Intro to Hoodie and React

BY JAKE PEYSER ON JULY 19, 2017
IN WEBDEV, REACT

Let's take a look at Hoodie, the "Back-End as a Service" (BaaS) built specifically for front-end developers. Hoodie is a relatively new, but very promising open-source tool and deserves more exposure among the spectrum of conversations that it gets today. I've put together a demo that demonstrates some of the key features of the service, but I feel the need to first set the scene. In this post I'll cover how to jump-start to the demo repo if you want to get the code. Otherwise, join me for a brief overview.

Setting the Scene

It is no secret that JavaScript is eating the world these days and with its explosion in popularity, an ever-expanding ecosystem of tooling has arisen. The process of developing a web app has skyrocketed in recent years thanks to these tools. Developer tools Prettier and ESLint give us freedom to write how we like and still ensure our code is consistent and error-free. Bundlers and task runners/pipelines for creating productive experiences. Build tools like Webpack and Babel allow us to use the latest and greatest language features and patterns without sacrificing speed and efficiency.

Much of the focus in JavaScript these days seems to be on front-end tools, but it does not mean there is no love to be found on the back-end. This same pattern of automation and abstraction is available on the server side, too, primarily in the form of what we call "Backend as a Service" (BaaS). This model provides a way for front end developers to link their web or mobile apps to backend services without the need to write server code.

Many of these services have been around for awhile, but no real winner has come forth. Parse, one of the early player in the space, was gobbled up by Facebook in 2013 and subsequently shut down. Firebase was acquired by Google and is slowly making improvements to developing realtime share. There are a few weeds-ages, MongoDB announced their own BaaS, Stitch, with hopes of capitalizing on the market penetration of their DB.

BaaS Advantages

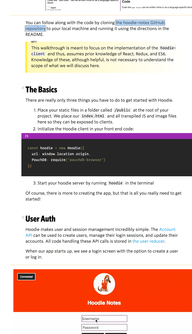
There are an overwhelming number of BaaS options, however, they all have the same primary advantages at their core.

- **Streamlined development**: The obvious advantage of having no custom server is that it removes the need to develop one! This means your development team can perform less context switching and ultimately have more time to focus on core logic. No server language knowledge required!
- **No boilerplate screens**: Many servers end up existing for the sole purpose of connecting a client with relevant data. This often results in massive amounts of work framework and DAL boilerplate code. The BaaS model removes the need for this repetitive code.

These are just the main advantages of BaaS. Hoodie provides these and many more unique capabilities that we will walk through in the next section.

Try on your Hoodie

To demonstrate some of the out-of-the-box functionality provided by Hoodie, I am going to walk you through a few pieces of a simple blogging app. It's a single page web application. It is going to handle user authentication, full CRUD of users notes, and the ability to keep working even when a connection to the internet is lost.



You can follow along with the code by cloning the Hoodie-notes GitHub repository to your local machine and running. It using the directions in the README:

> **Note:** This walkthrough is meant to focus on the implementation of the Hoodie API. As such, it assumes basic knowledge of React, Redux, and ES6. Knowledge of these, although helpful, is not necessary to understand the scope of what we will discuss here.

The Basics

There are really only three things you have to do to get started with Hoodie.

1. Place your static files in a folder called `/public` at the root of your project. We place our `index.html` and all transpiled JS and image files here so they can be exposed to clients.
2. Initialize the Hoodie client in your front end code.

```
$ cd hoodie --> cmd hoodie
$ vi window.location setups
$ hoodie --> require('./assets/dr-browser')
```

3. Start your hoodie server by running `hoodie` in the terminal

Of course, there is more to creating the app, but that is all you really need to get started!

User Auth

Hoodie makes user and session management incredibly simple. The Accounts API can be used to create users, manage their login sessions, and update their accounts. All code handling these API calls is stored in the /src/ folder.

When our app starts up, we see a login screen with the option to create a user or log in.

When either of these buttons are pressed, the corresponding Redux thunk is dispatched to handle the authorization. We use the `signup` and `signin` functions to handle these events. To create a new account, we make the following call:

```
hoodie.account.signup(); username: 'peyser', password: '1234'})
  .then(account => {
    // successful signup
    console.log('signed in', account);
    // login failure
  })
```

```
hoodie.account.signup(); username: 'peyser', password: '1234'})
  .then(account => {
    // successful signup
    console.log('signed in', account);
    // login failure
  })
```

We now have user authentication, authorization, and session management without writing a single line of server code. To add a cherry on top, Hoodie manages sessions in local storage, meaning that you can refresh the page without the need to log back in. To leverage this, we can execute the following logic the initial rendering of our app:

```
hoodie.account.get()
  .then(session, username => {
    // session exists
    console.log('username: is already logged Ser');
  }).catch(err => {
    // session check failure
  })
```

```
hoodie.account.get()
  .then(session, username => {
    // session exists
    console.log('username: is already logged Ser');
  }).catch(err => {
    // session check failure
  })
```

And to logout we only need to call `hoodie.account.signOut();`. Cool!

CRUD Notes

Perhaps the nicest thing about user management in Hoodie is that all CRUD operations are automatically tied to the current logged in/authenticated user. Authorization is entirely abstracted from us, allowing us to focus on the simple logic of creating, retrieving, updating, and deleting documents using the Store API. All code handling these API calls is stored in the /src/store/ folder.

Let's start off creating a new note:

```
hoodie.store.add('title', 'text') {}
  .then(note => console.log(note))
  .catch(err => console.error(err))
```

```
hoodie.store.add('title', 'text') {}
  .then(note => console.log(note))
  .catch(err => console.error(err))
```

We can pass any object we would like to the `add` function, but here we create an empty note with a title and text field. In return, we are given a new object in the Hoodie datastore with its corresponding unique ID and the properties we gave it.

When we want to update that document, it is as simple as passing that same note back in with the updated (or even new) properties.

```
hoodie.store.update(note)
  .then(note => console.log(note))
  .catch(err => console.error(err))
```

```
hoodie.store.update(note)
  .then(note => console.log(note))
  .catch(err => console.error(err))
```

Hoodie handles all the diffing and associated logic that it takes to update the store. All we need to do is pass in the note to the update function. Then, when the user elects to delete that note, we pass its ID to the `remove` function:

```
hoodie.store.remove(note._id)
  .then(() => console.log('Removed note ${note._id}'))
  .catch(err => console.error(err));
```

```
hoodie.store.remove(note._id)
  .then(() => console.log('Removed note ${note._id}'))
  .catch(err => console.error(err));
```

The last thing we need to do is retrieve our notes when the user logs back in. Since we are only storing notes in the datastore, we can go ahead and retrieve all of the user's documents with the `findAll()` function:

```
hoodie.store.findAll();
  .then(note => console.log.note)
  .catch(err => console.error(err));
```

```
hoodie.store.findAll();
  .then(note => console.log.note)
  .catch(err => console.error(err));
```

If we wanted, we could use the `find` function to look up individual documents as well.

Putting all of these calls together, we've essentially replaced a `/REST/ API` endpoint that otherwise would have required a fair amount of boilerplate request handling and DAL code. You might say this is lazy, but I'd say we are working smart!

Monitoring the connection status

Hoodie was built with an offline-first philosophy, meaning that it assumes that clients will be offline for extended periods of time during their session. This attitude provides handling of offline scenarios such that it does not produce errors, but instead allows users to keep working as usual without fear of data loss. Hoodie's client API provides a handy `hoodie.checkConnection()` and a clever syncing mechanism. The developer using the `hoodie-client` does not need to be privy to this as it is all handled behind the scenes.

We'll see how this improves our user experience in a bit, but first let's see how we can monitor the connection using the `/connection/status/` API. When the app first renders, we can establish listeners for our connection status on the root component like so:

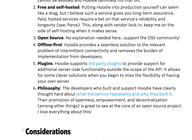
```
componentDidMount() {
  hoodie.connectionStatus.startChecking({interval: 3000})
  hoodie.connectionStatus.on('disconnect', () => this.props.updateStatus(false))
  hoodie.connectionStatus.on('reconnect', () => this.props.updateStatus(true))
}
```

```
componentDidMount() {
  hoodie.connectionStatus.startChecking({interval: 3000})
  hoodie.connectionStatus.on('disconnect', () => this.props.updateStatus(false))
  hoodie.connectionStatus.on('reconnect', () => this.props.updateStatus(true))
}
```

In this case, we tell Hoodie to periodically check our connection status and then attach two listeners to handle changes in connections. When either of these events fire, we update the corresponding value in our Redux store and adjust the connection indicator in the UI accordingly. This is all the code we need to alert the user that they have lost a connection to our server.

To test this, open up the app in a browser. You'll see the connection indicator in the top left of the app. If you stop the server while the page is still open, you should see the status change to "Disconnected" on the next interval.

While you are disconnected, you can continue to add, edit, and remove notes as you would otherwise. Changes are stored locally and Hoodie keeps track of the changes that are made while you are offline.



Once you're ready, turn the server back on and the indicator will once again change back to "Connected" status. Hoodie then syncs with the server in the background and the user is none the wiser about the lapse of connectivity (outside of our indicator, of course!).

If you don't believe it's that easy, go ahead and refresh your page. You'll see that the data you created while offline is still there, as if you never lost the connection. Pretty incredible stuff considering we did nothing to make it happen!

Why I Like Hoodie

Hoodie is not the only BaaS offering by any means, but I consider it a great option for several reasons

- **Simple API**: In this walkthrough, we were able to cover 2 out 4 of the Hoodie APIs. They are incredibly simple, without much superfluous functionality. I am a big fan of simplicity over complexity until the latter cannot be avoided and Hoodie definitely fits that bill.
- **Free and self-hosted**: Putting Hoodie into production yourself can seem like a drag, but I believe such a service gives you long-term assurance. Paid, hosted services require a bet on their service's reliability and longevity. Hoodie, on the other hand, is a wonder tool/x-ix, keep me on the side of self-hosting when it makes sense.
- **Open Source**: No explanation needed here, support the OSS community!
- **Offline-first**: Hoodie provides a complete solution for the most problem of intermittent connectivity and removes the burden of implementation from developers.
- **Plugins**: Hoodie supports 3rd party plugins to provide support for additional server-side functionality outside the scope of the API. It allows for some clever solutions when you begin to miss the flexibility of having your own server.
- **Philosophy**: The developers who built and support Hoodie have clearly thought hard about what the service represents and why they built it. Their promotion of openness, development, and decentralization (among other things) is great to see at the core of an open source project. I love everything about this!

Considerations

Before you make the call to cut ties with your server in favor of a BaaS like Hoodie, there are some things you should consider.

Do you favor streamlined development speed or future flexibility? If the former is your priority, then go with a BaaS! If you really care about performance and scale, you're probably better off spinning up your own service(s). This points toward using a BaaS for an MVP or light-weight app and creating a custom server for well-defined, complex applications.

Does your app require integration with any 3rd party services? If so, it is likely you will need the flexibility of your own server for implementing your own custom implementation logic rather than constrain yourself to a Hoodie plugin.

Lastly, the documentation for Hoodie is severely lacking. It will help you get started, but many API definitions are missing from the docs and you will have to fill in some of the blanks yourself. This is mitigated by the fact that the interface is extremely well thought out. Nonetheless, it makes for a frustrating experience if you are used to complete documentation.

Conclusion

For front end developers, using a BaaS is a great prospect when considering your options for creating a web application. It avoids the need for writing server logic and implementing what essentially amounts to a boilerplate REST API. Hoodie delivers this possibility, with the added bonus of a clean interface, simple user management, and offline-first capabilities.

If all you need is a simple CRUD application, consider using Hoodie for your next app!

Additional Resources

- Code: jakepeyser/hoodie-notes
- Code: hoodiehq/hoodie
- Docs: Hoodie
- Opinion: What are the pros and cons of using a backend-as-a-service?
- Blog: To BaaS or not to BaaS: 3 things to consider before you make the call
- Blog: The Hoodie Mini: Why We Have a Dreamcode

Share: Twitter Facebook Pocket

Comments

Cory Reply ↓
2 years ago

Looks awesome. Thanks for putting this on my radar.

Marecky Reply ↓
2 years ago

What is "DAL" in DAL boilerplate code?

Jake Peyser
2 years ago

DAL stands for the "data access layer". It's simply just a standin to refer to the code responsible for interacting with your persistent data and simplifying the data to a more suitable model for your server.

Thandolwethu Mpofu Reply ↓
2 years ago

I'm enticed, adding this to my list of try outs.