following `index.js` file:

```
import { read } from 'utilities';

eventHandler = (e) => {
    read({ book: e.target.value })
}
```

In this example, `nap()` isn't important and therefore won't be included in the bundle.

Side Effects

When we're importing an ES6 module, this module gets executed instantly. It could happen that although we're not referencing the module's exports anywhere in our code, the module itself affects the global scope while it's being executed (polyfills or global stylesheets, for example). This is called a side effect. Although we're not referencing the exports of the module itself, if the module has exported values to begin with, the module cannot be tree-shaken due to the special behavior when it's being imported!

The Webpack documentation gives a clear explanation on tree-shaking and how to avoid breaking it.

Preload

Inform the browser of critical resources before they are discovered

Preload (`<link rel="preload">`) is a browser optimization that allows critical resources (that may be discovered late) to be to be requested earlier. If you are comfortable thinking about how to manually order the loading of your key resources, it can have a positive impact on loading performance and metrics in the Core Web Vitals. That said, preload is not a panacea and requires an awareness of some trade-offs.

```
<html>
  <head>
    <link rel="preload" href="emoji-picker.js" as="script">
  </head>
  <body>
    ...
    <script src="stickers.js" defer></script>
    <script src="video-sharing.js" defer></script>
    <script src="emoji-picker.js" defer></script>
  </body>
</html>
```

When optimizing for metrics like Time To Interactive or First Input Delay, preload can be useful to load JavaScript bundles (or chunks) that are necessary for interactivity. Keep in mind that great care is needed when using preload as you want to avoid improving interactivity at the cost of

delaying resources (like hero images or fonts) necessary for First Contentful Paint or Largest Contentful Paint.

If you are trying to optimize the loading of first-party JavaScript, you can also consider using `<script defer>` in the document `<head>` vs. `<body>` to help with early discover of these resources.

Preload in single-page apps

While prefetching is a great way to cache resources that may be requested some time soon, we can preload resources that need to be used instantly. Maybe it's a certain font that is used on the initial render, or certain images that the user sees right away.

Say our `EmojiPicker` component should be visible instantly on the initial render. Although it should not be included in the main bundle, it should get loaded in parallel. Just like prefetch, we can add a magic comment in order to let Webpack know that this module should be preloaded.

```
const EmojiPicker = import(/* webpackPreload: true */ "./EmojiPicker");
```

ChatInput.js

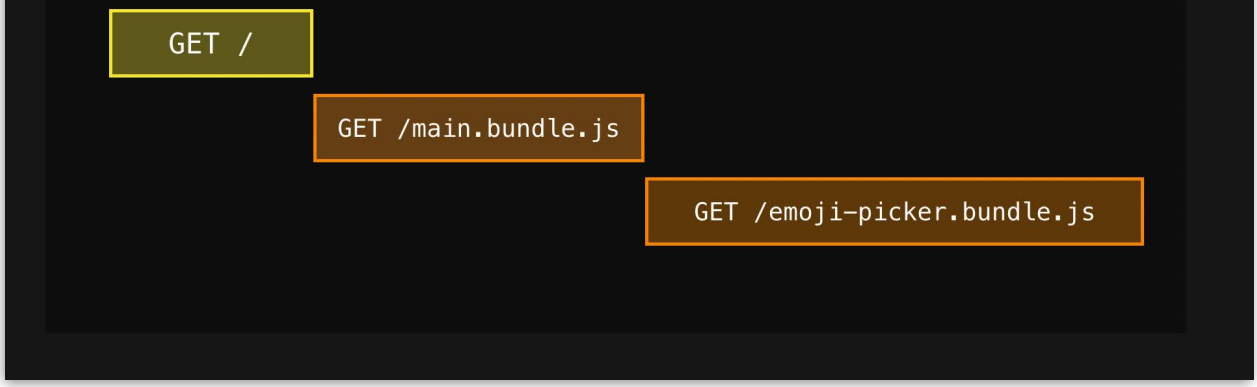
```
import React, { Suspense, lazy } from "react";
import Send from "./icons/Send";
import Emoji from "./icons/Emoji";

const EmojiPicker = lazy(() => import("./EmojiPicker"));
const ChatInput = () => {
  const [pickerOpen, togglePicker] = React.useReducer(state => !state, true);

  return (
    <div className="chat-input-container">
      <input type="text" placeholder="Type a message..." />
      <Emoji onClick={togglePicker} />
      {pickerOpen && (
        <Suspense fallback={<p id="loading">loading</p>}>
          <EmojiPicker />
        </Suspense>
      )}
      <Send />
    </div>
  );
};

console.log("ChatInput loading", Date.now());

export default ChatInput;
```

webpack.config.js

```
const path = require("path");
const HTMLWebpackPlugin = require("html-webpack-plugin");
const PreloadWebpackPlugin = require("preload-webpack-plugin");

module.exports = {
  entry: {
    main: "./src/index.js",
    emojiPicker: "./src/components/EmojiPicker.js"
  },
  module: { ... },
  resolve: {... },
  output: { ... },
  plugins: [
    new HTMLWebpackPlugin({
      template: path.resolve(__dirname, "dist", "index.html")
    }),
    new PreloadWebpackPlugin({
      rel: "preload",
      as: "script",
      include: ["emojiPicker"]
    })
  ]
};
```

Webpack 4.6.0+ allows preloading of resources by adding `/* webpackPreload: true */` to the import. In order to make preloading work in older versions of webpack, you'll need to add the `preload-webpack-plugin` to your webpack config.

```
import( "./EmojiPicker")
```




```
import( /* webpackPreload: true */ "./EmojiPicker")
```

After building the application, we can see that the `EmojiPicker` will be prefetched.

```
 Asset                            Size       Chunks                         Chunk Names
    emoji-picker.bundle.js        1.49 KiB   emoji-picker [emitted]         emoji-picker
    vendors~emoji-picker.bundle.js 171 KiB   vendors~emoji-picker [emitted] vendors~emoji-picker
    main.bundle.js                1.34 MiB   main  [emitted]                main

Entrypoint main = main.bundle.j
(preload: vendors~emoji-picker.bundle.js emoji-picker.bundle.js)
```

The actual output is visible as a `link` tag with `rel="preload"` in the head of our document.

```
<link rel="preload" href="emoji-picker.bundle.js" as="script" />
<link rel="preload" href="vendors~emoji-picker.bundle.js" as="script" />
```

The preloaded `EmojiPicker` could be loaded in parallel with the initial bundle. Unlike prefetch, where the browser still had a say in whether it think it's got a good enough internet connection and bandwidth to actually prefetch the resource, a preloaded resource will get preloaded no matter what.

Instead of having to wait until the `EmojiPicker` gets loaded after the initial render, the resource will be available to us instantly! As we're loading assets with smarter ordering, the initial loading time may increase significantly depending on your users device and internet connection. Only preload the resources that have to be visible ~1 second after the initial render.

Preload + the async hack

Should you wish for browsers to download a script as high-priority, but not block the parser waiting for a script, you can take advantage of the `preload` + `async` hack below. The download of other resources may be delayed by the preload in this case, but this is a trade-off a developer has to make:

```
<link rel="preload" href="emoji-picker.js" as="script">
<script src="emoji-picker.js" async>
```

Conclusions

Again, use preload sparingly and always measure its impact in production. If the preload for your image is earlier in the document than it is, this can help browsers discover it (and order relative to other resources). When used incorrectly, preloading can cause your image to delay First Contentful Paint (e.g CSS, Fonts) - the opposite of what you want. Also note that for such reprioritization efforts to be effective, it also depends on servers prioritizing requests correctly.

You may also find `<link rel="preload">` to be helpful for cases where you need to fetch scripts without executing them.

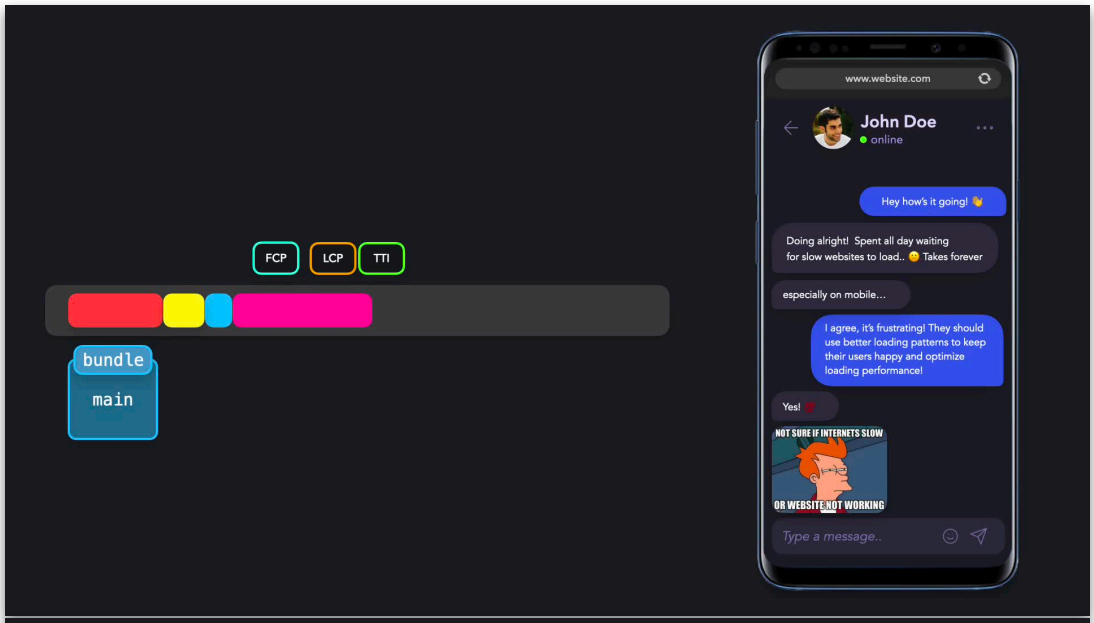
Prefetch

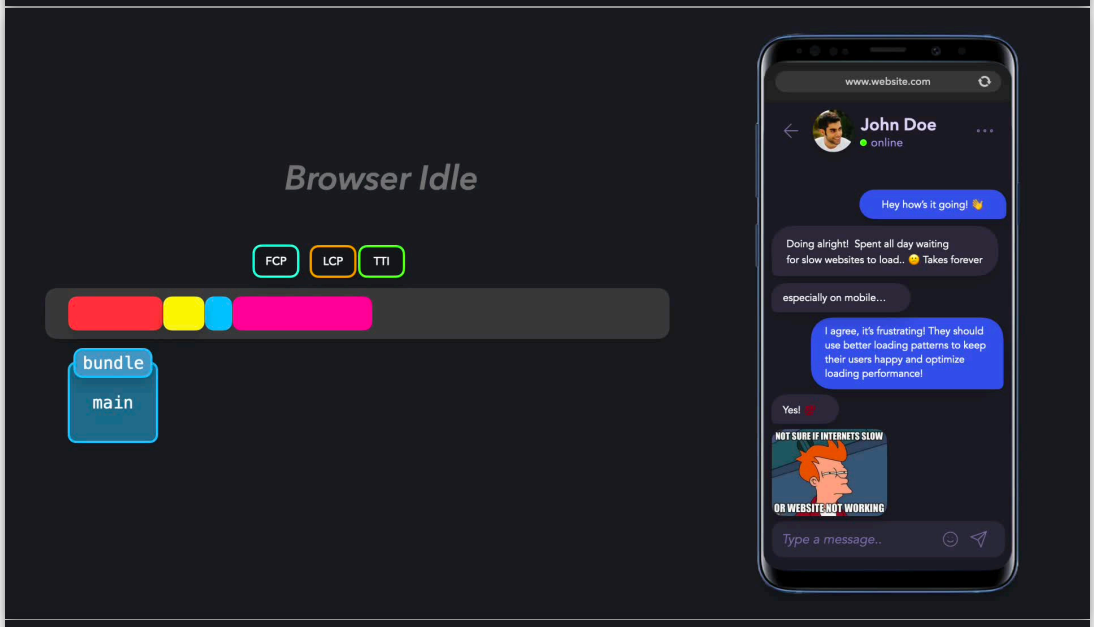
Fetch and cache resources that may be requested some time soon

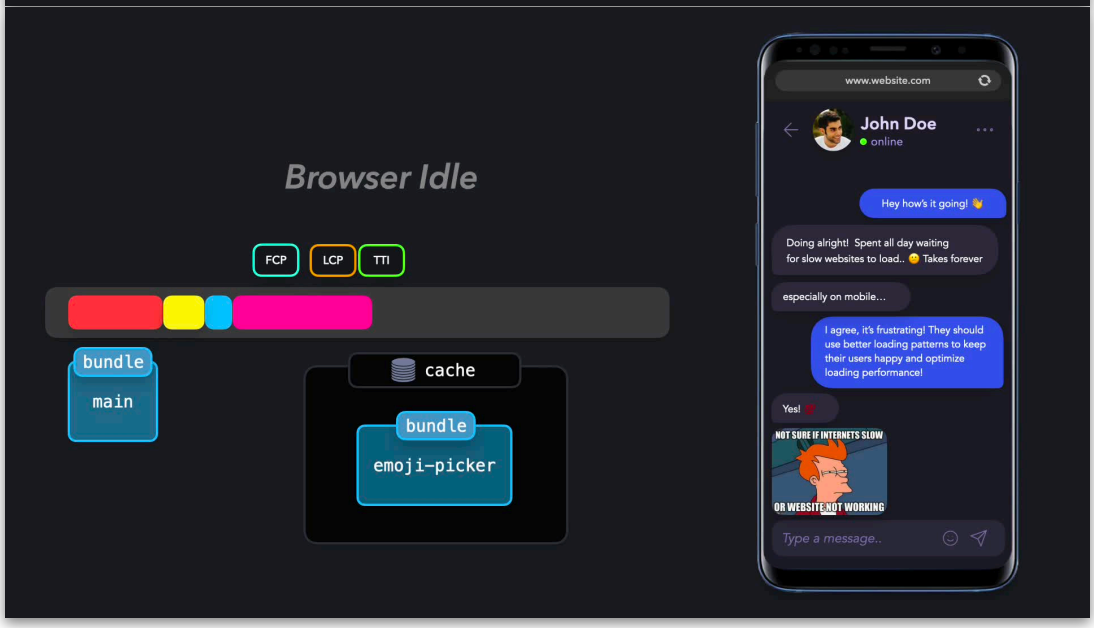
Prefetch (`<link rel="prefetch">`) is a browser optimization which allows us to fetch resources that may be needed for subsequent routes or pages before they are needed. Prefetching can be achieved in a few ways. It can be done declaratively in HTML (such as in the example below), via a HTTP Header (`Link: </js/chat-widget.js>; rel=prefetch`), Service Workers or via more custom means such as through Webpack.

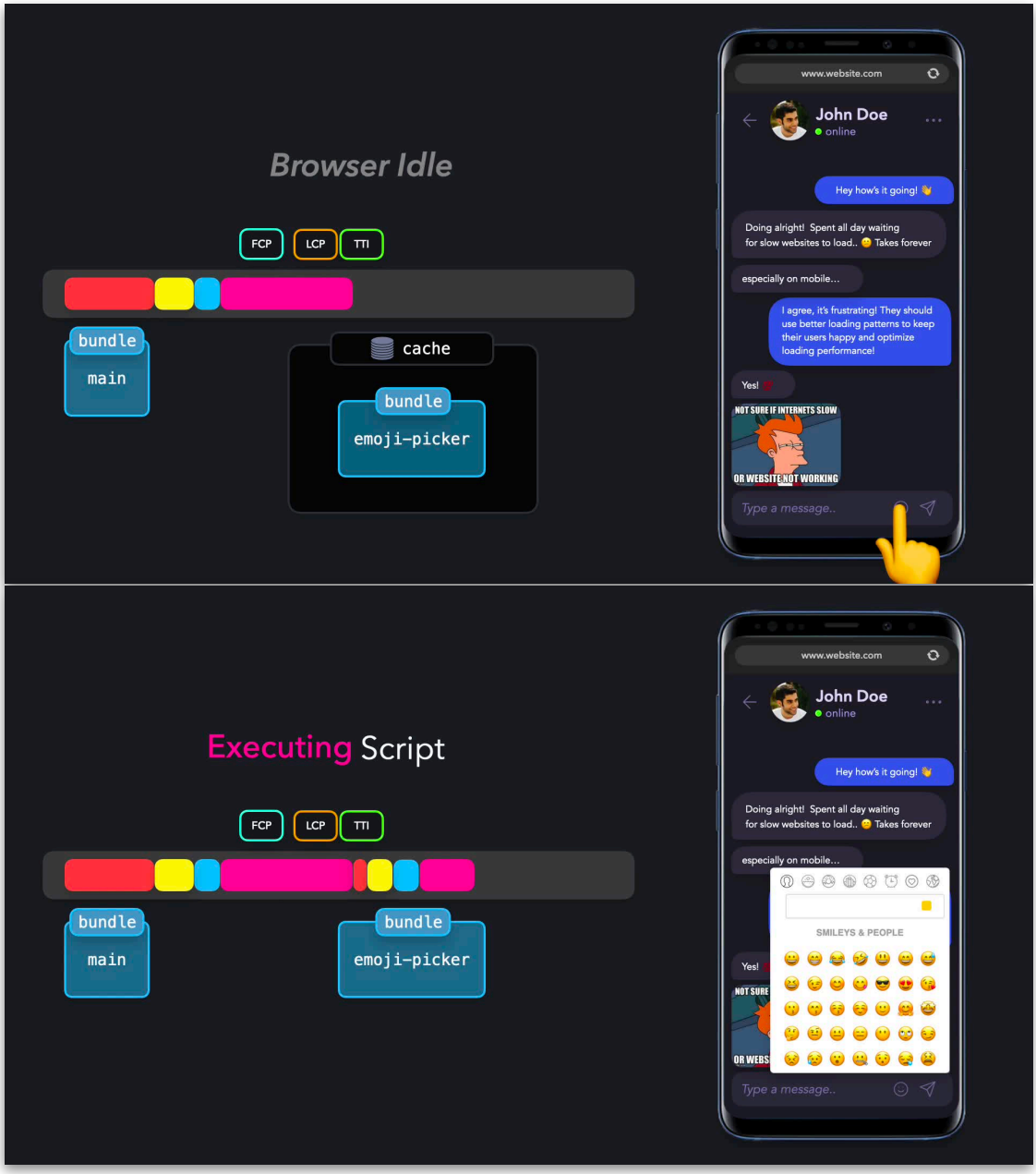
```
<link rel="prefetch" href="/pages/next-page.html">
<link rel="prefetch" href="/js/emoji-picker.js">
```

In the examples showing how we can import modules based on visibility or interaction, we saw that there was often some delay between clicking on the button in order to toggle the component, and showing the actual component on the screen. This happened, since the module still had to get requested and loaded when the user clicked on the button!


FCP
LCP
TTI
bundle
main
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..


Browser Idle
FCP
LCP
TTI
bundle
main
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..


Browser Idle
FCP
LCP
TTI
bundle
main
cache
bundle
emoji-picker
www.website.com
John Doe
online
Hey how's it going!
Doing alright! Spent all day waiting for slow websites to load.. Takes forever
especially on mobile...
I agree, it's frustrating! They should use better loading patterns to keep their users happy and optimize loading performance!
Yes!
NOT SURE IF INTERNETS SLOW
OR WEBSITE NOT WORKING
Type a message..



In many cases, we know that users will request certain resources soon after the initial render of a page. Although they may not visible instantly, thus shouldn't be included in the initial bundle, it would be great to reduce the loading time as much as possible to give a better user experience!

Components or resources that we know are likely to be used at some point in the application can be prefetched. We can let Webpack know that certain bundles need to be prefetched, by adding a magic comment to the import statement: `/* webpackPrefetch: true */`.

```
const EmojiPicker = import(/* webpackPrefetch: true */ "./EmojiPicker");
```

After building the application, we can see that the EmojiPicker will be prefetched.

```
<link rel="prefetch" href="emoji-picker.bundle.js" as="script" />
<link rel="prefetch" href="vendors~emoji-picker.bundle.js" as="script" />
```

The actual output is visible as a link tag with rel="prefetch" in the head of our document.

```
 Asset                              Size       Chunks                         Chunk Names
    emoji-picker.bundle.js          1.49 KiB   emoji-picker [emitted]         emoji-picker
    vendors~emoji-picker.bundle.js  171 KiB    vendors~emoji-picker [emitted] vendors~emoji-picker
    main.bundle.js                  1.34 MiB   main  [emitted]                main

Entrypoint main = main.bundle.js
(prefetch: vendors~emoji-picker.bundle.js emoji-picker.bundle.js)
```

Modules that are prefetched are requested and loaded by the browser even before the user requested the resource. When the browser is idle and calculates that it's got enough bandwidth, it will make a request in order to load the resource, and cache it. Having the resource cached can reduce the loading time significantly, as we don't have to wait for the request to finish after the user has clicked the button. It can simply get the loaded resource from cache.

Although prefetching is a great way to optimize the loading time, don't overdo it. If the user ended up never requesting the EmojiPicker component, we unnecessarily loaded the resource. This could potentially cost a user money, or slow down the application. Only prefetch the necessary resources.

List Virtualization

Optimize list performance with list virtualization

In this guide, we will discuss list virtualization (also known as windowing). This is the idea of rendering only visible rows of content in a dynamic list instead of the entire list. The rows rendered are only a small subset of the full list with what is visible (the window) moving as the user scrolls. This can improve rendering performance.

If you use React and need to display large lists of data efficiently, you may be familiar with react-virtualized. It's a windowing library by Brian Vaughn that renders only the items currently visible in a list (within a scrolling "viewport"). This means you don't need to pay the cost of thousands of rows of data being rendered at once. A video walkthrough of list virtualization with react-window accompanies this write-up.

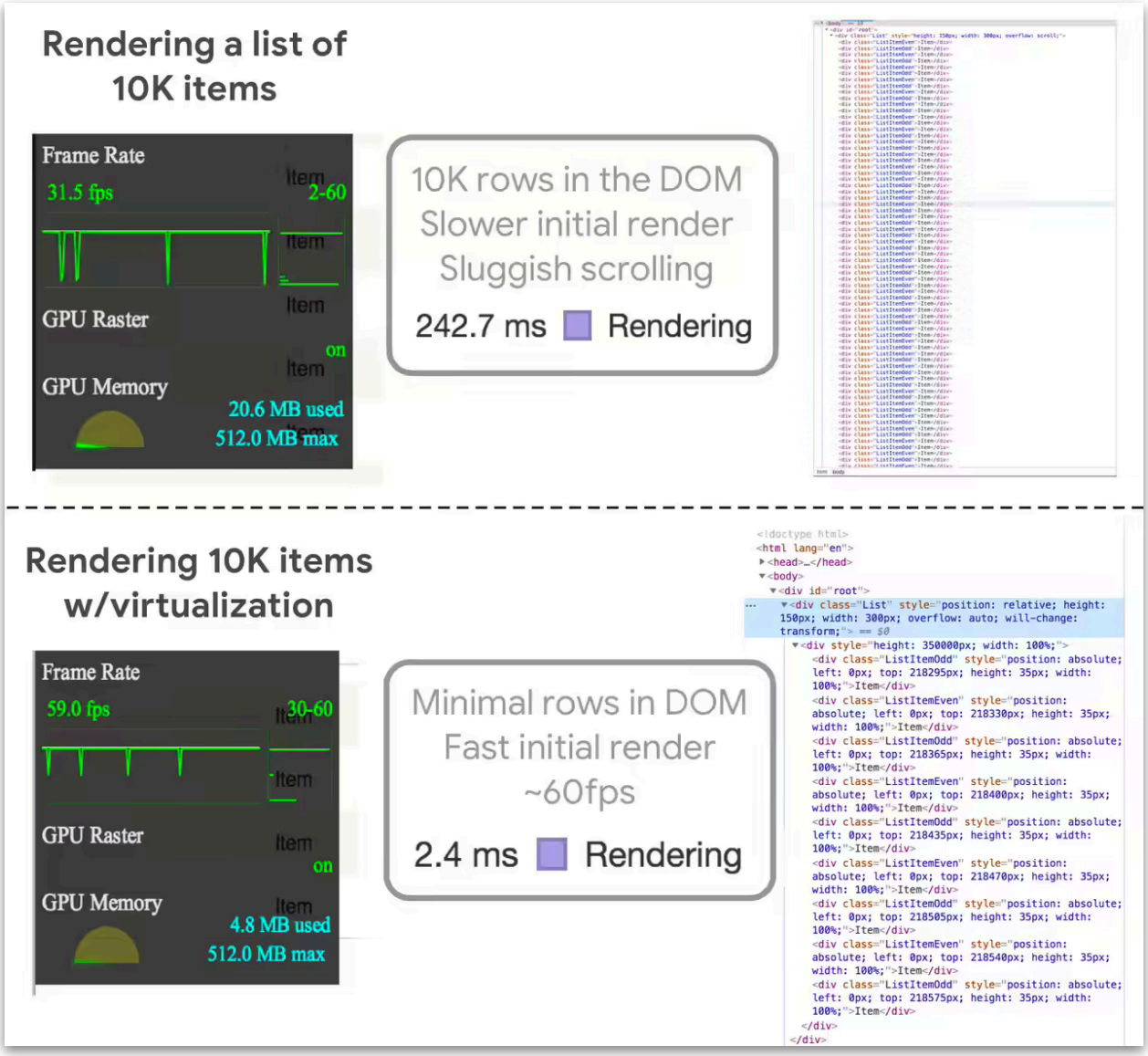
How does list virtualization work?

"Virtualizing" a list of items involves maintaining a window and moving that window around your list. Windowing in react-virtualized works by:

- Having a small container DOM element (e.g <ul>) with relative positioning (window)
- Having a big DOM element for scrolling

- Absolutely positioning children inside the container, setting their styles for top, left, width and height.

Rather than rendering 1000s of elements from a list at once (which can cause slower initial rendering or impact scroll performance), virtualization focuses on rendering just items visible to the user.

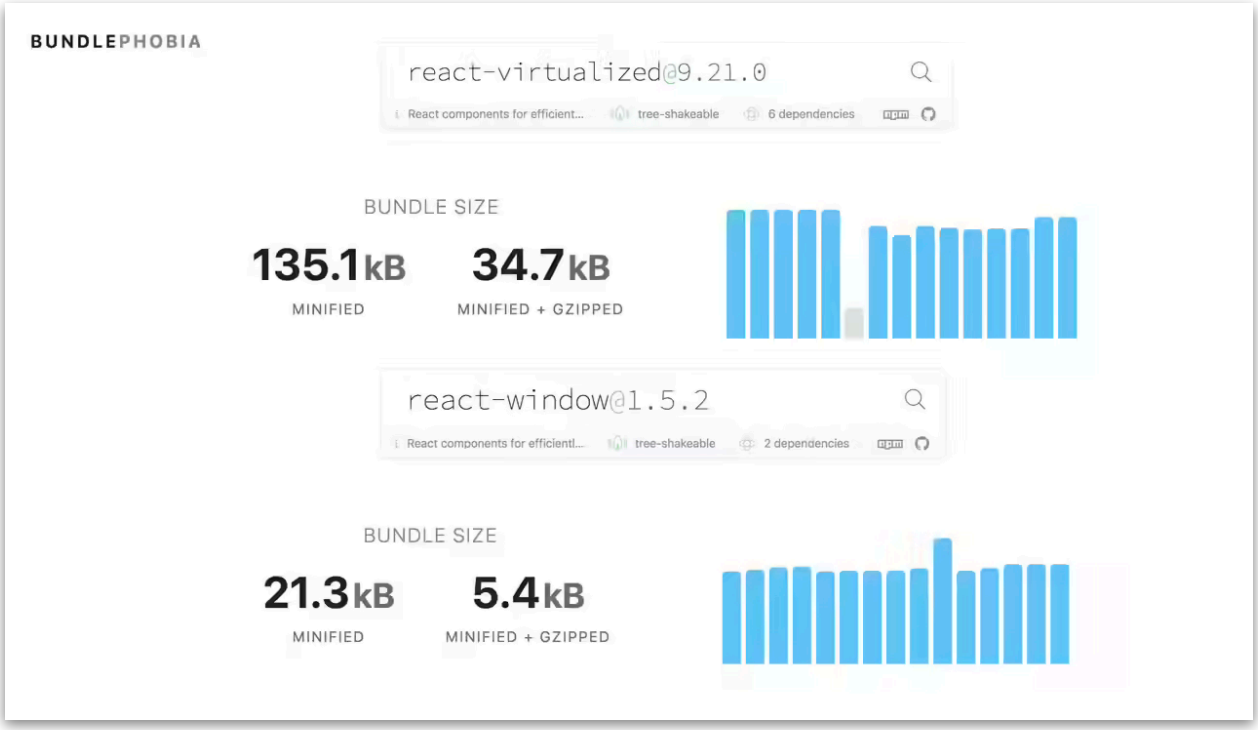


This can help keep list rendering fast on mid to low-end devices. You can fetch/display more items as the user scrolls, unloading previous entries and

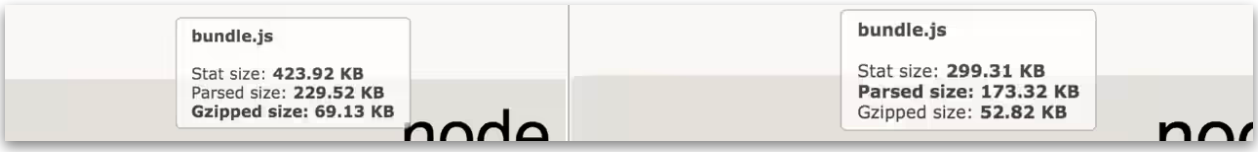
replacing them with new ones.

A smaller alternative to react-virtualized

`react-window` is a rewrite of `react-virtualized` by the same author aiming to be smaller, faster and more tree-shakeable.



In a tree-shakeable library, size is a function of which API surfaces you

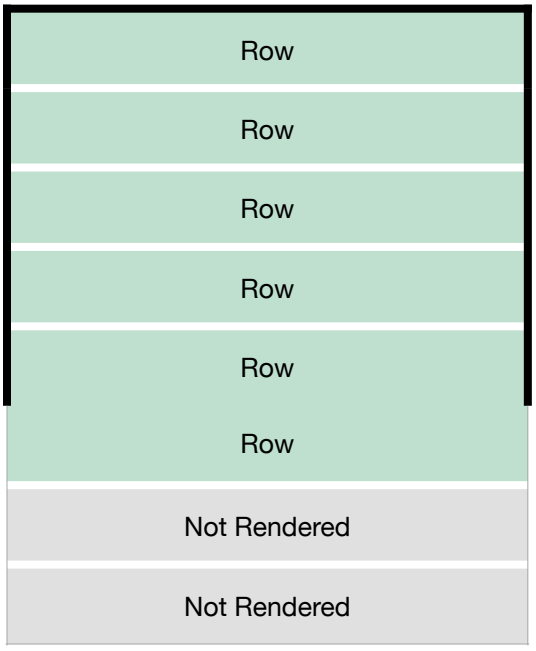


choose to use. I've seen `~20–30KB` (gzipped) savings using it in place of `react-virtualized`:

The APIs for both packages are similar and where they differ, react-window tends to be simpler. react-window's components include:

List

Lists render a windowed list (row) of elements meaning that only the visible rows are displayed to users (e.g `FixedSizeList`, `VariableSizeList`). Lists use a `Grid` (internally) to render rows, relaying props to that inner `Grid`.



Rendering a list of data using React

Here's an example of rendering a list of simple data (`itemsArray`) using React:

```
import React from "react";
import ReactDOM from "react-dom";

const itemsArray = [
  { name: "Drake" },
  { name: "Halsey" },
  { name: "Camillo Cabello" },
  { name: "Travis Scott" },
  { name: "Bazzi" }
];

const Row = ({ index, style }) => (
  <div className={index % 2 ? "ListItemOdd" : "ListItemEven"} style={style}>
    {itemsArray[index].name}
  </div>
);

const Example = () => (
  <div className="List">
    {itemsArray.map((item, index) => Row({ index }))}
  </div>
);

ReactDOM.render(<Example />, document.getElementById("root"));
```

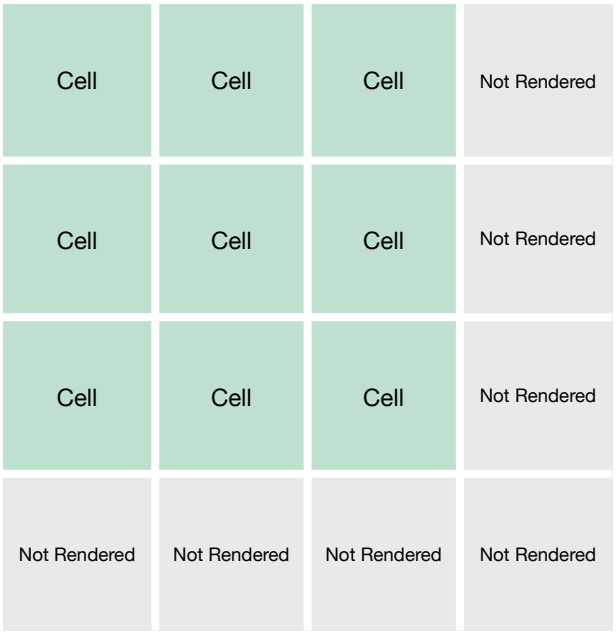
Rendering a list using `react-window`

...and here's the same example using react-window's FixedSizeList, which takes a few props (`width, height, itemCount, itemSize`) and a row rendering function passed as a child:

```
import React from "react";
import ReactDOM from "react-dom";
import { FixedSizeList as List } from "react-window";

const itemsArray = [...]; // our data

const Row = ({ index, style }) => (
  <div className={index % 2 ? "ListItemOdd" : "ListItemEven"} style={style}>
    {itemsArray[index].name}
  </div>
);

const Example = () => (
  <List
    className="List"
    height={150}
    itemCount={itemsArray.length}
    itemSize={35}
    width={300}
  >
    {Row}
  </List>
);

ReactDOM.render(<Example />, document.getElementById("root"));
```

Grid

Grid renders tabular data with virtualization along the vertical and horizontal axes (e.g `FizedSizeGrid`, `VariableSizeGrid`). It only renders the `Grid` cells needed to fill itself based on current horizontal/vertical scroll positions.



If we wanted to render the same list as earlier with a grid layout, assuming our input is a multi-dimensional array, we could accomplish this using `FixedSizeGrid` as follows:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { FixedSizeGrid as Grid } from 'react-window';

const itemsArray = [
  [{},{},{},...],
  [{},{},{},...],
  [{},{},{},...],
  [{},{},{},...],
];

const Cell = ({ columnIndex, rowIndex, style }) => (
  <div
    className={
      columnIndex % 2
        ? rowIndex % 2 === 0
          ? 'GridItemOdd'
          : 'GridItemEven'
        : rowIndex % 2
          ? 'GridItemOdd'
          : 'GridItemEven'
    }
    style={style}
  >
    {itemsArray[rowIndex][columnIndex].name}
  </div>
);

const Example = () => (
  <Grid
    className="Grid"
    columnCount={5}
    columnWidth={100}
    height={150}
    rowCount={5}
    rowHeight={35}
    width={300}
  >
    {Cell}
  </Grid>
);

ReactDOM.render(<Example />, document.getElementById('root'));
```

More in-depth react-window examples

Scott Taylor implemented an open-source Pitchfork music reviews scraper (src) using `react-window` and `FixedSizeGrid`.

Pitchfork scraper uses `react-window-infinite-loader` (demo) which helps break large data sets down into chunks that can be loaded as they are scrolled into view. Here's a snippet of how `react-window-infinite-loader` is incorporated in this app:

```
import React, { Component } from 'react';
import { FixedSizeGrid as Grid } from 'react-window';
import InfiniteLoader from 'react-window-infinite-loader';
...
  render() {
    return (
      <InfiniteLoader
        isItemLoaded={this.isItemLoaded}
        loadMoreItems={this.loadMoreItems}
        itemCount={this.state.count + 1}
      >
        {({ onItemsRendered, ref }) => (
          <Grid
            onItemsRendered={this.onItemsRendered(onItemsRendered)}
            columnCount={COLUMN_SIZE}
            columnWidth={180}
            height={800}
            rowCount={Math.max(this.state.count / COLUMN_SIZE)}
            rowHeight={220}
            width={1024}
            ref={ref}
          >
            {this.renderCell}
          </Grid>
        )}
      </InfiniteLoader>
    );
  }
}
```

What if we have even more complex needs for a grid virtualization solution? We found a The Movie Database demo app that used react-virtualized and Infinite Loader under the hood.

```
return (
  <InfiniteLoader
    isItemLoaded={this.isItemLoaded}
    loadMoreItems={this.loadMoreItems}
    itemCount={this.state.count}
  >
    {({ onItemsRendered, ref }) => (
      <section>
        <FixedSizeList
        itemCount={this.state.count}
        itemSize={ROW_HEIGHT}
        onItemsRendered={onItemsRendered}
        height={this.state.height}
        width={this.state.width}
        ref={ref}
        >
          {this.renderCell}
        </FixedSizeList>
      </section>
    )}
  </InfiniteLoader>
);
```

Porting it over to react-window and react-window-infinite-loader didn't take long, but we did discover a few components were not yet supported. Regardless, the final functionality is pretty close. The missing components were WindowScroller and AutoSizer...which we'll look at next.

```
...
    return (
      <section>
        <AutoSizer disableHeight>
          {({width}) => {
            const {movies, hasMore} = this.props;
            const rowCount = getRowsAmount(width, movies.length, hasMore);
            ...
            return (
              <InfiniteLoader
                ref={this.infiniteLoaderRef}
                ...
                {({onRowsRendered, registerChild}) => (
                  <WindowScroller>
                    {({height, scrollTop}) => (
```

What's missing from `react-window`?

react-window does not yet have the complete API surface of `react-virtualized`, so do check the comparison docs if considering it. What's missing?

- `WindowScroller` - This is a `react-virtualized` component that enables Lists to be scrolled based on the window's scroll positions. There are currently no plans to implement this for react-window so you'll need to solve this in userland.

- `AutoSizer` - HOC that grows to fit all of the available space, automatically adjusting the width and height of a single child. Brian implemented this as a standalone package. Follow this issue for the latest.
- `CellMeasurer` - HOC automatically measuring a cell's content by rendering it in a way that is not visible to the user. Follow here for discussion on support.

That said, we found react-window sufficient for most of our needs with what it includes out of the box.

Improvements in the web platform

Some modern browsers now support CSS content-visibility. `content-visibility:auto` allows you to skip rendering & painting offscreen content until needed. If you have a long HTML document with costly rendering, consider trying the property out.

For rendering lists of dynamic content, I still recommend using a library like react-window. It would be hard to have a `content-visbility: hidden` version of such a library that beats a version aggressively using `display: none` or removing DOM nodes when offscreen like many list virtualization libraries may do today.

Conclusions

That's a wrap for our book! We hope you've enjoyed it as much as we did writing it.

Patterns are time-tested templates for writing code. They can be really powerful, whether you're a seasoned developer or beginner, bringing a valuable level of resilience and flexibility to your codebase.

Keep in mind that patterns are not a silver bullet. Take advantage of them when you have a practical need to solve a problem and when you can use them to write better code. Otherwise, be careful to avoid applying patterns arbitrarily. If a problem you're attempting to solve is just hypothetical, maybe it's premature to consider a pattern.

Always keep simplicity in mind. We try to when evaluating these ideas for the apps we write and hope you will too. Ultimately what works best is often a balance of trade-offs.

Understand if a pattern is helping you achieve your goals; whether it's better user-experience, developer-experience or just smarter architecture. When you have a seasoned knowledge of patterns, you'll appreciate when it may be a good time to use one. Otherwise, study patterns and explore if they may be a good fit for the problem you're attempting to solve. Once you've picked a pattern, make sure you're evaluating the trade-offs of using it. If it looks reasonable, you can use it.

Feel free to share "Learning Patterns" with your friends and colleagues. The book is freely available at Patterns.dev and we welcome any feedback you have. Until next time, so long and good luck, friends!

~ Lydia and Addy