Component based User Interface Rendering with State Caching Between Routes

Jinu Sophia J[1] Naveen Kumar A[2], Nithish S[3], Purushothaman J[4]

[1]*Assistant Professor, Department of Computer Science and Engineering, Rajalakshmi Engineering College, Chennai, Tamilnadu, India .*
[2-4]*Undergraduate student, Computer Science and Engineering, Rajalakshmi Engineering College, Chennai, Tamilnadu, India .*

---***---

Abstract - *The web has transformed itself into a platform that can now run full-fledged application like websites enhancing the user experience and avoiding the hustle of installing bundles to run native applications. This change in the usage of the web as a platform to host applications has made client-side Javascript more the workhorse than it usually was in conventional websites. Routing is a major aspect in making the user experience a more native feel and so came into the limelight the concept of client-side routing. This enables the routing to be taken care of by the client-side Javascript, making the application's execution context remain the same for all the views of the application. The execution context not being disposed of for every action of changing route enables the state of the application to be stored in a cache for later use when the user visits a view that is already in the history of routes. Caching takes a major leap into providing a near native-like application fell as the user gets to come back to the same state of view that they left behind.*

***Key Words*: Component Based UI, State Driven UI, Client-Side Routing, Rendering HTML, Caching State Between Routes**

1. INTRODUCTION

The Web has evolved into a platform that can host applications rather than conventional websites. For long websites were used as a source for transferring information with hyperlinked documents. Then came websites that dynamically showed content based on the user. The user navigated between views of the website by clicking links embedded in the document with anchor tags. A hot reload of the page had to be performed as the browser fetched the HTML(HyperText Markup Language), CSS(Cascading Style Sheet), Javascript and other dependent files from the server. Once fetched the browser then rendered the page and executed the Javascript linked with the page. Hot-reloading of the page results in the application's execution context changing for every view change resulting in any specific application state getting lost once and for all. This incoherent behavior alienates the web application from feeling seamless. Apart from delivering a non-seamless user experience, the HTML, CSS, and Javascript files for a view are fetched from the server every time the link is visited. Network calls involved in fetching these files prove costly and repeated calls within a short period make the network calls a costly redundant work.

The downsides mentioned can be worked around by making the application client-side routed. In client-side routing, the entire application bundle is fetched in the very first request to the server and subsequent HTML to be rendered in the event of a view change is handled by the client-side Javascript by preventing the reload, hence preserving the execution context. This opens up the possibility to cache application state.

1.1 Server-Side Routing

In server-side rendering/ server-side routing, the client(browser) requests the server with the URL(Uniform Resource Locator) that the user wants to visit. The server parses the URL for the path name of the route requested and finds the match with the routes registered in the server. The server now responds back with the HTML, CSS, Javascript and other dependent files.

In dynamic websites that show different content depending on the user interacting with the website, the HTML built by the server using a template engine is shipped to the browser. The initial load time of a server-side routed web page is fast compared to a client-side routed website as the documents corresponding only to the requested route are shipped. Server-side routing gets the upper hand in First Paint, First Contentful Paint, and Time to Interactive. Server-side routing has a clear upper hand in making websites easy to scrap so that search engines can readily index the websites.

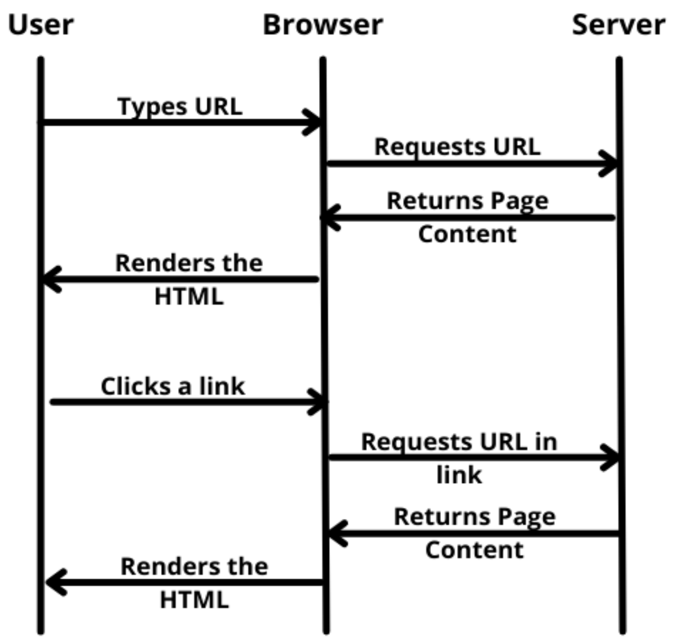


Fig -1: Server-Side Routing

1.2 Client-Side Routing

In client-side routing/ client-side rendering, the client(browser) requests the server with the URL that the user wants to visit. In this routing method, only the base URL is taken into consideration by the server. The server responds with the entire application bundle containing all the documents needed for all the views within the application. Irrespective of the route requested by the client, the server's response is the entire application bundle.

Although the initial load time is slow, subsequent route transitions will have swift load times because no request has to be made to the server. Any data coming from an API can be fetched using AJAX(Asynchronous Javascript and XML) calls. AJAX allows browsers to make network calls without performing a reload. The API in most cases responds with dynamic JSON data depending on the user. This data is used to present information in the UI to the user and the HTML is built by the client.

Building HTML in the client minimizes the workload of the server thus increasing its throughput. Also sending only the JSON data needed reduces the overhead of sending the entire document over the network. All these put together provide a seamless and native-application-like feel to the user. These advantages of client-side routing make the compromises made over First Paint, First Contentful Paint, and Time to Interactive justifiable.

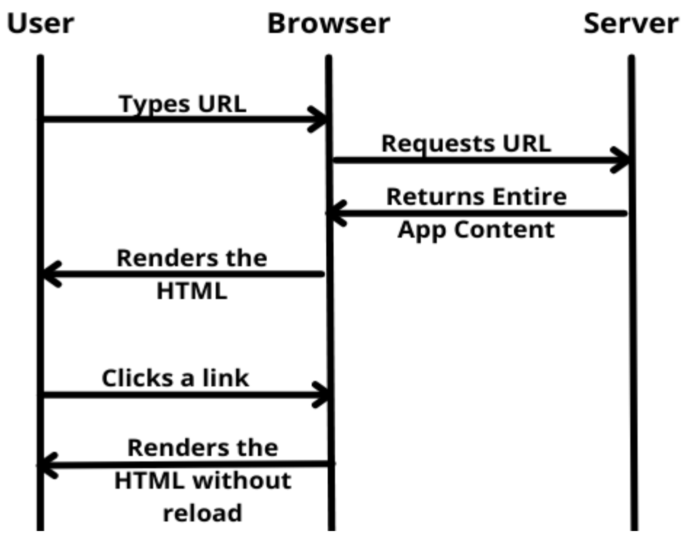


Fig -2: Client-Side Routing

1.3 View Layer

The view layer of the MVC(Model View Controller) model contains the presentable visual aspects of the application shown as UI to the user. The HTML and CSS used to present information to the user qualify as the view in our context.

1.4 UI Components

The User Interface is a complex structure of markup that needs to be constantly monitored for any interactions from the user. This complex structure can be compartmentalized by breaking down the UI into small and individual pieces of markup with the business logic encapsulated into it. Once all the small components are composed, they can be put together to form the complete markup.

The components encapsulated with the business logic can be placed inside a function signature so that the function can be called to return the markup. This makes way for easy reusability of the component, increasing the development time. Each component can be made to listen for a state change and consequently return the state appropriate markup. Application state is a mutable and observable property that controls the behavior of the component. When the state is changed, it causes the entire component to re-run its function and return the markup based upon the current value of the state.

2. VIRTUAL DOM

The DOM(Document Object Model) maps the markup of a page into a Javascript object that can be manipulated to present visual changes in the UI. The DOM contains a tree representation of the HTML along with the properties and styles of the tags. This representation containing methods to manipulate and change the view is exposed to the client-side Javascript by the browser using which the DOM can be manipulated.

A virtual DOM on the other hand is a lightweight, virtual representation of the real DOM in memory created by a library on top of the DOM API provided by the browser. The virtual DOM only has the necessary document structure and the properties associated with it. It does not contain methods to change the view or the styles associated with tags in the document making it a lightweight wrap-around over the real DOM.

3. RENDERING

The virtual DOM contains the UI to be painted on the screen. The contents of the virtual DOM are only for reconciliation purposes and so cannot be directly appended to the DOM or used for rendering. An object of the virtual DOM will contain the type, properties, and its children. This object can be passed to the render method to recursively generate DOM objects for itself and its children so that they can be appended to the real DOM.

Rendering recursively has its pitfall of blocking the main thread until the entire recursive calls are done with their execution. When this happens, Javascript being a single-threaded language, the browser cannot listen for user events or perform other essential work it is entitled to do. This results in the UI becoming unresponsive. To overcome this issue, the object to be rendered is broken down into small pieces, each of which separately can be passed to the render function so that it does not fall into the recursion hell.

Each small piece of the virtual DOM object is called fiber. At the start of the render process, the DOM object for the root fiber is created and its subsequent child is made the next work in progress root. Although this process seems recursive, the use of the browser API requestIdleCallback can be used to schedule work when the browser is free and has no essential task to perform making it safe from recursion hell.

The browser has some essential tasks to perform in the name of calculating style, rasterization, listening for events through event loop, and so on. All of these are vital and any issues in performing these tasks will directly reflect in the application's performance going down. The requestIdleCallback comes in handy to schedule the rendering of UI whenever the browser's main thread is free from performing the essential tasks. This ensures that the application does not become unresponsive during the rendering process.

The DOM tree is rendered bottom-up meaning the children are created first, the parent next and the children are appended to the parent(Fig-2 and Fig - 3). This method of bottom-up rendering helps in including support for fragments and also improves performance as the real DOM's append method is called only once for each render whereas the top-down approach calls the append method for every fiber.

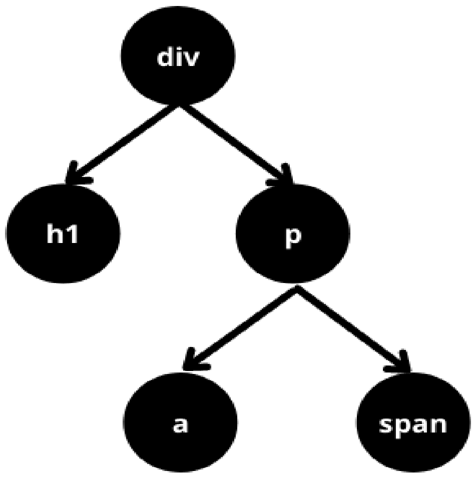
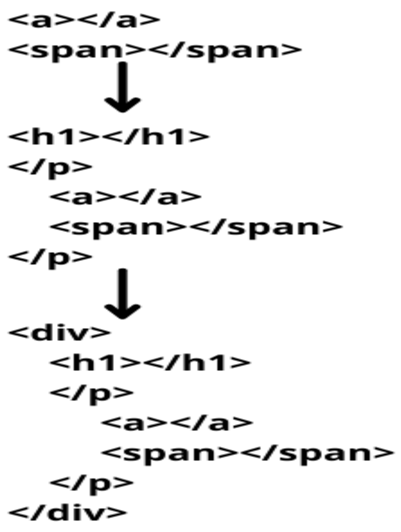


Fig -3: HTML Tree



```
<a></a>
<span></span>
    ↓
<h1></h1>
</p>
  <a></a>
  <span></span>
</p>
    ↓
<div>
  <h1></h1>
  </p>
     <a></a>
     <span></span>
  </p>
</div>
```

Fig -4: Bottom Up Rendering for Tree in Fig - 3

4. RECONCILIATION

Every state change will result in the component returning a different HTML tree compared to the previous tree. The process involved in identifying the differences between the two trees and applying the changes to the DOM is called reconciliation. A direct approach of removing the previous HTML structure and appending the new structure to the DOM is a very costly operation because every manipulation of the DOM will result in the browser recalculating the styles, boxes for every element, and rendering. To have very minimal interaction with the DOM, the virtual DOM is maintained in memory and the newly generated tree is compared with the virtual DOM.

The comparison flags the virtual DOM elements in three ways. Update, if the type remains the same and only the properties are different. Placement, if the UI element is new to the DOM. Remove, if the element is present in the old tree and absent in the new tree. This flagging happens to every element of the virtual DOM as they are traversed once. State-of-the-art algorithms to convert one tree to another take O(n3) time. But the above-mentioned method walks the entire tree only once taking only O(n) time where n is the number of elements.

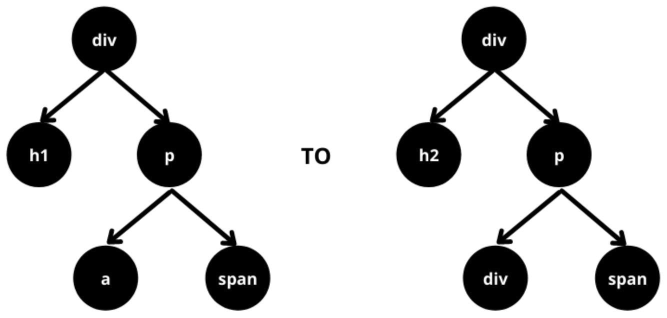


Fig -5: Converting One Tree to Another

5. CACHING STATE

The execution context remains the same with the use of client-side routing for navigating within the application. This can be used to maintain an in-memory cache for the components to subscribe and store their state. With every state change and a fresh run of the component function, the component subscribes its most recent state with the cache. When the user visits an already visited route, the components can use the most recently cached state for its execution. Fetching the state from the cache happens only when an already visited route is navigated to and not in other situations. This provides a native-like-application feel inside of a web browser. JSON data from the server can be stored in state variables so that when cached, the redundant network call to the server again is prevented. The cache is in memory, so a hard refresh of the page flushes the cache.

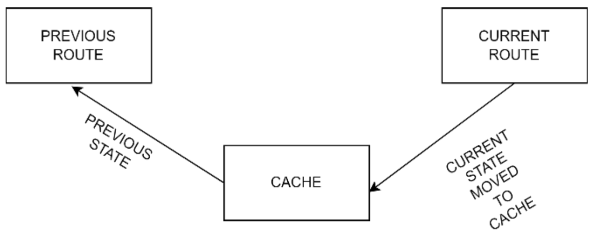


Fig -6: Caching Mechanism

6. CONCLUSION

This method of developing web applications in a declarative, component based, state based, client-routed, and state cached environment takes away the non-seamless nature of web applications and provides a more native-like-application feel without the user having to install bundles to use a native application. Caching also majorly improves performance by preventing redundant calls and provides an ideal user experience by maintaining the application state even after the user has navigated to a different route within the application. Component based development will shorten development time and bugs can be easily mapped and resolved. This approach to developing modern web applications is more efficient and can be adopted easily.

REFERENCES

[1] Shahzad, Farrukh. "Modern and responsive mobile-enabled web applications." Procedia Computer Science 110 (2017): 410-415.

[2] Mukhamadiev, Aidar. "Transitioning from server-side to client-side rendering of the web-based user interface: a performance perspective." (2018).

[3] Aggarwal, Sanchit. "Modern web-development using reactjs." International Journal of Recent Research Aspects 5.1 (2018): 133-137.

[4] Kishore, P., and B. M. Mahendra. "Evolution of Client-Side Rendering over Server-Side Rendering." Recent Trends in Information Technology and its Application 3.2 (2020).

[5] Grundy, John, and John Hosking. "Developing adaptable user interfaces for component-based systems." Interacting with computers 14.3 (2002): 175-194.

[6] Iskandar, Taufan Fadhilah, et al. "Comparison between client-side and server-side rendering in the web development." IOP Conference Series: Materials Science and Engineering. Vol. 801. No. 1. IOP Publishing, 2020.

[7] 7.Chansuwath, Wutthichai, and Twittie Senivongse. "A model-driven development of web applications using AngularJS framework." 2016

[8] Chansuwath, Wutthichai, and Twittie Senivongse. "A model-driven development of web applications using AngularJS framework." 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS). IEEE, 2016.