SWR

React Hooks for Remote Data Fetching

SWR is created by the same team(@vercel) behind Next.js, the React framework.

Phurunbit 2020.12

1. Prequerisites of lesson

We assume that you are good with

- React.js

- React Hooks

- Redux or Mobx for State Management

2. Why do we need SWR? What is the advantage to learn this?

1) What is Redux and how it is used?

Redux is used mostly for application state management. To summarize it, Redux maintains the state of an entire application in a single immutable state tree (object), which can't be changed directly.

Redux is a predictable state container for JavaScript apps, and it allows you to manage state for your web applications built in any JavaScript framework such as React.

2) What is Mobx and how to use?

MobX is a simple, scalable and battle tested state management solution.

The observer HoC automatically subscribes React components to any observables that are used during rendering. As a result, components will automatically re-render when relevant observables change.

3) Why did I stop using Redux?

If we use redux + react. the usual flow is,

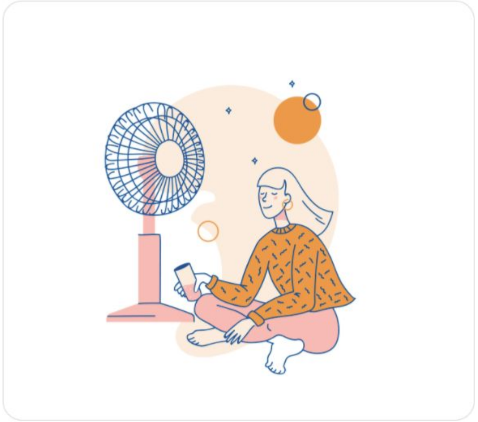
1) dispatch action -> redux saga or redux thunk -> api call
2) set the response data to global sate
3) get the state as props in the target component
This is boring job...

But!
If we use swr, there’s no need these steps any more.
We just have to call useUser() in every component where we need.

- No worries about multiple duplicated request,
- No need to handle global state or context any more.
- No need all that kind of annoying actions, constants, reducers, sagas, thunk and etc redux family.

4) Can we replace Redux with SWR totally?

Yes, we can.



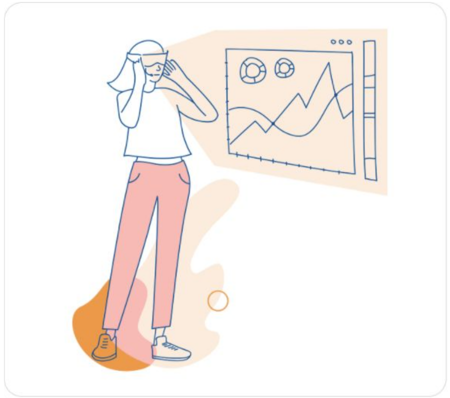
5) What are the advantages of SWR?

1) With SWR, the UI will be always fast and reactive.
   The components will get a stream of data updates constantly and automatically.

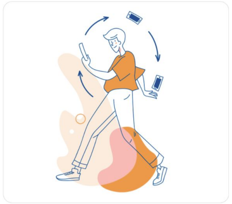
2) With just one single line of code, you can simplify the logic of data fetching in your project.

3. Agenda

1. Introduction & Features
2. Real-world example
3. Options
4. Global Configuration
5. Data Fetching
6. Error Handling
7. Auto Revalidation
8. Conditional Data Fetching
9. Arguments
10. Mutation
11. Pagination
12. Prefetching
13. Usage with next.js
14. Advanced - Performance



Introduction



SWR is React Hooks library for data fetching.

The name “SWR” is derived from stale-while-revalidate, a HTTP cache invalidation strategy popularized by HTTP RFC 5861. SWR is a strategy to first return the data from cache (stale), then send the fetch request (revalidate), and finally come with the up-to-date data.

With SWR, components will get a stream of data updates constantly and automatically.

And the UI will be always fast and reactive.

Lightweight

Backend Agnostic

Realtime

Jamstack Oriented

TypeScript Ready

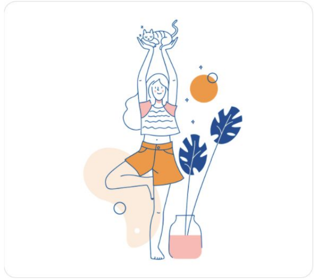
Remote + Local

```
import useSWR from 'swr'

function Profile() {
 const { data, error } = useSWR('/api/user', fetcher)
 if (error) return <div>failed to load</div>
 if (!data) return <div>loading...</div>
 return <div>hello {data.name}!</div>
}
```

In this example, the useSWR hook accepts a key string and a fetcher function. key is a unique identifier of the data (normally the API URL) and will be passed to fetcher. fetcher can be any asynchronous function which returns the data, you can use the native fetch or tools like Axios.

The hook returns 2 values: data and error, based on the status of the request.



Features - 1

With just one single line of code, you can simplify the logic of data fetching in your project, and also have all these amazing features out-of-the-box:

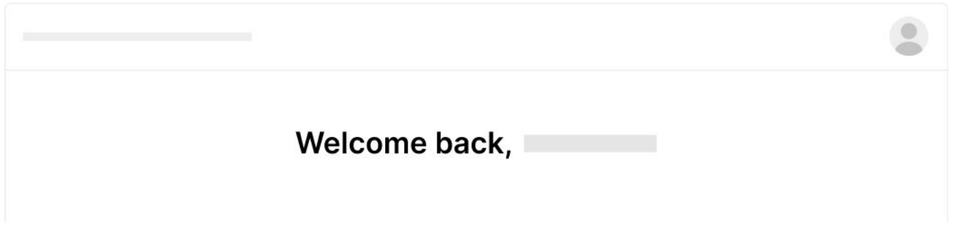
- Fast, lightweight and reusable data fetching
- Built-in cache and request deduplication
- Real-time experience
- TypeScript ready
- React Native

Features - 2

SWR has you covered in all aspects of speed, correctness, and stability to help you build better experiences:

- Fast page navigation
- Polling on interval
- Data dependency
- Revalidation on focus
- Revalidation on network recovery
- Local mutation (Optimistic UI)
- Smart error retry
- Pagination and scroll position recovery
- React Suspense

Real-World Example



In a real-world example, our website shows a navbar and the content, both depend on user.

```
// page component
function Page () {
 const [user, setUser] = useState(null)
 // fetch data
 useEffect(() => {
   fetch('/api/user')
     .then(res => res.json())
     .then(data => setUser(data))
 }, [])
 // global loading state
 if (!user) return <Spinner/>
 return <div>
   <Navbar user={user} />
   <Content user={user} />
 </div>
}
// child components
function Navbar ({ user }) {
 return <div>
   ...
   <Avatar user={user} />
 </div>
}
function Content ({ user }) {
 return <h1>Welcome back, {user.name}</h1>
}
function Avatar ({ user }) {
 return <img src={user.avatar} alt={user.name} />
}
```

Real world Example using traditional way.

Traditionally, we fetch data once using useEffect in the top level component, and pass it to child components via props (notice that we don't handle error state for now).

Usually, we need to keep all the data fetching in the top level component and add props to every component deep down the tree. The code will become harder to maintain if we add more data dependency to the page.

Although we can avoid passing props using Context, there's still the dynamic content problem: components inside the page content can be dynamic, and the top level component might not know what data will be needed by its child components.

Real world Example using SWR.

```
function useUser (id) {
 const { data, error } = useSWR(`/api/user/${id}`, fetcher)
 return {
   user: data,
   isLoading: !error && !data,
   isError: error
 }
}

// page component
function Page() {
 return <div>
   <Navbar/>
   <Content/>
 </div>
}

// child components
function Navbar() {
 return <div>
   ...
   <Avatar/>
 </div>
}

function Content() {
 const {user, isLoading} = useUser()
 if (isLoading) return <Spinner/>
 return <h1>Welcome back, {user.name}</h1>
}

function Avatar() {
 const {user, isLoading} = useUser()
 if (isLoading) return <Spinner/>
 return <img src={user.avatar} alt={user.name}/>
}
```

SWR solves the problem perfectly. Data is now bound to the components which need the data, and all components are independent to each other. All the parent components don't need to know anything about the data or passing data around. They just render. The code is much simpler and easier to maintain now.

The most beautiful thing is that there will be only 1 request sent to the API, because they use the same SWR key and the request is deduped, cached and shared automatically.

Also, the application now has the ability to refetch the data on user focus or network reconnect! That means, when the user's laptop wakes from sleep or they switch between browser tabs, the data will be refreshed automatically.

API Options

```
const { data, error, isValidating, mutate } = useSWR(key, fetcher, options)
```

Parameters#

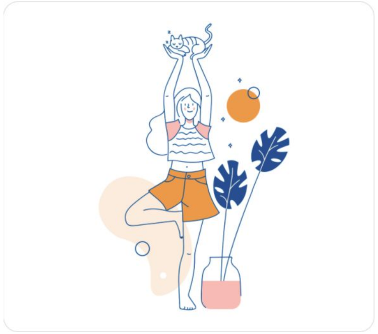
- `key`: a unique key string for the request (or a function / array / null) (advanced usage)
- `fetcher`: (*optional*) a Promise returning function to fetch your data (details)
- `options`: (*optional*) an object of options for this SWR hook

Return Values#

- `data`: data for the given key resolved by `fetcher` (or undefined if not loaded)
- `error`: error thrown by `fetcher` (or undefined)
- `isValidating`: if there's a request or revalidation loading
- `mutate(data?, shouldRevalidate?)` : function to mutate the cached data

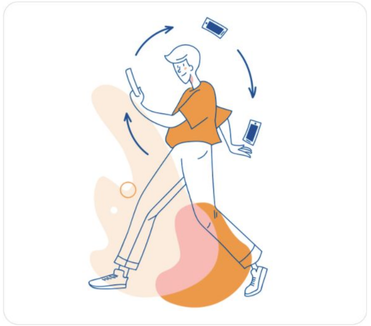
Options#

- `suspense = false` : enable React Suspense mode
- `fetcher = window.fetch(url).then(res => res.json())` : the fetcher function
- `initialData` : initial data to be returned (note: This is per-hook)
- `revalidateOnMount` : enable or disable automatic revalidation when component is mounted (by default revalidation occurs on mount when initialData is not set, use this flag to force behavior)
- `revalidateOnFocus = true` : auto revalidate when window gets focused
- `revalidateOnReconnect = true` : automatically revalidate when the browser regains a network connection (via `navigator.onLine`)
- `refreshInterval = 0` : polling interval (disabled by default)
- `refreshWhenHidden = false` : polling when the window is invisible (if `refreshInterval` is enabled)
- `refreshWhenOffline = false` : polling when the browser is offline (determined by `navigator.onLine`)
- `shouldRetryOnError = true` : retry when fetcher has an error
- `dedupingInterval = 2000` : dedupe requests with the same key in this time span
- `focusThrottleInterval = 5000` : only revalidate once during a time span
- `loadingTimeout = 3000` : timeout to trigger the onLoadingSlow event



Options#

- `errorRetryInterval = 5000` : error retry interval
- `errorRetryCount` : max error retry count
- `onLoadingSlow(key, config)` : callback function when a request takes too long to load (see `loadingTimeout`)
- `onSuccess(data, key, config)` : callback function when a request finishes successfully
- `onError(err, key, config)` : callback function when a request returns an error
- `onErrorRetry(err, key, config, revalidate, revalidateOps)` : handler for error retry
- `compare(a, b)` : comparison function used to detect when returned data has changed, to avoid spurious rerenders. By default, dequal is used.



Global Configuration

```
import useSWR, {SWRConfig} from 'swr'

function Dashboard() {
 const {data: events} = useSWR('/api/events')
 const {data: projects} = useSWR('/api/projects')
 const {data: user} = useSWR('/api/user', {refreshInterval: 0}) // override
 // ...
}

function App() {
 return (
   <SWRConfig
     value={{
       refreshInterval: 3000,
       fetcher: (resource, init) => fetch(resource, init).then(res => res.json())
     }}
   >
     <Dashboard/>
   </SWRConfig>
 )
}
```

In this example, all SWR hooks will use the same fetcher provided to load JSON data, and refresh every 3 seconds by default

Data Fetching

```
/***
*  Data Fetching using Axios
*/
import axios from 'axios'
const fetcher = url => axios.get(url).then(res => res.data)
function App () {
 const { data, error } = useSWR('/api/data', fetcher)
 // ...
}

/***
*  Data Fetching using Graphql
*/
import { request } from 'graphql-request'
const fetcher = query => request('https://api.graph.cool/simple/v1/movies', query)
function App () {
 const { data, error } = useSWR(
   `{
     Movie(title: "Inception") {
       releaseDate
       actors {
         name
       }
     }
   }`,
   fetcher
 ); }
```

Error Handling

```
/**
*  Error Retry
*/
useSWR('/api/user', fetcher, {
 onErrorRetry: (error, key, config, revalidate, { retryCount }) => {
   // Never retry on 404.
   if (error.status === 404) return
   // Never retry for a specific key.
   if (key === '/api/user') return
   // Only retry up to 10 times.
   if (retryCount >= 10) return
   // Retry after 5 seconds.
   setTimeout(() => revalidate({ retryCount: retryCount + 1 }), 5000)
 }
})


/**
*  Global Error Report
*/
<SWRConfig value={{
 onError: (error, key) => {
   if (error.status !== 403 && error.status !== 404) {
     // We can send the error to Sentry,
     // or show a notification UI.
   }
 }
}}>
 <MyApp />
</SWRConfig>
```

Exploring the use case for SWR and if i can use it as a replacement for Redux

- Application state !== remote data from a server. Remote data should be in a cache, application state should be in state (React Context is fine)
- Once you accept the above, you'll stop using Redux to store everything that comes back from the server (i.e. remote data)
- Because you stopped making a client side copy of the server side data, you no longer have to worry about your state becoming stale
- You can access the data from the SWR cache the same way you can access data from a store. Because SWR cache is a global state object. (fyi, server side data is also called server cache)
- You can use Axios with SWR, it will not force you to give it up. As a matter of fact, you can use any library you want for making HTTP requests
- With SWR, you no longer have to worry about manually updating error, loading, success state and write reducers for them. SWR gives you these out of the box
- SWR is only meant for reading data (GET), for other CRUD operations, you can handle them outside SWR
- You can use a mutate function to update SWR cache
- You still need global app state, (be it Redux or React Context), we just don't store remote data in it.

Bandwidth concerns

SWR is great if we're using GraphQL, because then you are not making multiple HTTP calls every time you need data, so all the *revalidating* is *less expensive*. But if you're using plain old REST APIs with lots of HTTP requests, do you wonder how much bandwidth it'll consume and if we can afford making so many calls so frequently?

SWR’s built-in caching and deduplication skips unnecessary network requests, but the performance of the useSWR hook itself still matters. In a complex app, there could be hundreds of useSWR calls in a single page render.

SWR ensures that your app has:

- no unnecessary requests
- no unnecessary re-renders
- no unnecessary code imported

without any code changes from you.

Links

https://swr.vercel.app/
https://dev.to/juliang/managing-remote-data-with-swr-7cf
https://dev.to/juliang/react-state-management-in-2020-3c58
https://medium.com/better-programming/why-you-should-be-separating-your-server-cache-fro
m-your-ui-state-1585a9ae8336
https://dev.to/juliang/how-swr-works-4lkb
https://github.com/vercel/swr/discussions/364
https://www.bitnative.com/2020/07/06/four-ways-to-fetch-data-in-react/
https://sergiodxa.com/articles/swr/storage-sync

Thank you