# **Client-side vs. server-side rendering:**

# **why it’s not all black and white**

[**Cristian Vega**](https://www.freecodecamp.org/news/author/cristianvega/)



Since the dawn of time, the conventional method for getting your HTML up onto a screen was by using server-side rendering. It was the only way. You loaded up your .html pages on your server, then your server went and turned them into useful documents on your users’ browsers.

Server-side rendering worked great at the time too, since most webpages were mostly just for displaying static images and text, with little in the way of interactivity.

Fast-forward to today and that’s no longer the case. You could argue that websites these days are more like applications pretending to be websites. You can use them to send messages, update online information, shop, and so much more. The web is just a whole lot more advanced than it used to be.

So it makes sense that server-side rendering is slowly beginning to take a backseat to the ever-growing method of rendering webpages on the client side.

So which method is the better option? As with most things in development, it really depends on what you’re planning on doing with your website. You need to understand the pros and cons, then decide for yourself which route is best for you.

### **How server-side rendering works**

Server-side rendering is the most common method for displaying information onto the screen. It works by converting HTML files in the server into usable information for the browser.

Whenever you visit a website, your browser makes a request to the server that contains the contents of the website. The request usually only takes a few milliseconds, but that ultimately depends on a multitude of factors:

* Your internet speed
* the location of the server
* how many users are trying to access the site
* and how optimized the website is, to name a few

Once the request is done processing, your browser gets back the fully rendered HTML and displays it on the screen. If you then decide to visit a different page on the website, your browser will once again make another request for the new information. This will occur each and every time you visit a page that your browser does not have a cached version of.

It doesn’t matter if the new page only has a few items that are different than the current page, the browser will ask for the entire new page and will re-render everything from the ground up.

Take for example this HTML document that has been placed in an imaginary server with an HTTP address of example.testsite.com.

<!DOCTYPE html><html> <head> <meta charset="utf-8"> <title>Example Website</title> </head> <body> <h1>My Website</h1> <p>This is an example of my new website</p> <a href="http://example.testsite.com/other.html.">Link</a> </body></html>

If you were to type the address of the example website into the URL of your imaginary browser, your imaginary browser would make a request to the server being used by that URL and expect a response of some text to render onto the browser. In this case, what you would visually see would be the title, the paragraph content and the link.

Now, assume that you wanted to click on the link from the rendered page which contains the following code.

<!DOCTYPE html><html> <head> <meta charset="utf-8"> <title>Example Website</title> </head> <body> <h1>My Website</h1> <p>This is an example of my new website</p> <p>This is some more content from the other.html</p> </body></html>

The only difference between the previous page and this one is that this page does not have a link and instead has another paragraph. Logic would dictate that only the new content should be rendered and the rest should be left alone. Alas, that isn’t how server-side rendering works. What would actually happen would be that the entire new page would be rendered, and not just the new content.

While it might not seem like a big deal for these two examples, most websites are not this simple. Modern websites have hundreds of lines of code and are much more complex. Now imagine browsing a webpage and having to wait for each and every page to render when navigating the site. If you have ever visited a WordPress site, you have seen how slow they can be. This is one of the reasons why.

On the bright side, server-side rendering is great for SEO. Your content is present before you get it, so search engines are able to index it and crawl it just fine. Something that is not so with client-side rendering. At least not simply.

### **How client-side rendering works**

When developers talk about client-side rendering, they’re talking about rendering content in the browser using JavaScript. So instead of getting all of the content from the HTML document itself, you are getting a bare-bones HTML document with a JavaScript file that will render the rest of the site using the browser.

This is a relatively new approach to rendering websites, and it didn't really become popular until JavaScript libraries started incorporating it into their style of development. Some notable examples are Vue.js and React.js, which I’ve [written more about here](https://medium.freecodecamp.com/reacts-jsx-vs-vue-s-templates-a-showdown-on-the-front-end-b00a70470409#.ycvoyji7a).

Going back to the previous website, example.testsite.com, assume that you now have an index.html file with the following lines of code.

<!DOCTYPE html><html><head> <meta charset="utf-8"> <title>Example Website</title></head><body> <div id="root"> <app></app> </div> <script src="https://vuejs.org"type="text/javascript"></script> <script src="location/of/app.js"type="text/javascript"></script></body></html>

You can see right away that there are some major changes to the way the index.hmtl works when rendering using the client.

For starters, instead of having the content inside the HTML file, you have a container div with an id of root. You also have two script elements right above the closing body tag. One that will load the Vue.js JavaScript library and one that will load a file called app.js.

This is radically different than using server-side rendering because the server is now only responsible for loading the bare minus of the website. The main boilerplate. Everything else is handled by a client-side JavaScript library, in this case, Vue.js, and custom JavaScript code.

If you were to make a request to the URL with only the code above, you would get a blank screen. There is nothing to load since the actual content needs to be rendered using JavaScript.

To fix that, you would place the following lines of code into the app.js file.

var data = { title:"My Website", message:"This is an example of my new website" } Vue.component('app', { template: ` <div> <h1>{{title}}</h1> <p id="moreContent">{{message}}</p> <a v-on:click='newContent'>Link</a> </div> `, data: function() { return data; }, methods:{ newContent: function(){ var node = document.createElement('p'); var textNode = document.createTextNode('This is some more content from the other.html'); node.appendChild(textNode); document.getElementById('moreContent').appendChild(node); } } }) new Vue({ el: '#root', });

Now if you visit the URL, you would see the same content as you did the server-side example. The key difference is that if you were to click on the link the page to load more content, the browser will not make another request to the server. You are rendering items with the browser, so it will instead use JavaScript to load the new content and Vue.js will make sure that only the new content is rendered. Everything else will be left alone.

This is much faster since you are only loading a very small section of the page to fetch the new content, instead of loading the entire page.

There are some trade offs with using client-side rendering, though. Since the content is not rendered until the page is loaded on the browser, SEO for the website will take a hit. There are ways to get around this, but it’s not as easy as it is with server-side rendering.

Another thing to keep in mind is that your website/application won’t be able to load until ALL of the JavaScript is downloaded to the browser. Which makes sense, since it contains all the content that will be needed. If your users are using slow internet connection, it could make their initial loading time a bit long.

### **The pros and cons of each approach**

So there you have it. Those are the main differences between server-side and client-side rendering. Only you the developer can decide which option is best for your website or application.

Below is a quick breakdown of the pros and cons for each approach:

#### **Server-side pros:**

1. Search engines can crawl the site for better SEO.
2. The initial page load is faster.
3. Great for static sites.

#### **Server-side cons:**

1. Frequent server requests.
2. An overall slow page rendering.
3. Full page reloads.
4. Non-rich site interactions.

#### **Client-side pros:**

1. Rich site interactions
2. Fast website rendering after the initial load.
3. Great for web applications.
4. Robust selection of JavaScript libraries.

#### **Client-side cons:**

1. Low SEO if not implemented correctly.
2. Initial load might require more time.
3. In most cases, requires an external library.

## **1. REAL DOM**

What does DOM stand for? DOM is an acronym for “Document Object Model” and is the user interface UI of the application. Every time change of state or property occurs in the application UI, the DOM is updated via a batch update to show the changes occurring. However, frequent manipulations to the DOM are detrimental to system performance making its responses slow.

## **2. VIRTUAL DOM**

What is a virtual DOM? To avoid this situation, the virtual DOM concept is very handy and its performance much more effective than the real DOM suffering frequent manipulations of property or state. It is just a virtual representation of the real DOM and hence called virtual DOM.

Thus, whenever a state of change occurs in the application, the virtual DOM in React reads, compares with the older version and gets updated instead of directly affecting changes to the real DOM. The real DOM is then updated using a batch update from the virtual DOM keeping the process more efficient, fast and free of slowing down of the real DOM.

## **3. HOW IS VIRTUAL DOM FASTER?**

Firstly, what is virtual DOM and what causes the DOM manipulation to slow it down? The answer is that the DOM has a tree structure and is represented visually in computer language as a tree data structure. Hence changes occurring are updated across the DOM structure very rapidly. However, these changes need to be repainted or re-applied to the application UI and all children elements by the DOM which slows it down.

The more the number of UI components connected the longer it takes and gets more expensive to render an update. Also, note that every update or change of property/ state needs to re-render across all UI components and the application itself each time a change occurs. Consider the addition of a new element like the virtual DOM when it is added to the UI. A tree data-structure is created with a virtual DOM, where each element on it is represented as a tree-node. When the state of any of these node elements changes, the virtual DOM tree is also changed and the virtual tree ‘diffs’ or compares the changes to the previous state of the virtual DOM tree.

The virtual DOM React then calculates the best method using the React algorithm to affect the change to the real DOM and ensures the use of minimal resources and operations. Once done the virtual DOM in React sends this information to the real Dom in what is called a batch update which the real DOM uses to repaint all the elements and UI application.

## **4. HOW DOES REACT USE VIRTUAL DOM?**

So, how does ReactJS work and leverage the virtual DOM? What is react DOM? In JSReact, every component is a UI piece and each component will have a state at any given point in time.

React listens for state changes and follows the observable pattern changes. Whenever the component’s state or properties change, React updates the tree representing the virtual DOM in React. The virtual DOM when updated causes React to compare the changes and the current virtual DOM version to the previous virtual DOM version in a process known as “diffing” or differentiating.

Once React is aware of the changes to the virtual DOM in React objects whose states have changed, it updates only those changed objects for re-painting in the real DOM. This makes unnecessary and frequent changes of all elements unnecessary and the performance improves vastly when compared to affecting direct changes to the real DOM. far better when compared to changing or manipulating the real DOM directly. Also, it saves time in affecting the changes only to the changed elements and not all components. This then makes React, the JavaScript library’s performance more efficient and praise-worthy in terms of efficiency.

Thus, one just needs to inform React about the state one wants the virtual DOM in React and the UI to be in and it will make sure that the real DOM applies changes and matches that state. Hence, developers need not learn manual DOM updates, attribute manipulation or event handling as all these changes are affected automatically and behind the scenes. Given below are how to affect these changes in React.

# React State

State is **a plain JavaScript object used by React to represent an information about the component's current situation**.

React components has a built-in state object.

The state object is where you store property values that belongs to the component.

When the state object changes, the component re-renders.

## **When to use Redux**

Lately one of the biggest debates in the frontend world has been about Redux. Not long after its release, Redux became one of the hottest topics of discussion. Many favored it while others pointed out issues.

Redux allows you to manage your app’s state in a single place and keep changes in your app more predictable and traceable. It makes it easier to reason about changes occurring in your app. But all of these benefits come with tradeoffs and constraints. One might feel it adds up boilerplate code, making simple things a little overwhelming; but that depends upon the architecture decisions.

One simple answer to this question is you will realize for yourself when you need Redux. If you’re still confused as to whether you need it, you don’t. This usually happens when your app grows to the scale where managing app state becomes a hassle; and you start looking out for making it easy and simple.

## **What is state management in Redux?**

State management is essentially a way to facilitate communication and sharing of data across components. It creates a tangible data structure to represent the state of your app that you can read from and write to. That way, you can see otherwise invisible states while you’re working with them.

Most libraries, such as React, Angular, etc. are built with a way for components to internally manage their state without any need for an external library or tool. It does well for applications with few components, but as the application grows bigger, managing states shared across components becomes a chore.

In an app where data is shared among components, it might be confusing to actually know where a state should live. Ideally, the data in a component should live in just one component, so sharing data among sibling components becomes difficult.

For instance, in React, to share data among siblings, a state has to live in the parent component. A method for updating this state is provided by the parent component and passed as props to these sibling components.

Here’s a simple example of a login component in React. The input of the login component affects what is displayed by its sibling component, the status component:

class App extends React.Component { constructor(props) { super(props); // First the Parent creates a state for what will be passed this.state = { userStatus: "NOT LOGGED IN"} this.setStatus = this.setStatus.bind(this); } // A method is provided for the child component to update the state of the // userStatus setStatus(username, password) { const newUsers = users; newUsers.map(user => { if (user.username == username && user.password === password) { this.setState({ userStatus: "LOGGED IN" }) } }); } render() { return ( <div> // the state is passed to the sibling as a props as is updated whenever // the child component changes the input <Status status={this.state.userStatus} /> // this method is passed to the child component as a props which it // uses to change the state of the userStatus <Login handleSubmit={this.setStatus} /> </div> ); }});

U,{a118164c-fe93-4d50-af71-bed845e0926f}{74},3.125,3.125 *Remember, this data is not needed by the parent component, but because its children need to share data, it has to provide a state.*

Now imagine what happens when a state has to be shared between components that are far apart in the component tree. The state has to be passed from one component to another until it gets to where it is needed.

Basically, the state will have to be lifted up to the nearest parent component and to the next until it gets to an ancestor that is common to both components that need the state, and then it is passed down. This makes the state difficult to maintain and less predictable. It also means passing data to components that do not need it.

It’s clear that state management gets messy as the app gets complex. This is why you need a state management tool like Redux that makes it easier to maintain these states. Let’s get a good overview of Redux concepts before considering its benefits.

## **How Redux works**

The way Redux works is simple. There is a central store that holds the entire state of the application. Each component can access the stored state without having to send down props from one component to another.

There are three building parts: actions, store, and reducers. Let’s briefly discuss what each of them does. This is important because they help you understand the benefits of Redux and how it’s to be used. We’ll be implementing a similar example to the login component above but this time in Redux.

### **Actions in Redux**

Simply put, actions are events. They are the only way you can send data from your application to your Redux store. The data can be from user interactions, API calls, or even form submissions.

Actions are sent using the store.dispatch() method. Actions are plain JavaScript objects, and they must have a type property to indicate the type of action to be carried out. They must also have a payload that contains the information that should be worked on by the action. Actions are created via an action creator.

Here’s an example of an action that can be carried out during login in an app:

{ type: "LOGIN", payload: { username: "foo", password: "bar" }}

Here is an example of its action creator:

const setLoginStatus = (name, password) => { return { type: "LOGIN", payload: { username: "foo", password: "bar" } }}

As explained earlier, the action must contain the type property and then the other payload to be stored.

### **Reducers in Redux**

Reducers are pure functions that take the current state of an application, perform an action, and return a new state. These states are stored as objects, and they specify how the state of an application changes in response to an action sent to the store.

It is based on the reduce function in JavaScript, where a single value is calculated from multiple values after a callback function has been carried out.

Here is an example of how reducers work in Redux:

const LoginComponent = (state = initialState, action) => { switch (action.type) { // This reducer handles any action with type "LOGIN" case "LOGIN": return state.map(user => { if (user.username !== action.username) { return user; } if (user.password == action.password) { return { ...user, login\_status: "LOGGED IN" } } });default: return state; } };

U,{a118164c-fe93-4d50-af71-bed845e0926f}{241},3.125,3.125 *Reducers take the previous state of the app and return a new state based on the action passed to it.*

As pure functions, they do not change the data in the object passed to them or perform any side effect in the application. Given the same object, they should always produce the same result.

### **Store in Redux**

The store holds the application state. It is highly recommended to keep only one store in any Redux application. You can access the state stored, update the state, and register or unregister listeners via helper methods.

Let’s create a store for our login app:

const store = createStore(LoginComponent);

Actions performed on the state always return a new state. Thus, the state is very easy and predictable.

Now that we know a little more about Redux, let’s go back to our login component example that was implemented earlier and see how Redux can improve the component.

class App extends React.Component { render() { return ( <div> <Status user={this.props.user.name}/> <Login login={this.props.setLoginStatus}/> </div> ) }}

With Redux, there’s one general state in the store, and each component has access to the state. This eliminates the need to continuously pass state from one component to another. You can also select the slice from the store for a particular component; this makes your app more optimized.

## **Redux middleware**

Redux allows developers to intercept all actions dispatched from components before they are passed to the reducer function. This interception is done via middlewares.

Building on the example Login component discussed in the last section, we might want to sanitize the user’s input before it reaches our store for further processing. This can be achieved via [Redux middleware](https://redux.js.org/understanding/history-and-design/middleware).

Technically, middlewares are functions that call the next method received in an argument after processing the current action. These are called after every dispatch.

Here’s what a simple middleware looks like:

function simpleMiddleware({ getState, dispatch }) { return function(next){ return function(action){ // processing const nextAction = next(action); // read the next state const state = getState(); // return the next action or you can dispatch any other action return nextAction; } }}

This might look a little overwhelming, but in most cases, you might not need to create your own middleware since the huge Redux community has already made a number of them available. If you feel middleware is required, you will enjoy it because it gives you a lot of power to do tons of great work with the best abstraction.

## **Why use Redux?**

When using Redux with React, states will no longer need to be lifted up. This makes it easier for you to trace which action causes any change.

As you can see in the example above, the component does not need to provide any state or method for its children components to share data among themselves. Everything is handled by Redux. This greatly simplifies the app and makes it easier to maintain.

This is the primary reason why you should use Redux, but it’s not the only benefit. Take a look at the list below for a summary of what you stand to gain by using Redux for state management.

### **Redux makes the state predictable**

In Redux, the state is always predictable. If the same state and action are passed to a reducer, the same result is always produced because reducers are pure functions. The state is also immutable and is never changed. This makes it possible to implement arduous tasks like infinite undo and redo. It is also possible to implement time travel — that is, the ability to move back and forth among the previous states and view the results in real time.

### **Redux is maintainable**

Redux is strict about how code should be organized, which makes it easier for someone with knowledge of Redux to understand the structure of any Redux application. This generally makes it easier to maintain. This also helps you segregate your business logic from your component tree. For large scale apps, it’s critical to [keep your app more predictable and maintainable.](https://buttercms.com/blog/react-best-practices-maintaining-large-scale-projects)

### **Debugging is easy in Redux**

Redux makes it easy to debug an application. By logging actions and state, it is easy to understand coding errors, network errors, and other forms of bugs that might come up during production.

Besides logging, it has [great DevTools that allow you to time-travel actions](https://blog.logrocket.com/redux-devtools-tips-tricks-for-faster-debugging/), persist actions on page refresh, etc. For medium- and large-scale apps, debugging takes more time then actually developing features. Redux DevTools makes it easy to taker advantage of all Redux has to offer.

### **Performance Benefits**

You might assume that keeping the app’s state global would result in some performance degradation. To a large extent, that’s not the case.

React Redux implements many performance optimizations internally so that your own connected component only rerenders when it actually needs to.

### **Ease of testing**

It is easy to test Redux apps since functions used to change the state of pure functions.

### **State persistence**

You can persist some of the app’s state to local storage and restore it after a refresh. This can be really nifty.

### **Server-side rendering**

Redux can also be used for server-side rendering. With it, you can handle the initial render of the app by sending the state of an app to the server along with its response to the server request. The required components are then rendered in HTML and sent to the clients.

## **Implementing Redux in your app? Track Redux state and actions with LogRocket**

Debugging React applications can be difficult, especially when there is complex state. If you’re interested in monitoring and tracking Redux state for all of your users in production, [try LogRocket](https://logrocket.com/signup/).