Hoodie Documentation

Release

Hooodie Project

Sep 28, 2017

Contents

CHAPTER 1

Welcome to Hoodie

Hoodie is a backend for web applications with a JavaScript API for your frontend. If you love building apps with HTML, CSS and JavaScript or a frontend framework, but *dread* backend work, Hoodie is for you.

Hoodie's frontend API gives your code superpowers by allowing you to do things that usually only a backend can do (user accounts, emails, payments, etc.).

All of Hoodie is accessible through a simple script include, just like jQuery or lodash:

```
<script src="/hoodie/client.js"></script>
```

From that point on, things get really powerful really quickly:

```
// In your front-end code:
hoodie.ready.then(function () {
  hoodie.account.signUp({
    username: username,
    password: password
  })
})
```

That's how simple signing up a new user is, for example. But anyway:

Hoodie is a frontend abstraction of a generic backend web service. As such, it is agnostic to your choice of frontend application framework. For example, you can use jQuery for your web app and Hoodie for your connection to the backend, instead of raw jQuery.ajax. You could also use React with Hoodie as a data store, or any other frontend framework or library, really.

Open Source

Hoodie is an Open Source project, so we don't own it, can't sell it, and it won't suddenly vanish because we got aquired. The source code for Hoodie is available on GitHub under the Apache License 2.0.

How to proceed

You *could read up on some of the ideological concepts behind Hoodie*, such as noBackend and Offline First. These explain why Hoodie exists and why it looks and works the way it does.

If you're more interested in the technical details of Hoodie, check out *How Hoodie Works*. Learn how Hoodie handles data storage, does syncing, and where the offline support comes from.

Eager to build stuff? Skip ahead to the *quickstart guide*!

CHAPTER 2

Quickstart

In this guide you'll learn how to create a demo Hoodie app, learn about the basic structure of a Hoodie project and its folders, the endpoints and app URLs and how to include and use the Hoodie library in your project.

Prerequisites

For all operating systems, you'll need Node.js installed. You can download Node from nodejs.org. We recommend the LTS (Long Term Support) version.

Make sure you have version 4 or higher. You can find out with

```
$ node -v
```

Create a new Hoodie Backend

First you need to create a new folder, let's call it **testapp**

```
$ mkdir testapp
```

Change into the `testapp` directory.

```
$ cd testapp
```

Now we need to create a **package.json** file. For that we can use npm which comes with Node by default. It will ask you a few questions, you can simply press enter to leave the default values.

```
$ npm init
```

Now we can install **hoodie** using npm

```
$ npm install hoodie --save
```

The resulting **package.json** file in the current folder, should look something like this

```
{
  "name": "funky",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "hoodie",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Now you can start Hoodie with

```
$ npm start
```

Great, your Hoodie backend started up and is now telling you at which URL you can access it. By default that is http://127.0.0.1:8080

Congratulations, you just created your first Hoodie Backend :) You can now load the Hoodie client on any website with

```
<script src="http://127.0.0.1:8080/hoodie/client.js"></script>
```

You can also create a `public/index.html` file, which will be served at http://127.0.0.1:8080 after you restart the server. All assets in the public folder, like images, CSS files or JavaScript files, will be served by your Hoodie Backend at `http://127.0.0.1:8080/<path/to/your/file.ext>`.

Note for npm v2

Because of how npm v2 installs sub dependencies, the hoodie client cannot be bundled. As a workaround, just install `pouchdb-browser` and `@hoodie/client` as a dependency of your hoodie app

```
$ npm install --save pouchdb-browser @hoodie/client
```

What's next?

Our Hoodie Tracker App is a great place to see how to use a Hoodie backend. It's an intentionally simple and well commented application built with only HTML, JavaScript and CSS, without using any library or framework. You can see it running at https://tracker.hood.ie/

Having Trouble?

Sorry it didn't go smoothly for you. Come chat with us or ask a question on StackOverflow

CHAPTER 3

Configuration

Your Hoodie back-end can be configured using default options that are part of your repository as well as using hidden files, CLI arguments and environment variables.

Options

Here is a list of all available options

Option	Default value	CLI argument	ENV variable	description
address	`'127.0.0.1'`	`--address`	`hoodie_address`	Address to which Hoodie binds
data	`'.hoodie'`	`--data`	`hoodie_data`	Data path
dbUrl	–	`--dbUrl`	`hoodie_dbUrl`	If provided, uses external CouchDB. Include credentials in *dbUrl*, or use *dbUrlUsername* and *dbUrlPassword*. Sets `dbAdapter` to `pouchdb-adapter-http`
dbUrlUsername	–	`dbUrlUsername`	`hoodie_dbUrlUsername`	If `dbUrl` is set, you can use `dbUrlUsername` to set the username to use when making requests to CouchDB
dbUrlPassword	–	`dbUrlPassword`	`hoodie_dbUrlPassword`	If `dbUrl` is set, you can use `dbUrlPassword` to set the password to use when making requests to CouchDB
dbAdapter	`'pouchdb-adapter-fsdb'`	`dbAdapter`	`hoodie_dbAdapter`	Sets default *PouchDB adapter <https://pouchdb.com/adapters.html>* unless `inMemory` or `dbUrl` set
loglevel	`'warn'`	`--loglevel`	`hoodie_loglevel`	One of: silent, error, warn, http, info, verbose, silly
inMemory	`false`	`-m`, `--inMemory`	`hoodie_inMemory`	Whether to start the PouchDB Server in memory. Sets `dbAdapter` to `pouchdb-adapter-memory`
port	`8080`	`--port`	`hoodie_port`	Port-number to run the Hoodie App on
public	`'public'`	`--public`	`hoodie_public`	path to static assets
url	•	`--url`	`hoodie_url`	Optional: external URL at which Hoodie Server is accessible (e.g. `http://myhoodieapp.com`)
adminPassword	•	`--adminPassword`	`hoodie_adminPassword`	Password to login to Admin Dashboard. Login is not possible unless set
name	`package.json`s name property	`--name`	`hoodie_name`	Name your application.

Defaults

Default options are set in your app's `package.json` file, using the `"hoodie"` key. Here is an example with all available options and their default values

```
{
  "hoodie": {
    "address": "127.0.0.1",
    "port": 8080,
    "data": ".hoodie",
    "public": "public",
    "dbUrl": "",
    "dbAdapter": "pouchdb-adapter-fs",
    "inMemory": false,
    "loglevel": "warn",
    "url": "",
    "adminPassword": "",
    "name": "my-hoodie-app"
  }
}
```

.hoodierc

The `.hoodierc` can be used to set configuration when running your Hoodie backend in that folder. It should not be committed to your repository.

The content can be in JSON or INI format. See the rc package on npm for more information

CLI arguments and environment variables

To pass CLI options when starting Hoodie, you have to separate them with `--`, for example:

```
$ npm start -- --port=8090 --inMemory
```

All environment variables are prefixed with `hoodie_`. So to set the port to `8090` and to start Hoodie in memory mode, you have to

- set the `hoodie_port` environment variable to `8090`
- set the `hoodie_inMemory` environment variable to `true`

Hoodie CLI is using rc for configuration, so the same options can be set with environment variables and config files. Environment variables are prefixed with `hoodie_`.

The priority of configuration

1. Command line arguments
2. Environment variables
3. `.hoodierc` files
4. Your app's defaults form the `"hoodie"` key in `"package.json"`

5. Hoodie's default values as shown in table above

CHAPTER 4

Plugins

You can extend your Hoodie app in two ways

1. App-specific plugins
2. 3rd party plugins

App-specific plugins

You can extend your Hoodie's client by creating the file `hoodie/client/index.js` in your app's repository, which should export a *Hoodie Client plugin <../api/client/hoodie.html#hoodie-plugin>*. It will dynamically be bundled into your client accessible at the `/hoodie/client.js` route.

Example

```
// /hoodie/client.js
module.exports = function (hoodie) {
  hoodie.hello = function (what) {
    return Promise.resolve('Hello, ' + (what || 'world') + '!')
  }
}
```

You can extend your Hoodie's server routes and API by creating `hoodie/server/index.js` in your app's, which should export a hapi plugin. All server routes defined in the plugin will be prefixed with `/hoodie/<app name>`

Example

```
module.exports.register = register
module.exports.register.attributes = {
  name: 'hoodie-app-plugin'
}

function register (server, options, next) {
  server.route({
    method: 'GET',
```

```
    path: '/api',
    handler: function (request, reply) {
      reply('Hello, world!')
    }
  })

  next()
}
```

3rd party plugins

Hoodie plugins are npm modules. We recommend to prefix your plugin names with `hoodie-plugin-`, but it's not required. The folder structure is the same as for app-specific plugins:

The server plugin must be loadable via `require('hoodie-plugin-foo/hoodie/server')`. A Hoodie server plugin is a hapi plugin. The client plugin must be loadable via `require('hoodie-plugin-foo/hoodie/client')` A Hoodie client plugin can be a function or an object, it will be passed into *hoodie.plugin() <../api/client/hoodie.html#hoodie-plugin>*

Hoodie plugins can extend the Hoodie client, the Hoodie server and provide a web UI for */hoodie/<plugin name>*. All extension points are optional. The `hoodie/public` folder will be exposed at */hoodie/<plugin name>* by the server if it exists. All server routes will be prefixed with */hoodie/<plugin name>*.

`<plugin name>` is the name property in your `package.json` file, but can be overridden with the `hoodie.name` property.

After installing and adding a Hoodie plugin to your app's dependencies, you also have to enable it by adding it to the `hoodie.plugins` array in your app's *package.json* file. The names are the npm package names.

The order in which server/client plugins are loaded is

1. core modules (account, store, task)
2. 3rd party plugins (npm dependencies)
3. app plugins

For an example plugin, have a look at Hoodie's "Hello, world!" plugin .

CHAPTER 5

Deployment

One line deploy

After you've built your Hoodie app you probably want to put it online. You can choose to deploy your app as read-only or deploy the backend couchdb database as well. This video and the text below describes how to deploy your app using one line of code. Alternatively, you can deploy your app using Docker, please refer to the Docker section.

Deploying to Now

Now allows you to deploy a Node application with its command line tool. It's 100% free for Open Source projects. You can deploy an app from your computer or right from a GitHub repository. For example, to deploy our Hoodie Tracker demo app all you have to do is to run this command:

```
$ now hoodiehq/hoodie-app-tracker --npm -e NODE_ENV=production -e hoodie_inMemory=true
```

To describe this further:

- `hoodiehq/hoodie-app-tracker` is the GitHub repository slug.
- `--npm` tells now to deploy using npm as there is also Dockerfile in the repository.
- `-e NODE_ENV=production` sets the NODE_ENV environment variable to production, which makes the deployment faster as no devDependencies will be installed.
- `-e hoodie_inMemory=true` makes the Hoodie app run in-memory mode, meaning that no data is persisted and no files are written. This is important because now is a read-only file system. That means that all user accounts and data will be lost on the next deployment, but it is great for for a quick test or demo of your application.

Alternatively, add this script to your package.json and you are good to go:

```
"now-start": "hoodie --inMemory",
```

Store Data With Cloudant

Cloudant is a DBaaS (database-as-a-service). It provides most of CouchDB's APIs and can be used as Hoodie's database backend. Signing up for a free account only takes a moment. After sign up, you need to slightly adjust the now deployment command above.

```
$ now hoodiehq/hoodie-app-tracker -e NODE_ENV=production -e hoodie_inMemory=true -e␣
↪hoodie_dbUrl=https://username:password@username.cloudant.com/
```

The `hoodie_inMemory` environment variable makes sure that Hoodie does not try to write any files like the bundled /hoodie/client.js library. The `hoodie_dbUrl` environment variable sets the address and credentials to your CouchDB. Replace username and password to whatever you signed up with.

Test and set an alias

When you deploy with now you will receive a random subdomain where you can access your application. It looks something like https://hoodie-app-tracker-randomxyz.now.sh/ and was already copied to your clipboard during the deployment. Open the URL in your browser to give it a try. Once everything is good, you can change the subdomain to your preference by running:

```
$ now alias set hoodie-app-tracker-randomxyz my-tracker-app
```

That will make your deployed Hoodie Tracker app accessible at https://my-tracker-app.now.sh. For example, here is the app that I deployed myself: https://hoodie-app-tracker.now.sh/

Docker

We continuously deploy our Hoodie Tracker App using Docker. You can read about our continuous deployment set at hoodie-app-tracker/deployment.md.

Deployment in linux

This guide is for Linux only at this point. I have tried to deploy Hoodie-App-Tracker as an example:

install dependencies

1. Install CouchDB 1.2.0 or later, 1.4.0 or later recommended for performance.
2. Install NodeJS LTS version or later. This includes npm.
3. Install git.

CouchDB

We assume you set up CouchDB with your package manager or manually following the installation procedure.

In order to test if CouchDB is running fine or not, we can simply run the following command which will retrieve the information through curl.

```
$ curl localhost:5984
```

If you are already using CouchDB for other things, we recommend starting a second instance of CouchDB that is completely separate from your original one. See below for instructions.

In this guide, we assume that your CouchDB is available at port 5984.

Create a CouchDB admin user called **admin** with a strong password of your choice at by clicking on the *Fix this* at Apache CouchDB-Futon:Overview link in the lower right corner. Use **admin** as username and keep your password in mind.

Next we have to change CouchDB's default configuration on a few points. The easiest thing is to go to and change the following fields (double click a value to enter the editing mode):

```
couchdb -> delayed_commits: false
couchdb -> max_dbs_open: 1024
```

System

Add this to `/etc/security/limits.conf`:

```
hoodie    soft    nofile    768
hoodie    hard    nofile    1024
```

Hoodie

Create a new system user:

```
$ sudo useradd --system \
    -m \
    --home /home/hoodie \
    --shell /bin/bash \
    --no-user-group \
    -c "Hoodie Administrator" hoodie
```

This will create a new user and its home directory /home/hoodie. But unless you have a password, you can not be a user. To set a password run:

```
$ sudo passwd hoodie
```

Give a password of your choice.

cd in to that directory.

To switch to **hoodie** user, run:

```
$ sudo su hoodie
```

As user Hoodie, install your application:

```
$ git clone <repo url>
```

make sure package.json has a valid *name* property.

cd into the directory.Run :

```
$ cd <repo name>
```

Now run:

```
$ npm install
```

To run Hoodie as the root:

```
$ sudo su hoodie
```

To launch Hoodie now, as root :

```
$ npm start -- --dbUrl=http://admin:yourpassword@localhost:5984/
```

Replace `yourpassword` with the password you choose when you created the admin user above.

That's it. The app should be running by now.

CHAPTER 6

Using Hoodie as hapi plugin

Here is an example usage of Hoodie as a hapi plugin:

```
var Hapi = require('hapi')
var hoodie = require('hoodie').register
var PouchDB = require('pouchdb-core')
  .plugin(require('pouchdb-mapreduce'))
  .plugin(require('pouchdb-adapter-memory'))

var server = new Hapi.Server()
server.connection({
  host: 'localhost',
  port: 8000
})

server.register({
  register: hoodie,
  options: { // pass options here
    inMemory: true,
    public: 'dist',
    PouchDB: PouchDB
  }
}, function (error) {
  if (error) {
    throw error
  }

  server.start(function (error) {
    if (error) {
      throw error
    }

    console.log(('Server running at:', server.info.uri))
  })
})
```

The available options are

option	default	description
PouchDB	–	PouchDB constructor. See also custom PouchDB builds
paths.data	`'.hoodie'`	Data path
paths.public	`public`	Public path
adminPassword	–	Password to login to Admin Dashboard. Login is not possible if `adminPassword` option is not set
inMemory	`false`	If set to true, configuration and other files will not be read from / written to the file system
client	`{}`	Hoodie Client options. `client.url `` is set based on hapi's ``server.info.host`
account	`{}`	Hoodie Account Server options. `account.admins`, `account.secret` and `account.usersDb` are set based on `db` option above
store	`{}`	Hoodie Store Server options. `store.couchdb`, `store.PouchDB` are set based on `db` option above. ‘‘store.hooks.onPreAuth‘ ‘ is set to bind user authentication for Hoodie Account to Hoodie Store
plugins	`[]`	Array of npm names or paths of locations containing plugins. See also Hoodie plugins docs
app	`{}`	App specific options for plugins

CHAPTER 7

Hoodie API

Hoodie provides two APIs

1. The Hoodie Client API

   The Hoodie Client API is what you load into your web application using a script tag. It connects to your Hoodie Backend's routes

2. The Hoodie Server API

   The Hoodie Server API is used within Hoodie's route handlers and by plugins to manage accounts, data and to securely integrate with 3rd party services.

The Hoodie Client API

hoodie

Introduction

This document describes the functionality of the hoodie base object. It provides a number of helper methods dealing with event handling and connectivity, as well as a unique id generator and a means to set the endpoint which Hoodie communicates with.

Initialisation

The Hoodie Client persists state in the browser, like the current user's id, session or the connection status to the backend.

```
hoodie.account.get('session').then(function (session) {
  if (session) {
    // user is signed in
  } else {
    // user is signed out
```

```
  }
})
```

Hoodie integrates Hoodie's client core modules:

- The account API
- The store API
- The connectionStatus API
- The log API

Example

```
var Hoodie = require('@hoodie/client')
var hoodie = new Hoodie({
  url: 'https://myhoodieapp.com',
  PouchDB: require('pouchdb')
})

hoodie.account.signUp({
  username: 'pat@Example.com',
  password: 'secret'
}).then(function (accountAttributes) {
  hoodie.log.info('Signed up as %s', accountAttributes.username)
}).catch(function (error) {
  hoodie.log.error(error)
})
```

Constructor

```
new Hoodie(options)
```

Argument	Type	Description	Required
options.PouchDB	Constructor	PouchDB constructor, see also PouchDB custom builds	Yes
options.url	String	Set to hostname where Hoodie server runs, if your app runs on a different host	Yes
options.account	String	account options. `options.url` is always set to `hoodie.url` + '/account/api'	No
options.store	String	store options. `options.PouchDB` is always set to Hoodie Client's constructor's options.PouchDB. `options.dbName` is always set to `hoodie.account.id`. `options.remote` is always set to `hoodie.url` + '/store/api'.	No
options.task	String	task options. options.userId is always set to hoodie.account.id. options.remote is always set to hoodie.url + '/task/api'	No
options.connectionStatus	String	connectionStatus options. `options.url` is always set to `hoodie.url` + '/connection-status/api'. `options.method` is always set to `HEAD`	No

hoodie.url

Read-only

```
hoodie.url
```

full url to the hoodie server, e.g. `http://example.com/hoodie`

hoodie.account

`hoodie.account` is an instance of hoodie-account-client. See account API

hoodie.store

`hoodie.store` is an instance of hoodie-store. See store API

hoodie.connectionStatus

`hoodie.connectionStatus` is an instance of hoodie-connection-status. See connectionStatus API

hoodie.log

`hoodie.log` is an instance of hoodie-log. See log API

hoodie.request

Sends an http request

```
hoodie.request(url)
// or
hoodie.request(options)
```

Argument	Type	Description	Required
url	String	Relative path or full URL. A path must start with `/` and sends a `GET` request to the path, prefixed by `hoodie.url`. In case a full URL is passed, a `GET` request to the url is sent.	Yes
options.url	String	Relative path or full URL. A path must start with `/` and sends a `GET` request to the path, prefixed by `hoodie.url`. In case a full URL is passed, a `GET` request to the url is sent.	Yes
options.method	String	*Defaults to* `GET`. One of `GET`, `HEAD`, `POST`, `PUT`, `DELETE`.	No
options.data	Object, Array, String or Number	For `PUT` and `POST` requests, an optional payload can be sent. It will be stringified before sending the request.	No
options.headers	Object	Map of Headers to be sent with the request.	No

Examples

```
// sends a GET request to hoodie.url + '/foo/api/bar'
hoodie.request('/foo/api/bar')
// sends a GET request to another host
hoodie.request('https://example.com/foo/bar')
// sends a PATCH request to /foo/api/bar
```

```
hoodie.request({
  method: 'PATCH',
  url: '/foo/api/bar',
  headers: {
    'x-my-header': 'my value'
  },
  data: {
    foo: 'bar'
  }
})
```

hoodie.plugin

Initialise hoodie plugin

```
hoodie.plugin(methods)
hoodie.plugin(plugin)
```

Argument	Type	Description	Required
methods	Object	Method names as keys, functions as values. Methods get directly set on hoodie, e.g. hoodie.plugin({foo: function () {}}) sets hoodie.foo to function () {}	Yes
plugins	Function	The passed function gets called with *hoodie* as first argument, and can directly set new methods / properties on it.	Yes

Examples

```
hoodie.plugin({
  sayHi: function () { alert('hi') }
})
hoodie.plugin(function (hoodie) {
  hoodie.sayHi = function () { alert('hi') }
})
```

hoodie.on

Subscribe to event.

```
hoodie.on(eventName, handler)
```

Example

```
hoodie.on('account:signin', function (accountProperties) {
  alert('Hello there, ' + accountProperties.username)
})
```

hoodie.one

Call function once at given event.

```
hoodie.one(eventName, handler)
```

Example

```
hoodie.one('mycustomevent', function (options) {
  console.log('foo is %s', options.bar)
})
hoodie.trigger('mycustomevent', { foo: 'bar' })
hoodie.trigger('mycustomevent', { foo: 'baz' })
// logs "foo is bar"
// DOES NOT log "foo is baz"
```

hoodie.off

Removes event handler that has been added before

```
hoodie.off(eventName, handler)
```

Example

```
hoodie.off('connectionstatus:disconnect', showNotification)
```

hoodie.trigger

Trigger custom events

```
hoodie.trigger(eventName[, option1, option2, ...])
```

Example

```
hoodie.trigger('mycustomevent', { foo: 'bar' })
```

Events

Event	Decription
`account:*`	events, see account events
`store:*`	events, see store events
`connectionStatus:*`	events, see connectionStatus events

Testing

Local setup

```
git clone https://github.com/hoodiehq/hoodie-client.git
cd hoodie-client
npm install
```

Run all tests

```
npm test
```

Run test from one file only

```
node tests/specs/id
```

hoodie.account

The account object in the client-side Hoodie API covers all user and authentication-related operations, and enables you to do previously complex operations, such as signing up a new user, with only a few lines of frontend code. Since data in Hoodie is generally bound to a user, it makes sense to familiarise yourself with **account** before you move on to store.

`hoodie-account-client` is a JavaScript client for the Account JSON API. It persists session information in localStorage (or your own store API) and provides front-end friendly APIs for the authentication-related operations as mentioned above.

Example

```
hoodie.account.get('session').then(function (sessionProperties) {
  if (!sessionProperties) {
    return redirectToHome()
  }

  renderWelcome(sessionProperties)
}).catch(redirectToHome)

hoodie.account.on('signout', redirectToHome)
```

hoodie.account.validate

Calls the function passed into the Constructor. Returns a Promise that resolves to `true` by default

```
hoodie.account.validate(options)
```

Argument	Type	Required
`options.username`	String	No
`options.password`	String	No
`options.profile`	Object	No

Resolves with an argument.

Rejects with any errors thrown by the function originally passed into the Constructor.

Example

```
hoodie.account.validate({
    username: 'DocsChicken',
    password: 'secret'
})

.then(function () {
    console.log('Successfully validated!')
})

.catch(function (error) {
    console.log(error) // should be an error about the password being too short
})
```

hoodie.account.hasInvalidSession

Checks `hoodie.account.session.invalid property`. Returns `true` if user has invalid session, otherwise `undefined`.

```
hoodie.account.hasInvalidSession()
```

hoodie.account.signUp

Creates a new user account on the Hoodie server. Does *not* sign in the user automatically, *hoodie.account.signIn* must be called separately.

```
hoodie.account.signUp(accountProperties)
```

Argument	Type	Required
`accountProperties.username`	String	Yes
`accountProperties.password`	String	Yes

Resolves with `accountProperties`:

```
{
    "id": "account123",
    "username": "pat",
    "createdAt": "2016-01-01T00:00.000Z",
    "updatedAt": "2016-01-01T00:00.000Z"
}
```

Rejects with:

InvalidError	Username must be set
`SessionError`	Must sign out first
`ConflictError`	Username **<username>** already exists
`ConnectionError`	Could not connect to server

Example

```
hoodie.account.signUp({
    username: 'pat',
    password: 'secret'
}).then(function (accountProperties) {
    alert('Account created for ' + accountProperties.username)
}).catch(function (error) {
    alert(error)
})
```

hoodie.account.signIn

Creates a user session

```
hoodie.account.signIn(options)
```

Argument	Type	Description	Required
`options.username`	String	•	Yes
`options.password`	String	•	Yes

Resolves with `accountProperties`:

```
{
    "id": "account123",
    "username": "pat",
    "createdAt": "2016-01-01T00:00.000Z",
    "updatedAt": "2016-01-02T00:00.000Z",
    "profile": {
        "fullname": "Dr. Pat Hook"
    }
}
```

Rejects with:

`UnconfirmedError`	Account has not been confirmed yet
`UnauthorizedError`	Invalid Credentials
`Error`	*A custom error set on the account object, e.g. the account could be blocked due to missing payments*
`ConnectionError`	Could not connect to server

Example

```
hoodie.account.signIn({
    username: 'pat',
    password: 'secret'
}).then(function (sessionProperties) {
    alert('Ohaj, ' + sessionProperties.username)
}).catch(function (error) {
    alert(error)
})
```

hoodie.account.signOut

Deletes the user's session

```
hoodie.account.signOut()
```

Resolves with `sessionProperties` like *hoodie.account.signIn*, but without the session id:

```
{
    "account": {
        "id": "account123",
        "username": "pat",
        "createdAt": "2016-01-01T00:00.000Z",
        "updatedAt": "2016-01-02T00:00.000Z",
        "profile": {
            "fullname": "Dr. Pat Hook"
        }
    }
}
```

Rejects with:

`Error`	A custom error thrown in a `before:signout` hook

Example

```
hoodie.account.signOut().then(function (sessionProperties) {
    alert('Bye, ' + sessionProperties.username)
}).catch(function (error) {
    alert(error)
})
```

hoodie.account.destroy

Destroys the account of the currently signed in user.

```
hoodie.account.destroy()
```

Resolves with `sessionProperties` like *hoodie.account.signIn*, but without the session id:

```
{
    "account": {
        "id": "account123",
        "username": "pat",
        "createdAt": "2016-01-01T00:00.000Z",
        "updatedAt": "2016-01-02T00:00.000Z",
        "profile": {
            "fullname": "Dr. Pat Hook"
        }
    }
}
```

Rejects with:

`Error`	A custom error thrown in a `before:destroy` hook
`ConnectionError`	Could not connect to server

Example

```
hoodie.account.destroy().then(function (sessionProperties) {
    alert('Bye, ' + sessionProperties.username)
}).catch(function (error) {
    alert(error)
})
```

hoodie.account.get

Returns account properties from local cache.

```
hoodie.account.get(properties)
```

Argument	Type	Description	Required
`properties`	String or Array of strings	When String, only this property gets returned. If array of strings, only passed properties get returned	No

Returns object with account properties, or `undefined` if not signed in.

Examples

```
var properties = hoodie.account.get()
alert('You signed up at ' + properties.createdAt)
var createdAt = hoodie.account.get('createdAt')
alert('You signed up at ' + createdAt)
var properties = hoodie.account.get(['createdAt', 'updatedAt'])
alert('You signed up at ' + properties.createdAt)
```

hoodie.account.fetch

Fetches account properties from server.

```
hoodie.account.fetch(properties)
```

Argument	Type	Description	Required
properties	String or Array of strings	When String, only this property gets returned. If array of strings, only passed properties get returned. Property names can have '.' separators to return nested properties.	No

Resolves with `accountProperties`:

```
{
    "id": "account123",
    "username": "pat",
    "createdAt": "2016-01-01T00:00.000Z",
    "updatedAt": "2016-01-02T00:00.000Z"
}
```

Rejects with:

`UnauthenticatedError`	Session is invalid
`ConnectionError`	Could not connect to server

Examples

```
hoodie.account.fetch().then(function (properties) {
    alert('You signed up at ' + properties.createdAt)
})
hoodie.account.fetch('createdAt').then(function (createdAt) {
    alert('You signed up at ' + createdAt)
})
hoodie.account.fetch(['createdAt', 'updatedAt']).then(function (properties) {
    alert('You signed up at ' + properties.createdAt)
})
```

hoodie.account.update

Update account properties on server and local cache

```
hoodie.account.update(changedProperties)
```

Argument	Type	Description	Required
changedProperties	Object	Object of properties & values that changed. Other properties remain unchanged.	No

Resolves with accountProperties:

```
{
    "id": "account123",
    "username": "pat",
    "createdAt": "2016-01-01T00:00.000Z",
    "updatedAt": "2016-01-01T00:00.000Z"
}
```

Rejects with:

`UnauthenticatedError`	Session is invalid
`InvalidError`	Custom validation error
`ConflictError`	Username **<username>** already exists
`ConnectionError`	Could not connect to server

Example

```
hoodie.account.update({username: 'treetrunks'}).then(function (properties) {
    alert('You are now known as ' + properties.username)
})
```

hoodie.account.profile.get

Returns profile properties from local cache.

```
hoodie.account.profile.get(properties)
```

Argument	Type	Description	Required
`properties`	String or Array of strings	When String, only this property gets returned. If array of strings, only passed properties get returned. Property names can have . separators to return nested properties.	No

Returns object with profile properties, falls back to empty object `{}`. Returns `undefined` if not signed in.

Examples

```
var properties = hoodie.account.profile.get()
alert('Hey there ' + properties.fullname)
var fullname = hoodie.account.profile.get('fullname')
alert('Hey there ' + fullname)
var properties = hoodie.account.profile.get(['fullname', 'address.city'])
alert('Hey there ' + properties.fullname + '. How is ' + properties.address.city + '?
↪')
```

hoodie.account.profile.fetch

Fetches profile properties from server.

```
hoodie.account.profile.fetch(options)
```

Argument	Type	Description	Required
`properties`	String or Array of strings	When String, only this property gets returned. If array of strings, only passed properties get returned. Property names can have '.' separators to return nested properties.	No

Resolves with `profileProperties`:

```
{
    "id": "account123-profile",
    "fullname": "Dr Pat Hook",
    "address": {
        "city": "Berlin",
        "street": "Adalberststraße 4a"
    }
}
```

Rejects with:

`UnauthenticatedError`	Session is invalid
`ConnectionError`	Could not connect to server

Examples

```
hoodie.account.fetch().then(function (properties) {
    alert('Hey there ' + properties.fullname)
})
hoodie.account.fetch('fullname').then(function (fullname) {
    alert('Hey there ' + fullname)
})
hoodie.account.fetch(['fullname', 'address.city']).then(function (properties) {
    alert('Hey there ' + properties.fullname + '. How is ' + properties.address.city␣
↪+ '?')
})
```

hoodie.account.profile.update

Update profile properties on server and local cache

```
hoodie.account.profile.update(changedProperties)
```

Argument	Type	Description	Required
`changedProperties`	Object	Object of properties & values that changed. Other properties remain unchanged.	No

Resolves with `profileProperties`:

```
{
    "id": "account123-profile",
    "fullname": "Dr Pat Hook",
    "address": {
        "city": "Berlin",
        "street": "Adalberststraße 4a"
    }
}
```

Rejects with:

`UnauthenticatedError`	Session is invalid
`InvalidError`	*Custom validation error*
`ConnectionError`	Could not connect to server

Example

```
hoodie.account.profile.update({fullname: 'Prof Pat Hook'}).then(function (properties)
↪{
    alert('Congratulations, ' + properties.fullname)
})
```

hoodie.account.request

Sends a custom request to the server, for things like password resets, account upgrades, etc.

```
hoodie.account.request(properties)
```

Argument	Type	Description	Required
`properties.type`	String	Name of the request type, e.g. "passwordreset"	Yes
`properties`	Object	Additional properties for the request	No

Resolves with `requestProperties`:

```
{
    "id": "request123",
    "type": "passwordreset",
    "contact": "pat@example.com",
    "createdAt": "2016-01-01T00:00.000Z",
    "updatedAt": "2016-01-01T00:00.000Z"
}
```

Rejects with:

`ConnectionError`	Could not connect to server
`NotFoundError`	Handler missing for "passwordreset"
`InvalidError`	*Custom validation error*

Example

```
hoodie.account.request({type: 'passwordreset', contact: 'pat@example.com'}).
↪then(function (properties) {
    alert('A password reset link was sent to ' + properties.contact)
})
```

hoodie.account.on

```
hoodie.account.on(event, handler)
```

Example

```
hoodie.account.on('signin', function (accountProperties) {
    alert('Hello there, ' + accountProperties.username)
})
```

hoodie.account.one

Call function once at given account event.

```
hoodie.account.one(event, handler)
```

Example

```
hoodie.account.one('signin', function (accountProperties) {
    alert('Hello there, ' + accountProperties.username)
})
```

hoodie.account.off

Removes event handler that has been added before

```
hoodie.account.off(event, handler)
```

Example

```
hoodie.account.off('singin', showNotification)
```

Events

Event	Description	Arguments
`signup`	New user account created successfully	`accountProperties` with `.session` property
`signin`	Successfully signed in to an account	`accountProperties` with `.session` property
`signout`	Successfully signed out	`accountProperties` with `.session` property
`passwordreset`	Email with password reset token sent	
`unauthenticated`	Server responded with "unauthenticated" when checking session	
`reauthenticated`	Successfully signed in with the same username (useful when session has expired)	`accountProperties` with `.session` property
`update`	Successfully updated an account's properties	`accountProperties` with `.session` property

Hooks

```
// clear user's local store signin and after signout
hoodie.account.hook.before('signin', function (options) {
    return localUserStore.clear()
})
hoodie.account.hook.after('signout', function (options) {
    return localUserStore.clear()
})
```

Hook	Arguments
`signin`	`options` as they were passed into `hoodie.account.signIn(options)`
`signout`	`{}`

See before-after-hook for more information.

Requests

Hoodie comes with a list of built-in account requests, which can be disabled, overwritten or extended in hoodie-account-server.

When a request succeeds, an event with the same name as the request type gets emitted. For example, `hoodie.account.request({type: 'passwordreset', contact: 'pat@example.com')` triggers a `passwordreset` event, with the `requestProperties` passed as argument.

`passwordreset`	Request a password reset token

Testing

Local setup

```
git clone https://github.com/hoodiehq/hoodie-account-client.git
cd hoodie-account-client
```

In Node.js

Run all tests and validate JavaScript Code Style using standard

```
npm test
```

To run only the tests

```
npm run test:node
```

To test hoodie-account-client in a browser you can link it into hoodie-account, which provides a dev-server:

```
git clone https://github.com/hoodiehq/hoodie-account.git
cd hoodie-account
npm install
npm link /path/to/hoodie-account-client
npm start
```

hoodie-account bundles hoodie-account-client on `npm start`, so you need to restart hoodie-account to see your changes.

hoodie.store

If you want to do anything with data in Hoodie, this is where it happens and this is the Hoodie Client for data persistence & offline sync.

Example

```
var Store = require('@hoodie/store-client')
    var store = new Store('mydbname', {
    PouchDB: require('pouchdb'),
    remote: 'http://localhost:5984/mydbname'
})
```

Or

```
var PresetStore = Store.defaults({
    PouchDB: require('pouchdb'),
    remoteBaseUrl: 'http://localhost:5984'
})
var store = new PresetStore('mydb')
```

Store.defaults

```
Store.defaults(options)
```

Argument	Type	Description	Required
`options.remoteBaseUrl`	String	Base url to CouchDB. Will be used as remote prefix for store instances	No
options.PouchDB	Constructor	PouchDB custom builds	Yes

Returns a custom Store Constructor with passed default options.

Example

```
var PresetStore = Store.defaults({
    remoteBaseUrl: 'http://localhost:5984'
})
var store = new PresetStore('mydb')
store.sync() // will sync with http://localhost:5984/mydb
```

Constructor

```
new Store(dbName, options)
```

Argument	Type	Description	Required
`dbName`	String	name of the database	Yes
`options.remote`	String	name or URL of remote database	Yes (unless remoteBaseUrl is preset, see Store.defaults)
`options.PouchDB`	Constructor	PouchDB custom builds	Yes (unless preset using Store.defaults))

Returns store API.

Example

```
var Store = require('@hoodie/store-client')
var store = new Store('mydb', { remote: 'http://localhost:5984/mydb' })
store.sync() // will sync with http://localhost:5984/mydb
```

store.add(properties)

```
store.add(properties)
```

Argument	Type	Description	Required
`properties`	Object	properties of document	Yes
`properties._id`	String	If set, the document will be stored at given id	No

Resolves with properties and adds _id (unless provided), createdAt and updatedAt properties.

```
{
    "foo": "bar",
    "hoodie": {
      "createdAt": "2016-05-09T12:00:00.000Z",
      "updatedAt": "2016-05-09T12:00:00.000Z"
    },
    "_id": "12345678-1234-1234-1234-123456789ABC",
    "_rev": "1-b1191b8cfee045f495594b1cf2823683"
}
```

Rejects with:

Add expected Errors: #102

table

Example

```
store.add({foo: 'bar'}).then(function (doc) {
    alert(doc.foo) // bar
}).catch(function (error) {
    alert(error)
})
```

store.add(arrayOfProperties)

```
store.add(arrayOfProperties)
```

Argument	Type	Description	Required
‘’arrayOfProperties’‘	Array	Array of properties, see store.add(properties)	Yes

Resolves with properties and adds _id (unless provided), createdAt and updatedAt properties. Resolves with array of properties items if called with propertiesArray.

```
{
    "foo": "bar",
    "hoodie": {
      "createdAt": "2016-05-09T12:00:00.000Z",
      "updatedAt": "2016-05-09T12:00:00.000Z"
    },
    "_id": "12345678-1234-1234-1234-123456789ABC",
    "_rev": "1-b1191b8cfee045f495594b1cf2823683"
}
```

Rejects with:

Add expected Errors: #102

Example: add single document

```
store.add({foo: 'bar'}).then(function (doc) {
    alert(doc.foo) // bar
}).catch(function (error) {
    alert(error)
})
```

Example: add multiple documents

```
store.add([{foo: 'bar'}, {bar: 'baz'}]).then(function (docs) {
    alert(docs.length) // 2
}).catch(function (error) {
    alert(error)
})
```

store.find(id)

```
store.find(id)
```

Argument	Type	Description	Required
id	String	Unique id of document	Yes

Resolves with properties

```
{
    "id": "12345678-1234-1234-1234-123456789ABC",
    "foo": "bar",
    "createdAt": "2016-05-09T12:00:00.000Z",
    "updatedAt": "2016-05-09T12:00:00.000Z"
}
```

Rejects with:

Add expected Errors: #102

Example

```
store.find('12345678-1234-1234-1234-123456789ABC').then(function (doc) {
    alert(doc.id)
}).catch(function (error) {
    alert(error)
})
```

store.find(doc)

```
store.find(doc)
```

Argument	Type	Description	Required
doc	Object	document with id property	Yes

Resolves with properties

```
{
    "id": "12345678-1234-1234-1234-123456789ABC",
    "foo": "bar",
    "createdAt": "2016-05-09T12:00:00.000Z",
    "updatedAt": "2016-05-09T12:00:00.000Z"
}
```

Rejects with:

Add expected Errors: #102

```
store.find(doc).then(function (doc) {
    alert(doc.id)
}).catch(function (error) {
    alert(error)
})
```

store.find(idsOrDocs)

```
store.find(idsOrDocs)
```

Argument	Type	Description	Required
idsOrDocs	Array	Array of id (String) or doc (Object) items	Yes

Resolves with array of properties

```
[{
    "id": "12345678-1234-1234-1234-123456789ABC",
    "foo": "bar",
    "createdAt": "2016-05-09T12:00:00.000Z",
    "updatedAt": "2016-05-09T12:00:00.000Z"
}]
```

Rejects with:

Add expected Errors: #102

Example

```
store.find(doc).then(function (doc) {
    alert(doc.id)
}).catch(function (error) {
    alert(error)
})
```

Testing

Local setup

```
git clone https://github.com/hoodiehq/hoodie-store-client.git
cd hoodie-store-client
npm install
```

In Node.js

Run all tests and validate JavaScript Code Style using standard

```
npm test
```

To run only the tests

```
npm run test:node
```

Run tests in browser

```
npm run test:browser:local
```

This will start a local server. All tests and coverage will be run at http://localhost:8080/__zuul

hoodie.connectionStatus

`hoodie-connection-status` is a browser library to monitor a connection status. It emits `disconnect` & `reconnect` events if the request status changes and persists its status.

Example

```
var connectionStatus = new ConnectionStatus('https://example.com/ping')

connectionStatus.on('disconnect', showOfflineNotification)
connectionStatus.on('reconnect reset', hideOfflineNotification)
myOtherRemoteApiThing.on('error', connectionStatus.check)
```

Constructor

```
new ConnectionStatus(options)
```

Argument	Type	Description	Required
`options.url`	String	Full url to send pings to	Yes
`options.method`	String	Defaults to *HEAD*. Must be valid http verb like `'GET'` or `'POST'` (case insensitive)	No
`options.interval`	Number	Interval in ms. If set a request is send immediately. The interval starts after each request response. Can also be set to an object to differentiate intervals by connection status, see below	No
`options.interval.connected`	Number	Interval in ms while `connectionStatus.ok` is not `false`. If set, a request is send immediately. The interval starts after each request response.	No
`options.interval.disconnected`	Number	Interval in ms while `connectionStatus.ok` is `false`. If set, a request is send immediately. The interval starts after each request response.	No
`options.cache`	Object or `false`	Object with `.get()`, `.set(properties)` and `.unset()` methods to persist the connection status. Each method must return a promise, `.get()` must resolve with the current state or an empty object. If set to `false` the connection status will not be persisted.	Defaults to a localStorage-based API
`options.cacheTimeout`	Number	time in ms after which a cache shall be invalidated. When invalidated on initialisation, a `reset` event gets triggered on next tick.	No

Example

```
var connectionStatus = new ConnectionStatus('https://example.com/ping')

connectionStatus.on('disconnect', showOfflineNotification)
connectionStatus.check()
```

connectionStatus.ready

Read-only

Promise that resolves once the ConnectionStatus instance loaded its current state from the cache.

connectionStatus.ok

Read-only

```
connectionStatus.ok
```

- Returns `undefined` if no status yet
- Returns `true` last check responded ok
- Returns `false` if last check failed

The state is persisted in cache.

connectionStatus.isChecking

Read-only

```
connectionStatus.isChecking
```

- Returns `undefined` if status not loaded yet, see *connectionStatus.ready*
- Returns `true` if connection is checked continuously
- Returns `false` if connection is not checked continuously

connectionStatus.check(options)

```
connectionStatus.check(options)
```

Argument	Type	Description	Required
`options.timeout`	Number	Time in ms after which a ping shall be aborted with a `timeout` error	No

Resolves without value.

Rejects with:

name	status	message
`TimeoutError`	0	Connection timeout
`ServerError`	*as returned by server*	*as returned by server*
`ConnectionError`	`undefined`	Server could not be reached

Example

```
connectionStatus.check()

.then(function () {
// Connection is good, connectionStatus.ok is true
})

.catch(function () {
// Cannot connect to server, connectionStatus.ok is false
})
```

connectionStatus.startChecking(options)

Starts checking connection continuously

```
connectionStatus.startChecking(options)
```

Argument	Type	Description	Required
`options.interval`	Number	Interval in ms. The interval starts after each request response. Can also be set to an object to differentiate interval by connection state, see below	Yes
`options.interval.connected`	Number	Interval in ms while `connectionStatus.ok` is not `false`. The interval starts after each request response.	No
`options.interval.disconnected`	Number	Interval in ms while `connectionStatus.ok` is `false`. The interval starts after each request response.	No
`options.timeout`	Number	Time in ms after which a ping shall be aborted with a `timeout` error.	No

Resolves without values.

Example

```
connectionStatus.startChecking({interval: 30000})
    .on('disconnect', showOfflineNotification)
```

connectionStatus.stopChecking()

Stops checking connection continuously.

```
connectionStatus.stopChecking()
```

Resolves without values. Does not reject.

connectionStatus.reset(options)

Clears status & cache, aborts all pending requests.

```
connectionStatus.reset(options)
```

`options` is the same as in *Constructor*

Resolves without values. Does not reject.

Example

```
connectionStatus.reset(options).then(function () {
    connectionStatus.ok === undefined // true
})
```

Events

disconnect	Ping fails and `connectionStatus.ok` isn't `false`
reconnect	Ping succeeds and `connectionStatus.ok` is `false`
reset	Cache invalidated on initialisation or `connectionStatus.reset()` called

Example

```
connectionStatus.on('disconnect', function () {})
connectionStatus.on('reconnect', function () {})
connectionStatus.on('reset', function () {})
```

Testing

Local setup

```
git clone git@github.com:hoodiehq/hoodie-connection-status.git
cd hoodie-connection-status
npm install
```

Run all tests and code style checks

```
npm test
```

Run all tests on file change

```
npm run test:watch
```

Run specific tests only

```
# run unit tests
node tests/specs

# run .check() unit tests
node tests/specs/check

# run walkthrough integration test
node tests/integration/walkthrough
```

hoodie.log

`hoodie-log` is a standalone JavaScript library for logging to the browser console. If available, it takes advantage of CSS-based styling of console log outputs.

Example

```
var log = new Log('hoodie')

log('ohaj!')
// (hoodie) ohaj!
log.debug('This will help with debugging.')
// (hoodie:debug) This will help with debugging.
log.info('This might be of interest. Or not.')
// (hoodie:info) This might be of interest. Or not.
log.warn('Something is fishy here!')
// (hoodie:warn) Something is fishy here!
log.error('oooops')
// (hoodie:error) oooops

var fooLog = log.scoped('foo')
```

```
fooLog('baz!')
// (hoodie:foo) baz!
```

Constructor

```
new Log(prefix)
// or
new Log(options)
```

Argument	Type	Description	Required
`prefix`	String	Prefix for log messages	Yes
`options.prefix`	String	Prefix for log messages	Yes
`options.level`	String	Defaults to `warn`. One of `debug`, `info`, `warn` or `error`. `debug` is the lowest level, and everything will be logged to the console. `error` is the highest level and nothing but errors will be logged.	No
`styles`	Boolean or Object	Defaults to `true`. If set to false, all log messages are prefixed by `(<prefix>:<log type>)`, e.g. `(fooprefix:warn) bar is not available.`. If set to true, styles are applied to the prefix. The styles can be customised, see below	No
`styles.default`	String	Defaults to `color: white; padding: .2em .4em; border-radius: 1em`. Base CSS styles for all log types	No
`styles.reset`	String	Defaults to `background: inherit; color: inherit`. Reset CSS styles, applied for message after prefix	No
`styles.log`	String	Defaults to `background: gray`. CSS Styles for default log calls without log level	No
`styles.debug`	String	Defaults to `background: green`. CSS Styles for debug logs	No
`styles.info`	String	Defaults to `background: blue`. CSS Styles for info logs	No
`styles.warn`	String	Defaults to `background: orange`. CSS Styles for warn logs	No
`styles.error`	String	Defaults to `background: red`. CSS Styles for error logs	No

Example

```
var log = new Log({
    prefix: 'hoodie',
    level: 'warn',
    styles: {
        default: 'color: white; padding: .2em .4em; border-radius: 1em',
        debug: 'background: green',
        log: 'background: gray',
        info: 'background: blue',
        warn: 'background: orange',
        error: 'background: red',
        reset: 'background: inherit; color: inherit'
    }
}
```

log.prefix

Read-only

```
log.prefix
```

Prefix used in log messages

Example

```
log = new Log('hoodie')
log.prefix // hoodie
log.warn("Something is fishy here!")
// (hoodie:warn) Something is fishy here!
```

log.level

One of `debug`, `info`, `warn` or `error`. `debug` is the lowest level, and everything will be logged to the console. `error` is the highest level and nothing but errors will be logged.

```
log.level
```

Example

```
log.level = 'debug'
log.debug('This will help with debugging.')
// (hoodie:debug) This will help with debugging.
log.level = 'info'
log.debug('This will help with debugging.')
// <no log>
log.level = 'foo'
// throws InvalidValue error
```

log()

Logs message to browser console. Accepts same arguments as console.log.

```
log("ohaj!")
```

log.debug()

Logs debug message to browser console if `level` is set to `debug`. Accepts same arguments as console.log.

```
log.debug('This will help with debugging.')
```

log.info()

Logs info message to browser console if `level` is set to `debug` or `info`. Accepts same arguments as console.log.

```
log.info('This might be of interest. Or not.')
```

log.warn()

Logs warning to browser console unless `level` is set to `error`. Accepts same arguments as console.log.

```
log.warn('Something is fishy here!')
```

log.error()

Logs error message to browser console. Accepts same arguments as console.log.

```
log.error('oooops')
```

log.scoped()

```
log.scoped(prefix)
```

Argument	Type	Description	Required
`prefix`	String	Prefix for log messages	Yes

Returns `log` API with extended `prefix`

Example

```
var log = new Log('hoodie')
log('ohaj!')
// (hoodie) ohaj!
var fooLog = log.scoped('foo')
fooLog('baz!')
// (hoodie:foo) baz!
```

Testing

Local setup

```
git clone git@github.com:hoodiehq/hoodie-log.git
cd hoodie-log
npm install
```

Run all tests and code style checks

```
npm test
```

Run all tests on file change

```
npm run test:watch
```

Run specific tests only

```
# run .debug() unit tests
node tests/specs/debug.js
```

This library, commonly called **Hoodie Client**, is what you'll be working with on the client side. It consists of:

- The Hoodie Client API, which has a couple of useful helpers

- The account API, which lets you do user authentication, such as signing users up, in and out
- The store API, which provides means to store and retrieve data for each individial user
- The connectionStatus API, which provides helpers for connectivity.
- The log API, which provides a nice API for logging all the things

The Hoodie Server API

The Hoodie Server API is currently work-in-progress. But you can have a look at the Account Server API and the Store Server API for a sneak peak.

CHAPTER 8

Contributing to Hoodie

Please take a moment to review this document in order to make the contribution process easy and effective for everyone involved.

Following these guidelines helps to communicate that you respect the time of the developers managing and developing this open source project. In return, they should reciprocate that respect in addressing your issue, assessing changes, and helping you finalize your pull requests.

As for everything else in the project, the contributions to Hoodie are governed by our Code of Conduct.

Using the issue tracker

First things first: **Do NOT report security vulnerabilities in public issues!** Please disclose responsibly by letting the Hoodie team know upfront. We will assess the issue as soon as possible on a best-effort basis and will give you an estimate for when we have a fix and release available for an eventual public disclosure.

The issue tracker is the preferred channel for *bug reports*, *features requests* and *submitting pull requests*, but please respect the following restrictions:

- Please **do not** use the issue tracker for personal support requests. Use the Hoodie Chat.
- Please **do not** derail or troll issues. Keep the discussion on topic and respect the opinions of others.

Bug reports

A bug is a *demonstrable problem* that is caused by the code in the repository. Good bug reports are extremely helpful - thank you!

Guidelines for bug reports:

1. **Use the GitHub issue search** — check if the issue has already been reported.
2. **Check if the issue has been fixed** — try to reproduce it using the latest `master` or `next` branch in the repository.

3. **Isolate the problem** — ideally create a reduced test case.

A good bug report shouldn't leave others needing to chase you up for more information. Please try to be as detailed as possible in your report. What is your environment? What steps will reproduce the issue? What OS experiences the problem? What would you expect to be the outcome? All these details will help people to fix any potential bugs.

Example:

> Short and descriptive example bug report title
>
> A summary of the issue and the browser/OS environment in which it occurs. If suitable, include the steps required to reproduce the bug.
>
> 1. This is the first step
> 2. This is the second step
> 3. Further steps, etc.
>
> `<url>` - a link to the reduced test case
>
> Any other information you want to share that is relevant to the issue being reported. This might include the lines of code that you have identified as causing the bug, and potential solutions (and your opinions on their merits).

Feature requests

Feature requests are welcome. But take a moment to find out whether your idea fits with the scope and aims of the project. It's up to *you* to make a strong case to convince the project's developers of the merits of this feature. Please provide as much detail and context as possible.

Pull requests

Good pull requests - patches, improvements, new features - are a fantastic help. They should remain focused in scope and avoid containing unrelated commits.

Please ask first before embarking on any significant pull request (e.g. implementing features, refactoring code), otherwise you risk spending a lot of time working on something that the project's developers might not want to merge into the project.

For new Contributors

If you never created a pull request before, welcome :tada: :smile: Here is a great tutorial on how to send one :)

1. Fork the project, clone your fork, and configure the remotes using command line:

```
# Clone your fork of the repo into the current directory
git clone https://github.com/<your-username>/<repo-name>

# Navigate to the newly cloned directory
cd <repo-name>

# Assign the original repo to a remote called "upstream"
git remote add upstream https://github.com/hoodiehq/<repo-name>
```

2. If you cloned a while ago, get the latest changes from upstream:

```
git checkout master    git pull upstream master
```

3. Create a new topic branch (off the main project development branch) to contain your feature, change, or fix:

```
git checkout -b <topic-branch-name>
```

4. Make sure to update, or add to the tests when appropriate. Patches and features will not be accepted without tests. Run `npm test` to check that all tests pass after you've made changes. Look for a `Testing` section in the project's README for more information.
5. If you added or changed a feature, make sure to document it accordingly in the `README.md` file.
6. Push your topic branch up to your fork:

```
git push origin <topic-branch-name>
```

8. Open a Pull Request with a clear title and description.

For Members of the Hoodie Contributors Team

1. Clone the repo and create a branch

```
git clone https://github.com/hoodiehq/<repo-name>
cd <repo-name>
git checkout -b <topic-branch-name>
```

2. Make sure to update, or add to the tests when appropriate. Patches and features will not be accepted without tests. Run `npm test` to check that all tests pass after you've made changes. Look for a `Testing` section in the project's README for more information.
3. If you added or changed a feature, make sure to document it accordingly in the `README.md` file.
4. Push your topic branch up to our repo

```
git push origin <topic-branch-name>
```

5. Open a Pull Request using your branch with a clear title and description.

Optionally, you can help us with these things. But don't worry if they are too complicated, we can help you out and teach you as we go :)

1. Update your branch to the latest changes in the upstream master branch. You can do that locally with

```
git pull --rebase upstream master
```

Afterwards force push your changes to your remote feature branch.

2. Once a pull request is good to go, you can tidy up your commit messages using Git's interactive rebase. Please follow our commit message conventions shown below, as they are used by semantic-release to automatically determine the new version and release to npm. In a nutshell:

Commit Message Conventions

- Commit test files with `test: ...` or `test(scope): ...` prefix
- Commit bug fixes with `fix: ...` or `fix(scope): ...` prefix
- Commit breaking changes by adding `BREAKING CHANGE:` in the commit body (not the subject line)

- Commit changes to `package.json`, `.gitignore` and other meta files with `chore(filenamewithoutext): ...`
- Commit changes to README files or comments with `docs: ...`
- Cody style changes with `style: standard`

IMPORTANT: By submitting a patch, you agree to license your work under the same license as that used by the project.

Triagers

There is a defined process to manage issues, because this helps to speed up releases and minimizes user pain. Triaging is a great way to contribute to Hoodie without having to write code. If you are interested, please leave a comment here asking to join the triaging team.

Maintainers

If you have commit access, please follow this process for merging patches and cutting new releases.

Reviewing changes

1. Check that a change is within the scope and philosophy of the component.
2. Check that a change has any necessary tests.
3. Check that a change has any necessary documentation.
4. If there is anything you don't like, leave a comment below the respective lines and submit a "Request changes" review. Repeat until everything has been addressed.
5. If you are not sure about something, mention `@hoodie/maintainers` or specific people for help in a comment.
6. If there is only a tiny change left before you can merge it and you think it's best to fix it yourself, you can directly commit to the author's fork. Leave a comment about it so the author and others will know.
7. Once everything looks good, add an "Approve" review. Don't forget to say something nice
8. If the commit messages follow our conventions
9. If there is a breaking change, make sure that `BREAKING CHANGE:` with *exactly* that spelling (incl. the ":") is in body of the according commit message. This is *very important*, better look twice :)
10. Make sure there are `fix: ...` or `feat: ...` commits depending on whether a bug was fixed or a feature was added. **Gotcha:** look for spaces before the prefixes of `fix:` and `feat:`, these get ignored by semantic-release.
11. Use the "Rebase and merge" button to merge the pull request.
12. Done! You are awesome! Thanks so much for your help
13. If the commit messages *do not* follow our conventions
14. Use the "squash and merge" button to clean up the commits and merge at the same time:
15. Is there a breaking change? Describe it in the commit body. Start with *exactly* `BREAKING CHANGE:` followed by an empty line. For the commit subject:

16. Was a new feature added? Use `feat: ...` prefix in the commit subject
17. Was a bug fixed? Use `fix: ...` in the commit subject

Sometimes there might be a good reason to merge changes locally. The process looks like this:

Reviewing and merging changes locally

```
git checkout master # or the main branch configured on github
git pull # get latest changes
git checkout feature-branch # replace name with your branch
git rebase master
git checkout master
git merge feature-branch # replace name with your branch
git push
```

When merging PRs from forked repositories, we recommend you install the hub command line tools.

This allows you to do:

```
hub checkout link-to-pull-request
```

meaning that you will automatically check out the branch for the pull request, without needing any other steps like setting git upstreams! :sparkles:

CHAPTER 9

Coding Style Guide

Please see Contributing to Hoodie for more guidelines on contributing to Hoodie.

Hoodie uses the Standard JavaScript coding style.

This file explains coding-style considerations that are beyond the syntax check of *Standard*.

There are three sections:

- *General*: coding styles that are applicable to all JavaScript code.
- *Client*: coding styles that are only applicable to in-browser code.
- *Server*: coding styles that are only applicable in server code.

Note: Client and Server coding styles can be contradicting, make sure to read these carefully.

General

File Structure

A typical JavaScript file looks like this (without the comments). Sort all modules that you `require` alphabetically within their blocks.

```
// If your module exports something, put it on top
module.exports = myMethod

// require Node.js core modules in the 1st block (separaeted by empty line).
// These are modules that come with Node.js and are not listed in package.json.
// See https://nodejs.org/api/ for a list of Node.js core modules
var EventEmitter = require('events').EventEmitter
var util = require('util')

// In the 2nd block, require all modules listed in package.json
var async = require('async')
var lodash = require('lodash')
```

```
// in the 3rd block, require all modules using relative paths
var helpers = require('./utils/helpers')
var otherMethod = require('./other-method')

function myMethod () {
  // code here
}
```

Avoid "this" and object-oriented coding styles.

Do this

```
function MyApi (options) {
  var state = {
    foo: options.foo
  }
  return {
    doSomething: doSomething.bind(null, state)
  }
}

function doSomething (state) {
  return state.foo ? 'foo!' : 'bar'
}
```

Instead of

```
function MyApi (options) {
  this.foo = options.foo
}

MyApi.prototype.doSomething = function () {
  return this.foo ? 'foo!' : 'bar'
}
```

The bind method allows for partially applied functions, that way we can pass internal state between different methods without exposing in the public API. At the same time we can easily test the different methods in isolation by setting the internal state to what ever context we want to test with.

Folder Structure

In the root, have

- `package.json`
- `.gitignore` (should at least list node_modules)
- `README.md`
- `LICENSE` (Apache License Version 2.0)

In most cases you will have `index.js` file which is listed in `package.json` as the `"main"` property.

If you want to split up logic into separate files, move them into a `server/` folder. Put reusable, state-less helper methods into `server/utils/`

For tests, create a `test/` folder. If your module becomes a bit more complex, split up the tests in `test/unit` and `test/integration/`. All files that contain tests should end with `-test.js`.

Misc

- Prefer lodash over underscore.

Client

Testing

Client code should be tested using tape. The reason we use tape is its support for browserify.

Libraries with sub-modules that can be required individually, like lodash

For client-side JavaScript code, it is important to limit the amount of code that is downloaded to the client to the code that is actually needed. The loadash library is a collection of utilities that are useful individually and in combination.

For example, if you want to use the `merge` function of lodash, require it like this:

```
var merge = require('lodash/merge')
```

If you want to use more than one function within one module, or if you want to combine multiple functions for a single operation, require the full lodash module:

```
var _ = require('lodash')
```

If multiple modules use the same lodash function, our frontend bundling tool will do the right thing and only include that code once.

Server

Testing

Server code should be tested using tap.

Libraries with sub-modules that can be required individually, like lodash

For server-side code, it is important to load the minimal amount of code into memory.

On the server require the full library, e.g.

```
var _ = require('lodash')

var c = _.merge(a, b)
```

That way, all of our server code will only ever load a single instance of lodash into memory.

CHAPTER 10

Triage new issues/PRs on GitHub

This document illustrates the steps the Hoodie community is taking to triage issues. The labels are used later on for *assigning work*. If you want to help by sorting issues please leave a comment here asking to join the triaging team.

Triaging Process

This process based on the idea of minimizing user pain from this blog post.

1. Open the list of non triaged issues
    - Sort by submit date, with the newest issues first
    - You don't have to do issues in order; feel free to pick and choose issues as you please.
    - You can triage older issues as well
    - Triage to your heart's content
2. Assign yourself: Pick an issue that is not assigned to anyone and assign it to you
3. Understandable? - verify if the description of the request is clear.
    - If not, *close it* according to the instructions below and go to the last step.
4. Duplicate?
    - If you've seen this issue before *close it*, and go to the last step.
    - Check if there are comments that link to a dupe. If so verify that this is indeed a dupe, *close it*, and go to the last step.
5. Bugs:
    - Label `Type: Bug`
    - Reproducible? - Steps to reproduce the bug are clear. If they are not, ask for a clarification. If there's no reply after a week, *close it*.
    - Reproducible on master?

6. Non bugs:
    - Label `Type: Feature`, `Type: Chore`, or `Type: Perf`
    - Belongs in core? – Often new features should be implemented as a plugin rather than an addition to the core. If this doesn't belong, *close it*, and go to the last step.
    - Label `needs: breaking change` - if needed
    - Label `needs: public api` - if the issue requires introduction of a new public API
7. Label `frequency: *` – How often does this issue come up? How many developers does this affect?
    - low - obscure issue affecting a handful of developers
    - moderate - impacts a common usage pattern
    - high - impacts most or all Hoodie apps
8. Label `severity: *` - How bad is the issue?
    - regression
    - memory leak
    - broken expected use - it's hard or impossible for a developer using Hoodie to accomplish something that Hoodie should be able to do
    - confusing - unexpected or inconsistent behavior; hard-to-debug
    - inconvenience - causes ugly/boilerplate code in apps
9. Label `starter` - These issues are good targets for PRs from the open source community. Apply to issues where the problem and solution are well defined in the comments, and it's not too complex.
10. Label `milestone: *` – Assign a milestone:
    - Backlog - triaged fixes and features, should be the default choice
    - x.y.z - e.g. 0.3.0
1. Unassign yourself from the issue

Closing an Issue or PR

We're grateful to anyone who takes the time to submit an issue, even if we ultimately decide not to act on it. Be kind and respectful as you close issues. Be sure to follow the code of conduct.

1. Always thank the person who submitted it.
2. If it's a duplicate, link to the older or more descriptive issue that supersedes the one you are closing.
3. Let them know if there's some way for them to follow-up.
    - When the issue is unclear or reproducible, note that you'll reopen it if they can clarify or provide a better example. Mention jsbin for examples. Watch your notifications and follow-up if they do provide clarification. :)
    - If appropriate, suggest implementing a feature as a third-party module.

If in doubt, ask a core team member what to do.

Example:

> Thanks for submitting this issue! Unfortunately, we don't think this functionality belongs in core. The good news is that you could implement this as a plugin and publish it to npm with the `hoodie-plugin` keyword.

Assigning Work

These criteria are then used to calculate a "user pain" score. Work is assigned weekly to core team members starting with the highest pain, descending down to the lowest.

```
pain = severity × frequency
```

severity:

- regression (5)
- memory leak (4)
- broken expected use (3)
- confusing (2)
- inconvenience (1)

frequency:

- low (1)
- moderate (2)
- high (3)

Note: Regressions and memory leaks should almost always be set to `frequency: high`.

CHAPTER 11

Contributing to Documentation

This guide describes how to make changes to Hoodie documentation.

Make small changes

We love small contributions, if you spot small errors or additions please feel free to request a change. Every page on Hoodie documentation has an "Edit on GitHub" button on the top right corner, please use this to make changes.

Hoodie documentation uses the reStructuredText format. This may be unfamiliar but provides advanced features which are useful for complex documentation.

The Github editor is very basic, if you need more editing tools try copying and pasting into this online editor. You can then click 'commit' and create a 'pull request' on Github. The pull request will be automatically tested for grammar, style and common misspellings. Your changes will then be reviewed by a Hoodie Admin, who may suggest changes. Please read the Documentation Style Guide for advice on writing and more info on testing.

Make big changes

For big changes, follow the Contributing to Hoodie guidelines for new contributors. This allows you to build and test the documentation locally. For example, adding, moving or updating several documents. The index.rst file in the docs/ folder controls the order in which the documents are displayed on the docs webpages. Remember to update the index file if you have removed, added or want to reorder the documents.

To build the docs locally, you will need to install python 2.7+

Then install two pip packages: Sphinx and sphinx_rtd_theme.

```
sudo pip install sphinx
```

```
sudo pip install sphinx_rtd_theme
```

Change directory to ..hoodie/docs/

```
make html
```

If you are using windows powershell, note there is a little deviation.

```
pip install sphinx

pip install sphinx_rtd_theme
```

Before execute the `make html` command you have to install make in windows if you are not already done. You can also see this Stackoverflow link for a clear understanding.

Now change directory to ..hoodie/docs/

```
make html
```

After building, your updated documents are in the docs/_build/html subdirectory. Click on any .html document, this will open your web browser and the documents will be viewable.

Get in touch if you have any questions or want to contribute to Hoodie documentation.

CHAPTER 12

Documentation Style Guide

This guide provides style advice for how to write documentation. Please take the time to read this before contributing a large change or update to documentation.

Style helps you and your reader

Word choice and writing style are a personal choice and we understand documentation can be difficult to write. These recommendations have been designed to help you write clear and beautiful documents.

Testing

The contributing to docs guide describes the process to follow when updating documentation. This process includes automatic testing. Testing provides you peace of mind that your contribution won't contain typos, broken links or other style whoopsies. Testing is not used to criticise your writing, we really love and appreciate any contributions. Please be patience through the testing and review process. Together we can keep Hoodie documentation awesome!

Style guidance

Please see the helpful guide provided by OpenStack documentation. This guide will further explain these key style tips:

- Use standard English
- Write in active voice
- Use the present simple tense
- Write in second person
- Use appropriate mood

- Keep sentences short
- Avoid ambiguous titles
- Be clear and concise
- Write objectively
- Describe the most common use case first
- Do not humanize inanimate objects
- Write positively
- Avoid prepositions at the end of sentences
- Do no overuse this, that, these, and it
- Do not split infinitives
- Avoid personification
- Eliminate needless politeness
- Use consistent terminology
- Use spelling and grammar checking tools

Automatic testing

The current tests we run on pull requests using Travis Continuous Integration (CI) service:

Style guide	Tested	Test type	Package
Keep sentences short, concise and readable		Warning	rousseau
Write in the active voice		Warning	rousseau
Avoid “Lexical illusion’s” – cases where a word is repeated		Warning	rousseau
Check for ‘So’ at the beginning of sentences		Warning	rousseau
Avoid adverbs that can weaken meaning: really, very, extremely, etc		Warning	rousseau
Use the most simple expressions		Warning	rousseau
Avoid using “weasel words”: quite, several, mostly etc		Warning	rousseau
Leave no space between a sentence and its ending punctuation		Warning	rousseau
Spell checker - we test for common misspelling but please check technical words		Error	common
Broken or dead links (excluding redirects)		Error	awesome

- Remember, follow the Code of Conduct

Bonus style points

- Be fun and friendly as long as it does not distract or confuse the reader
- Include videos or gifs to demostrated a feature
- You can use Humour but remember the reader is looking for an *answer* not a comedy sketch
- Cultural references and puns don’t always translate - keep jokes light
- Remember English is not the first language for many readers - keep language simple where possible

Further reading

This guide is influenced by the Open Stack style guide.

CHAPTER 13

Hoodie's Concepts

Hoodie was designed around a few core beliefs and concepts, and they explain a lot if the choices made in the code and the functionality. They are:

- *Dreamcode*
- *noBackend*
- *Offline First*

Dreamcode

While designing Hoodie's API, we realised that we wanted to do more than simply expose some server code to the frontend. **We wanted to reduce complexity, not move it around**. And to make something simple and intuitive, you can't start with the tech stack, you have to start with the humans that are going to use it. What would their dream API look like? Dreamcode is essentially user-centered design for APIs.

To put it bluntly: **Hoodie's API is optimized for being awesome**. For being intuitive and accessible. And it's optimized for making the lives of frontend developers as good as possible. It's also an API first: it's a promise - everything else can change or is replaceable. The API is all that matters.

Forget all the constraints of today's browsers. Then write down the code of your dreams for all the tasks you need to build your app. The implementation behind the API doesn't matter, it can be simple or tough as nails, but crucially: the users shouldn't have to care. This is dreamcode.

Everything is hard until someone makes it easy. We're making web app development easy.

Here's some further information and links to Dreamcode examples.

noBackend

Servers are difficult. Databases are difficult. The interplay between client and server is difficult, there are many moving parts, there are many entertaining mistakes to make, and **the barrier to entry for web app development is, in our**

mind, needlessly high. You shouldn't have to be a full stack developer to build a functioning app prototype, or code a small tool for yourself or your team, or launch a simple MVP.

People have been building web apps for quite a while now, and their basic operations (sign up, sign in, sign out, store and retrieve data, etc.) must have been written a million separate times by now. These things really shouldn't be difficult anymore. So we're proposing Hoodie as a noBackend solution. Yes, a backend does exist, but it doesn't have to exist in your head. You don't have to plan it or set it up. You simply don't have to worry about it for those basic operations, you can do all of them with Hoodie's frontend API. Of yourse, we let you dig as deep as you want, but for the start, you don't have to.

noBackend gives you time to work on the hard problems, the parts of the app that are justifiably difficult and non-abstractable, like the interface, the user experience, the things that make your product what it is.

With Hoodie, you scaffold out your app with

and you're good to go. Sign up users, store data... it's all right there, immediately. It's a backend in a box, empowering frontend developers to build entire apps without thinking about the backend at all. Check out some example Hoodie apps if you'd like to see some code.

More information about noBackend

See nobackend.org, Examples for noBackend solutions and @nobackend on Twitter.

Offline First

We make websites and apps for the web. The whole point is to be online, right? We're online when we build these things, and we generally assume our users to be in a state of permanent connectivity. That state, however, is a myth, and that assumption causes all sorts of problems.

With the stellar rise of mobile computing, we can no longer assume anything about our users' connections. Just as we all had to learn to accept that screens now come in all shapes and sizes, **we'll have to learn that connections can be present or absent, fast or slow, steady or intermittent, free or expensive...** We reacted to the challenge of unknowable screen sizes with Responsive Webdesign and Mobile First, and we will react to the challenge of unknowable connections with Offline First.

Offline First means: build your apps without the assumption of permanent connectivity. Cache data and apps locally. Build interfaces that accommodate the offline state elegantly. Design user interactions that will not break if their train goes into a tunnel. Don't freak out your users with network error messages or frustrate them with inaccessible data. **Offline First apps are faster, more robust, more pleasant to use, and ultimately: more useful.**

More information about Offline First

See offlinefirst.org, on GitHub and discussions and research

So now you know what motivates us

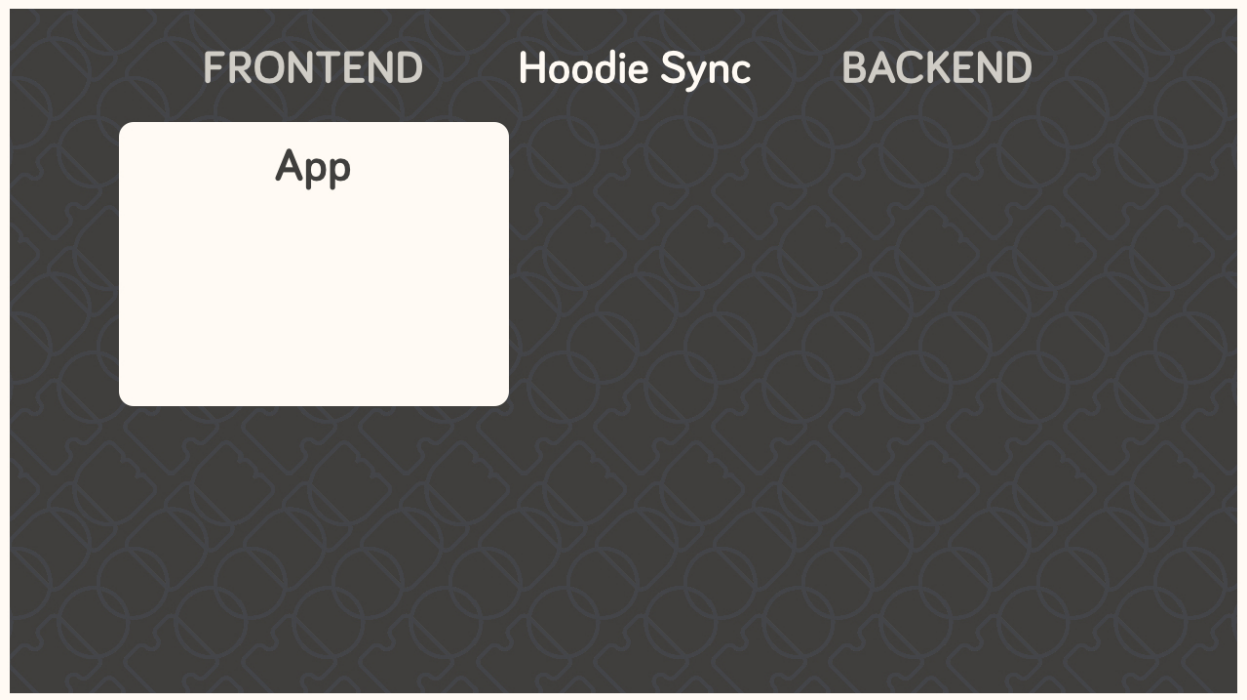
We hope this motivated you too! So let's continue to the system requirements for Hoodie.

CHAPTER 14

How Hoodie Works

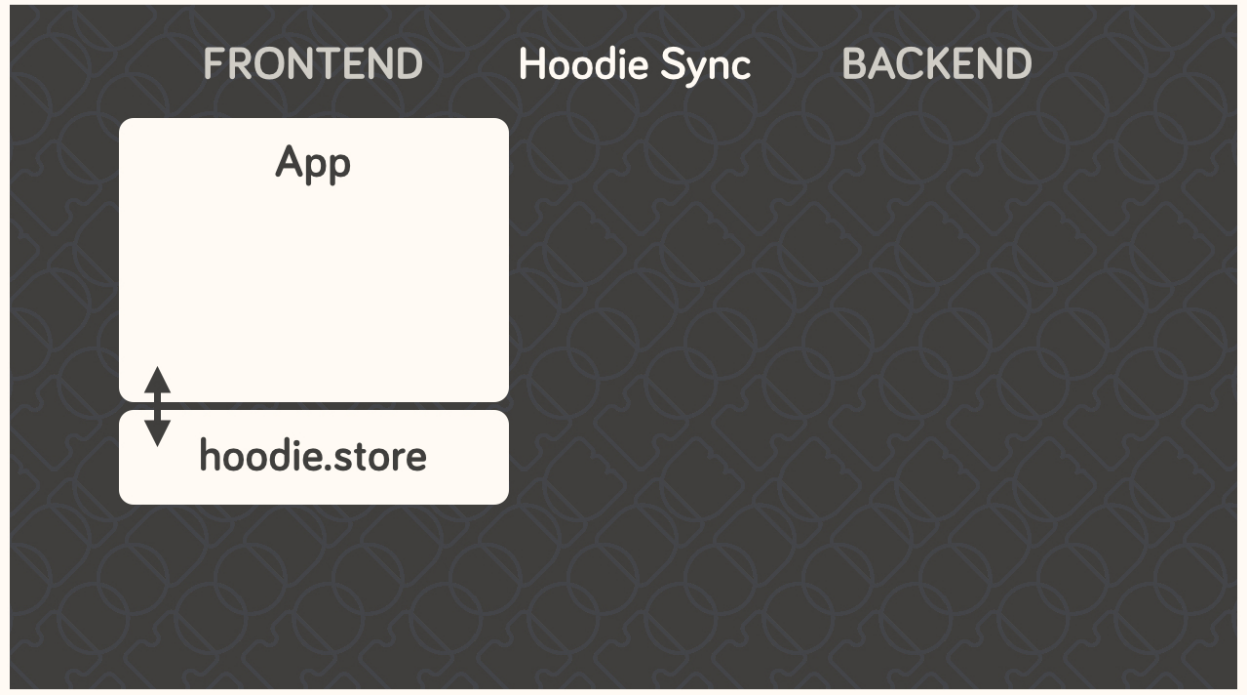
Hoodie has several components that work together in a somewhat atypical way to deliver our promise of simplicity, out-of-the-box syncing, and offline capability.

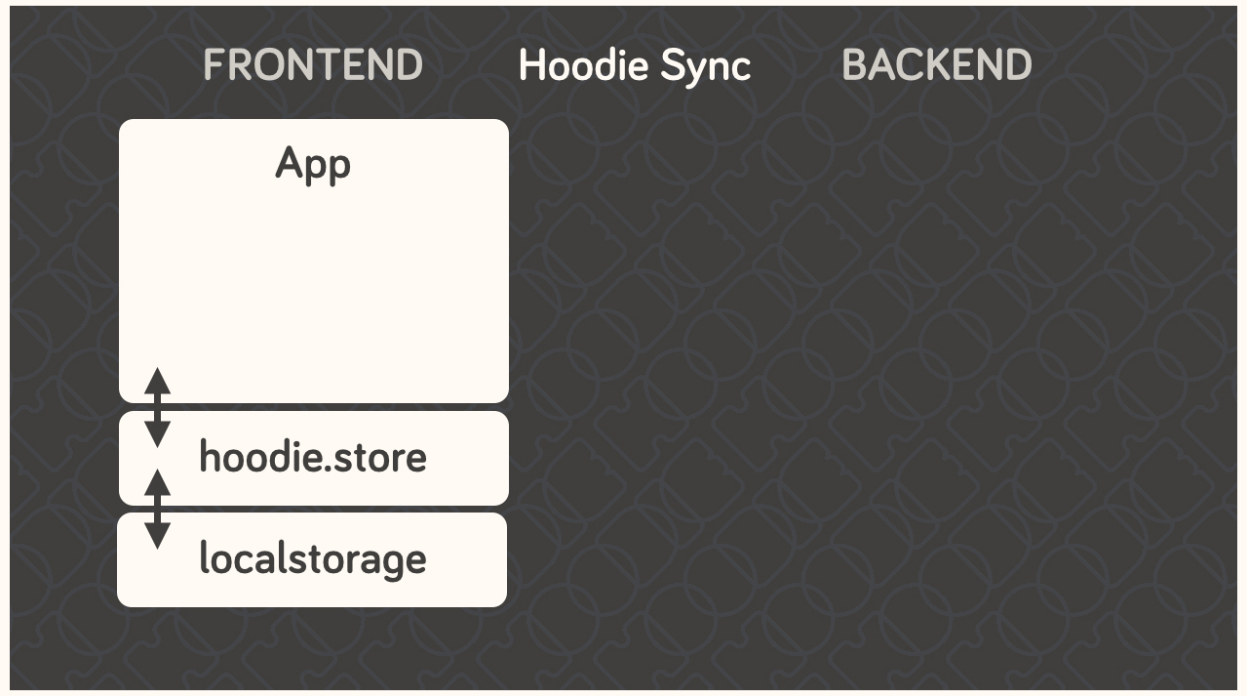
Everything starts in the frontend, with your app. This is your user interface, your client side business logic, etc.



The app code only talks to the Hoodie frontend API, never directly to the server-side code, the database, or even the in-browser storage.

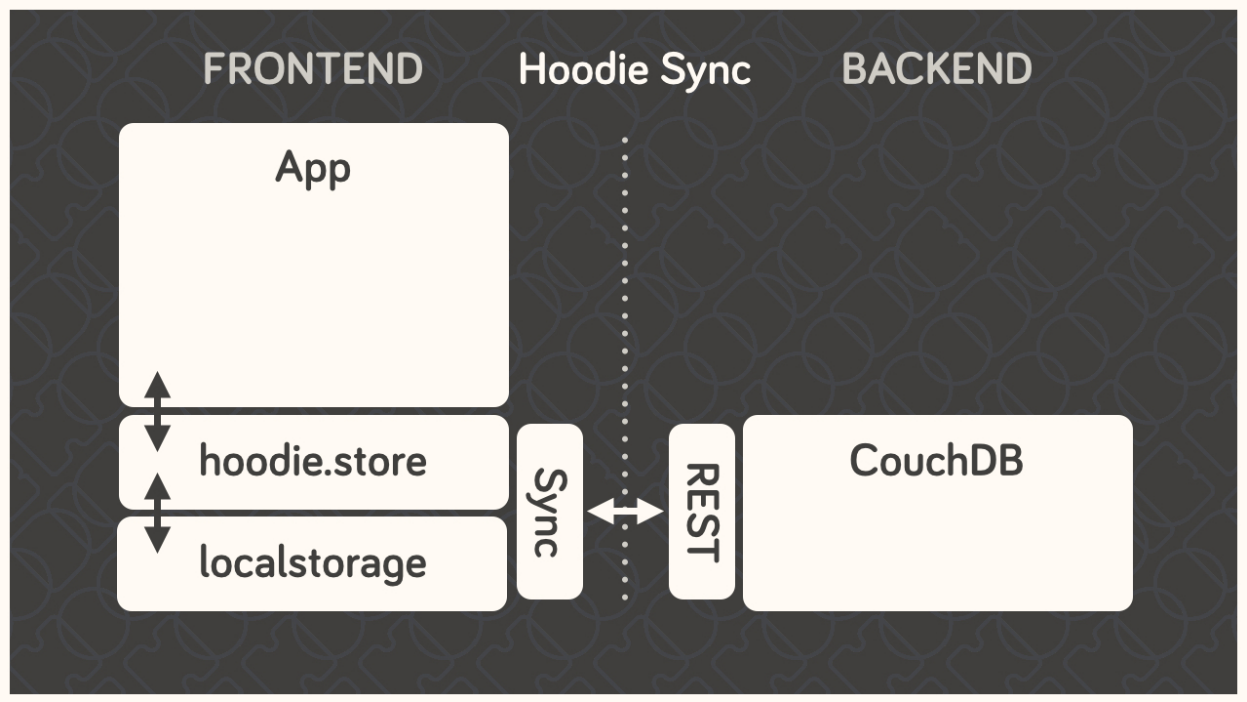
Hoodie uses PouchDB for storing data locally, which uses IndexedDb or WebSQL, whatever is available. Hoodie saves all data here first, before doing anything else. So if you're offline, your data is safely stored locally.


FRONTEND
Hoodie Sync
BACKEND
App
hoodie.store


FRONTEND
Hoodie Sync
BACKEND
App
hoodie.store
localstorage

This, by itself, is already enough for an app. But if you want to save your data remotely or send an email, for example, you'll need a bit more.

Hoodie relies on CouchDB, the database that replicates. We use it to sync data back and forth between the server and the clients, which is something that CouchDB happens to be really good at.

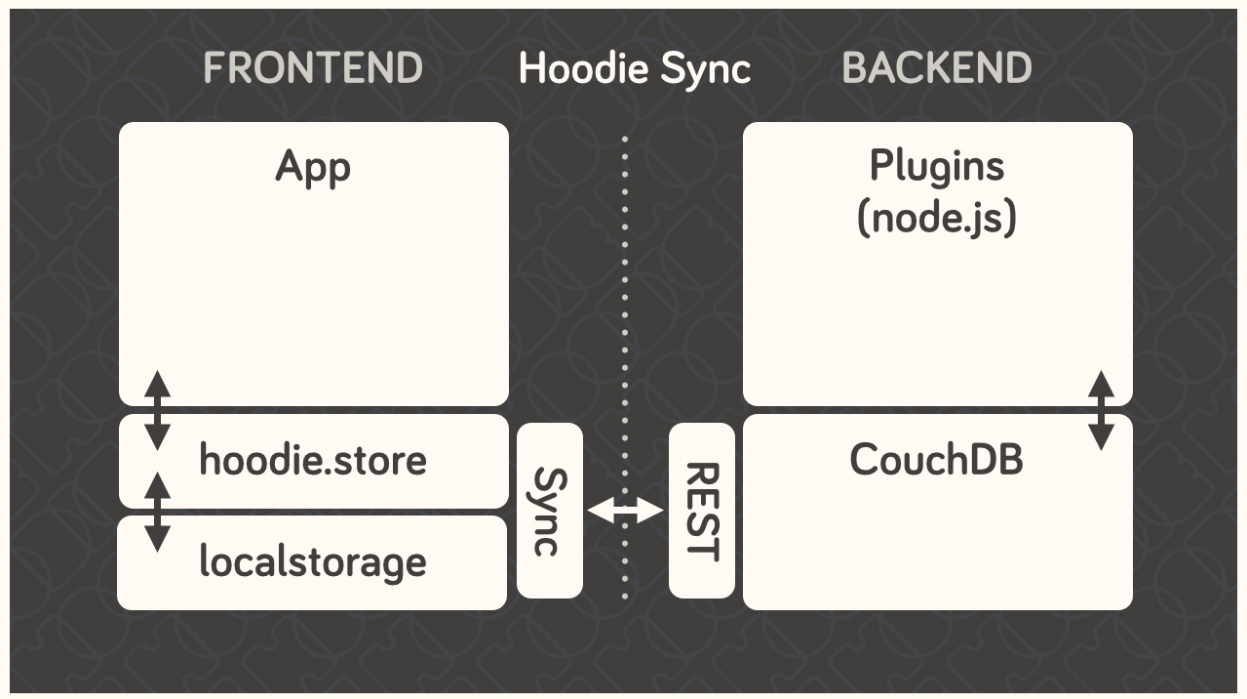


A small aside: In CouchDB, each user has their own private database which only they can access, so all user data is private by default. It can be shared to the public if the user decides to do so, but it can't happen by accident. This is why we'll often mention sharing and global data as a separate feature.

Behind the database, we have the actual server code in the form of a small node.js core with various plugins running alongside it. These then act upon the data in the CouchDB, which then replicates the changes back to the clients.

So Hoodie does **client database server** instead of the traditional **client server database**, and this is where many of its superpowers come from.

The clever bit is indicated by the dotted line in the middle; the connection between clients and server can be severed at any time without breaking the system. Frontend and backend never talk directly to each other. They only leave each other messages and tasks. It's all very loosely-coupled and event-based, and designed for eventual consistency.


FRONTEND
Hoodie Sync
BACKEND
App
Plugins
(node.js)
hoodie.store
localstorage
Sync
REST
CouchDB

CHAPTER 15

Architecture

After installing hoodie, `npm start` will run cli/index.js which reads out the configuration from all the different places using the rc package, then passes it as options to `server/index.js`, the Hoodie core hapi plugin.

In server/index.js, the passed options are merged with defaults and parsed into configuration for the Hapi server. It passes the configuration on to hoodie-server, which combines the core server modules. It also bundles the Hoodie client on first request to `/hoodie/client.js` and passes in the configuration for the client. It also makes the app's `public` folder accessible at the `/` root path, and Hoodie's Core UIs at `/hoodie/admin`, `/hoodie/account` and `/hoodie/store`.

Hoodie uses CouchDB for data persistence. If `options.dbUrl` is not set, it falls back to PouchDB.

Once all configuration is taken care of, the internal plugins are initialised (see server/plugins/index.js). We define simple Hapi plugins for logging and for serving the app's public assets and the Hoodie client.

Once everything is setup, the server is then started at the end of cli/start.js and the URL where hoodie is running is logged to the terminal.

Modules

Hoodie is a server built on top of hapi with frontend APIs for account and store related tasks. It is split up in many small modules with the goal to lower the barrier to new code contributors and to share maintenance responsibilities.

1. server

   Hoodie's core server logic as hapi plugin. It integrates Hoodie's server core modules: account-server, store-server

   (a) account-server

   Hapi plugin that implements the Account JSON API routes and exposes a corresponding API at `server.plugins.account.api.*`.

   (b) store-server

Hapi plugin that implements CouchDB's Document API. Compatible with CouchDB and PouchDB for persistence.

2. client

   Hoodie's front-end client for the browser. It integrates Hoodie's client core modules: account-client, store-client, connection-status and log

   (a) account-client

   Client for the Account JSON API. It persists session information on the client and provides front-end friendly APIs for things like creating a user account, confirming, resetting a password, changing profile information, or closing the account.

   (b) store-client

   Store client for data persistence and offline sync.

   (c) connection-status

   Browser library to monitor a connection status. It emits `disconnect` & `reconnect` events if the request status changes and persists its status on the client.

   (d) log

   JavaScript library for logging to the browser console. If available, it takes advantage of CSS-based styling of console log outputs.

5. admin

   Hoodie's built-in Admin Dashboard, built with Ember.js

   (a) admin-client

   Hoodie's front-end admin client for the browser. Used in the Admin Dashboard, but can also be used standalone for custom admin dashboard.

CHAPTER 16

Files & Folders

package.json

The `package.json` file describes your project in JSON. It its dependencies, scripts needed to run or use or develop on your project, and other information described in the NPM documentation for package.json.

The file is created when you run this:

```
npm init
```

It will prompt you for details about your projects, such as its name, version, description, test suite, author, and software license. Fill it out or leave it for later. You can always edit the file directly.

Hoodie modifies your package.json when it is installed to add a "start" script that starts your server by running *hoodie* without options.

When adding new dependencies, you can save their name and version information to `package.json` by using the `--save` flag, like this:

```
npm install --save <package name>
```

With your dependencies documented in `package.json`, you can install all your dependencies at once by running this:

```
npm install
```

This makes it very easy for others to get your project up and running quickly.

README.md

The `README.md` file describes your project in Markdown. It is intended for humans to read, and should include information about what your project is or does, how to install it, use it, test it, and contribute to it if appropriate.

For an example readme, try the one used by Hoodie :)

.hoodie/

The `.hoodie/` folder contains compiled client assets and database records, including query indexes. You should never need to modify these files directly.

hoodie/

The `hoodie/` folder contains the JavaScript code that runs in your server and the user's browser, and the code that they share. Hoodie uses two files as hooks to package code for the client and server:

- `hoodie/client/index.js` is included as a Hoodie plugin using Browserify, so it can use `require()` to include code from dependencies or other folders.
- `hoodie/server/index.js` is included in the server as a Hapi plugin. It can define new routes and other server-side logic.

Hoodie does not create a `hoodie/` folder, so you will need to create it:

```
mkdir hoodie
mkdir hoodie/{client,server}
touch hoodie/{client,server}/index.js
```

Although Hoodie doesn't treat it in any special way, you can use a folder like `hoodie/lib/` to store code shared between the client and the server. Client and server scripts can `require()` code from other folders like `hoodie/lib/`.

The `hoodie/client/index.js` file exports a Hoodie plugin. A Hoodie plugin exports a function that accepts a 'hoodie' object as its sole parameter. This object contains the interfaces to Hoodie's client APIs: 'account', 'store', 'connectionStatus', and 'log'.

You can also attach new methods to the 'hoodie' object, like the 'hello' method in this example `hoodie/client/index.js` file:

```
module.exports = function (hoodie) {
  hoodie.hello = function (what) {
    return Promise.resolve('Hello, ' + (what || 'world') + '!')
  }
}
```

The `hoodie/server/index.js` exports a Hapi plugin, like this:

```
module.exports.register = function (server, options, next) {
  server.route({
    method: 'GET',
    path: '/hello',
    handler: function (request, reply) {
      reply({ hello: 'world' })
    }
  })
  next()
}

module.exports.register.attributes = {
  name: '<app name>',
  version: '<app version>'
}
```

In this example, the register function is used to add a route to the server at `/hoodie/<app name>/hello` that responds with a JSON object like this: `{ "hello": "world" }`. All of your app's server routes are prefixed with `/hoodie/<app name>/`.

The 'register' method allows you to modify the server by adding routes and other server logic. You can read more about how to do that on Hapi's website. You can access Hoodie's server-side libraries via `server.plugins`.

public/

When you open your app in the browser you will see Hoodie's default page telling you that your app has no **public/** folder. So let's create it

```
mkdir public
touch public/index.html
```

Now edit the **public/index.html** file and pass in the following content.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My Hoodie App</title>
  </head>
  <body>
    <h1>My Hoodie App</h1>

    <script src="/hoodie/client.js"></script>
  </body>
</html>
```

You need to stop the server now (**ctrl + c**) and start it again. If you reload your app in your browser, you will now see your HTML file.

CHAPTER 17

Requirements

Before you start working with Hoodie, here's what you need to know regarding your development/server environment and the browsers Hoodie will run in.

System Requirements for Hoodie Server

- Mac OSX
- Windows 7 and up
- Linux (Ubuntu, Fedora 19+)

Browser Compatibilities (all latest stable)

- Firefox (29+)
- Chrome (34+)
- Desktop Safari (7+)
- Internet Explorer 10+
- Opera (21+)
- Android 4.3+
- iOS Safari (7.1+)

Important: This list is currently based on PouchDB's requirements, since Hoodie is using PouchDB for its in-browser storage.

CHAPTER 18

Glossary

CouchDB

CouchDB is a non-relational, document-based database that replicates, which means it's really good at syncing data between multiple instances of itself. All data is stored as JSON, all indices (queries) are written in JavaScript, and it uses regular HTTP as its API.

PouchDB

PouchDB is an in-browser datastore inspired by CouchDB. It enables applications to store data locally while offline, then synchronize it with CouchDB.

hapi

hapi is a rich framework for building applications and services, enabling developers to focus on writing reusable application logic and not waste time with infrastructure logic. You can load hoodie as a hapi plugin to use it in your existing hapi application.

Users

Hoodie isn't a CMS, but a backend for web apps, and as such, it is very much centered around users. All of the offline and sync features are specific to each individual user's data, and each user's data is encapsulated from that of all others by default. This allows Hoodie to easily know what to sync between a user's clients and the server: simply all of the user's private data.

Private User Store

Every user signed up with your Hoodie app has their private little database. Anything you do in the **hoodie.store** methods stores data in here.