Feathers

A minimalist real-time framework for tomorrow's apps.

The official guide

Table of Contents

FeathersJS

Feathers

> A minimalist real-time framework for tomorrow's apps.

With Feathers it's easy to create scalable real-time applications. This guide will tell you everything you need to know about creating web and mobile apps with Feathers.

We have an introductory tutorial to get you ramped up quickly and some helpful guides to do more advanced things. Beyond that, this guide can be used as a general reference.

Sounds good? Let's get started!

If you are looking for different formats we have the latest version of this doc available in:

- HTML
- PDF
- ePub
- Mobi

What is Feathers?

Feathers is a minimalist, service-oriented, real-time web framework for modern applications that puts real-time communication at the forefront rather than as an afterthought. What do we mean by that?

Minimalist

Built on top of Express, Feathers has embodied the same spirit. It is comprised of a bunch of small modules that are all completely optional and the core weighs in at just a few hundred lines of code. How's that for light weight?! Now you can see where Feathers got it's name. ☺

Service oriented

Services are the heart of every Feathers application. They are small, data-oriented objects that can be used to perform CRUD operations on a resource. A resource could be stored in a database, on another server or somewhere entirely different.

Modern Applications

There is a lot to think about when building a modern application; speed, maintainability, flexibility, accessibility, scalability, the list goes on.

We've tried to ease that pain by wrapping industry best practices into a *"Batteries included but easily swappable"* package.

Out of the box Feathers provides a lot of what you need to build a modern web app or API. All of this is completely optional so you can pick and choose what you want to include and what you don't.

Real-time At The Core

Most real-time* web frameworks only allow clients to be pushed data in real time. You interact with your server over REST and then receive events over websockets or, even worse, the client polls for changes (which isn't really real-time).

Feathers is different. Feathers allows you to send **and** receive data over websockets, bringing real-time to the forefront and making your apps incredibly snappy.

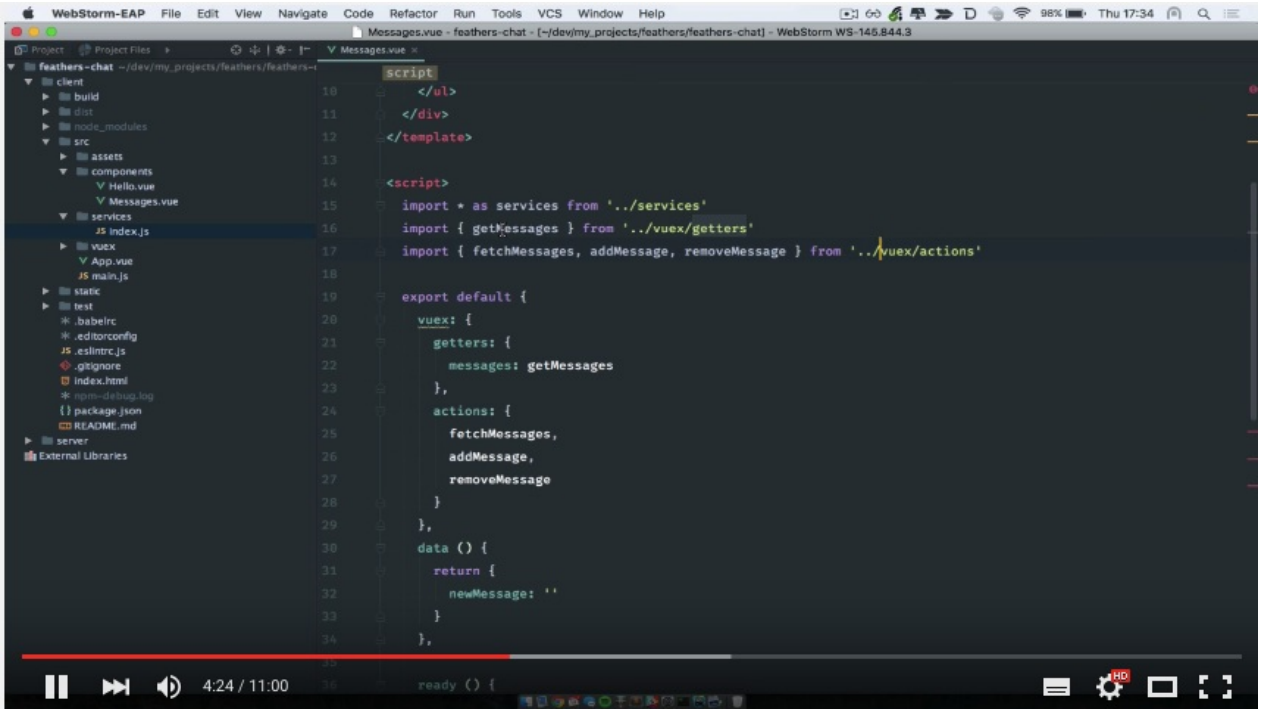
The whole Feathers ecosystem has been modeled around supporting real-time communication and making it a first class citizen instead of a hacky add-on. You can even forgo REST altogether and simply use websockets to communicate with your app, making it ideal for real-time IoT devices, among other things.

> *We're not talking about real-time computing in the traditional sense, we're talking about being able to push and pull data without making multiple HTTP requests. Like every other "real time" web framework Feathers does not guarantee a response time.

Ready to get started? Choose your own adventure:

- Quickly scaffold a real-time Feathers API
- Build your first Feathers app, or
- Dive a bit deeper into what Feathers is all about

Quick Start



> Mad ♥ to Chris Pena for putting together the video.

With NodeJS installed, you can quickly scaffold your first REST and real-time API by following these few steps.

> **ProTip:** Make sure you have set up your development environment properly. We recommend node v5.x+.

1. Install the Feathers CLI.

```
$ npm install -g feathers-cli
```

2. Create a directory for your new app.

```
$ mkdir feathers-chat
$ cd feathers-chat/
```

3. Generate the app and confirm all the prompts with the default by pressing enter:

```
$ feathers generate
```

4. Generate a new service. When asked `What do you want to call your service?`, type `messages` and confirm the other prompts with the default:

```
$ feathers generate service
```

5. Start your brand new app!

```
$ npm start
```

6. Go to localhost:3030 to see the homepage. The message CRUD service is available at localhost:3030/messages

7. Create a new message on the localhost:3030/messages endpoint. This can be done by sending a POST request with a REST client like Postman or via CURL like this:

```
$ curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --da
ta-binary '{ "text": "Hello Feathers!" }'
```

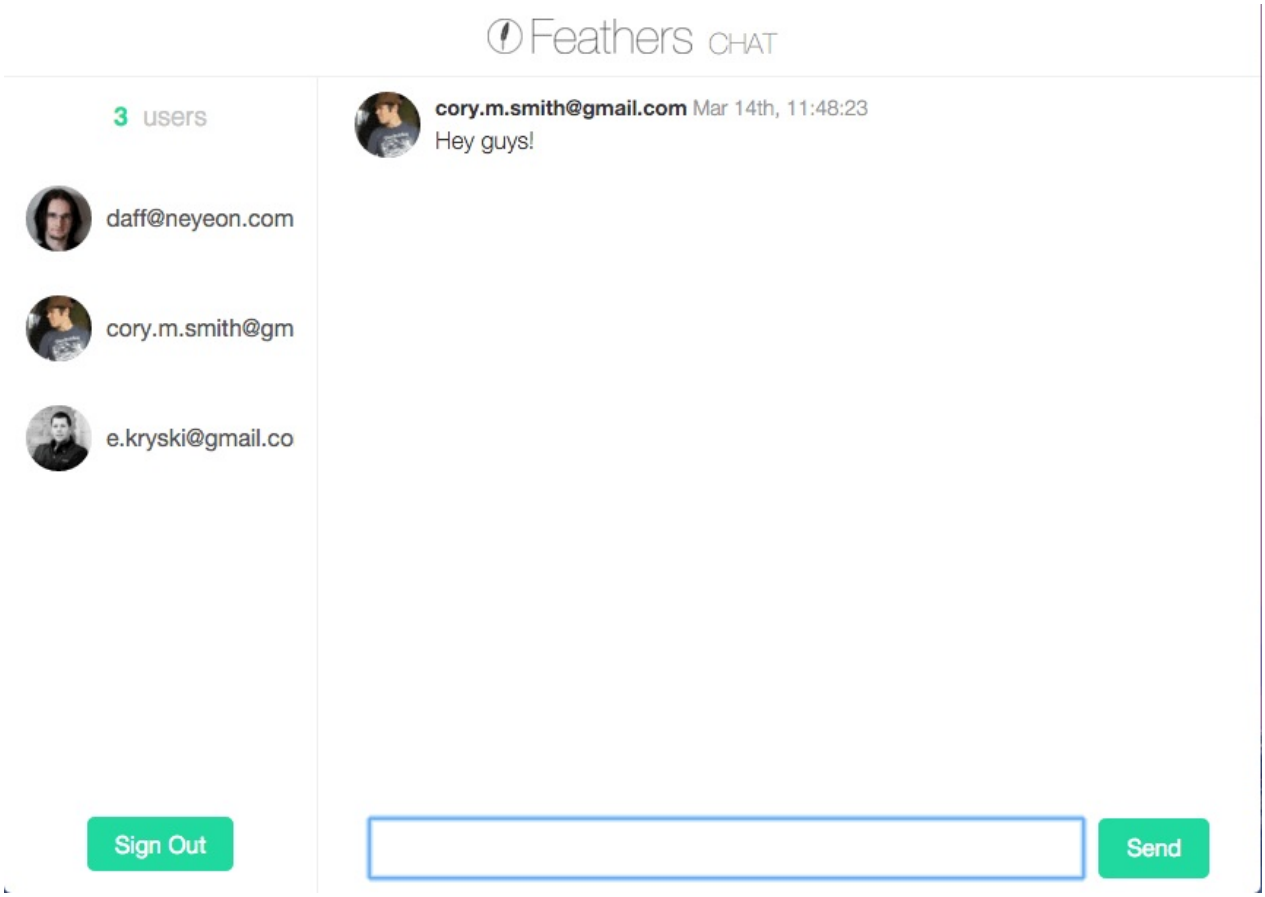
What's next?

In just a couple minutes we created a real-time API that is accessible via REST and websockets! We now have a database backed API that already provides CORS, authentication, pagination, logging, error handling and a few other goodies.

This is a great start! Let's take this a bit further and build our first real application.

> **ProTip:** If you chose a different database than the default NeDB you will have to start the database server and make sure the connection can be established.

Building Your First Feathers App



Well alright! Let's build our first Feathers app! We're going to build a real-time chat app with NeDB as the database. It's a great way to cover all the things that you'd need to do in a real world application and how Feathers can help. It also makes a lot more sense to have a real-time component than a Todo list. ☺

In this tutorial you go from nothing to a real-time chat app complete with signup, login, token based authentication, authorization, with a RESTful API and a working front-end. Let's get started.

You can find a complete working example here.

Setting up Your Environment

Feathers and most plug-ins work on Node `v0.12.x` and up but we recommend using the latest version or at least Node `v5.x`. All code samples in this documentation are written in ES2015 which is not supported in Node versions prior to `v5.0.0` without the use of a code transpiler like Babel (see below).

Node Version Manager

The easiest way to get the latest Node version and avoiding many problems especially around permissions is by using nvm or nvm-windows. Run the installation script and then, to see all available versions, run `nvm ls-remote`. To install e.g. Node 5.10 and set it as the default run:

```
nvm install 5.10
nvm alias default 5.10
```

Now `node --version` should show `5.10.x`. If it does then you can skip ahead to building the app and not have to worry about Babel.

Babel

If you can't use Node 5+ (although you really should) or would like to use even newer language features that are not part of Node yet you will need a JavaScript transpiler like Babel. The generator from the quick start guide and the tutorial does this automatically for you already if you are generating the application with an older Node version.

There are many ways to set up Babel. A quick and easy setup for development looks like this:

```
npm install babel-core babel-cli babel-preset-es2015 --save-dev
```

Create a `.babelrc` file like this:

```
{ "presets": ["es2015"] }
```

Update your `package.json` start and test scripts:

```
"scripts": {
  "start": "babel-node src/app",
  "test": "mocha test/ --recursive --compilers js:babel-register"
}
```

Now the application will run using ES2015 (including things like the `import` syntax which is not part of Node yet).

> **ProTip:** Babel is not included by default because we found that it slows down startup time and memory consumption during development considerably and Node 5 already supports most ES2015 features.

Scaffolding and Services

In this part we're going to scaffold a Feathers app, create our first service, send some data to the service and see it in real time! If you've already gone through the Quick Start section you can skip ahead to What Just Happened?.

Generate The App

Make sure you have NodeJS and npm installed and available on the command line as `node` and `npm`. Then we can install the Feathers CLI.

```
$ npm install -g feathers-cli
```

Create a directory for our new app:

```
$ mkdir feathers-chat
$ cd feathers-chat/
```

Generate your app and follow the prompts.

```
$ feathers generate
```

When presented with the project name just hit enter.

Next, enter in a short description of your app.

You're now presented with the option to choose which providers you want to support. Since we're setting up a real-time REST API we'll go with the default REST and Socket.io options. So just hit enter. You can learn more about Feathers providers in the Providers chapter.

Next we'll be prompted to support CORS. This basically allows your client to connect to the server from wherever. You can whitelist specific domains but again, just hit enter for now.

Now let's choose our database. You can see that Feathers supports a bunch through the generator and even more outside the generator. Let's use the default NeDB. NeDB is a local file-system based database so we don't have to start a separate database server. You can find out more about our database adapters in the Databases chapter.

Since pretty much every app needs authentication we generate almost everything you need to get going. You can learn more about authentication in Feathers by reading the Authentication chapter. In our case we will just use the default local authentication.

Your options should all look like this and you should have seen that a whole bunch of files were created.

```
[? Project name feathers-chat
[? Description A real-time chat application
? What type of API are you making? REST, Realtime via Socket.io
? CORS configuration Enabled for all domains
? What database do you primarily want to use? NeDB
? What authentication providers would you like to support? local
```

Next, npm will do it's thing and install all our dependencies. This can take a minute or two based on your Internet connection speed.

Now our app is ready to go but let's add a Message service for our chat app:

```
$ feathers generate service
```

The name should be `message` and the other options can be accepted with their default value (we will add authorization in the next chapter):

```
[? What do you want to call your service? message
? What type of service do you need? database
? For which database? NeDB
[? Does your service require users to be authenticated? No
   create src/services/message/index.js
   create src/services/message/hooks/index.js
   create test/services/message/index.test.js
```

We can now start our app with:

```
$ npm start
```

Open up localhost:3030 in your browser and you will see your new feathers app running. The endpoint for the Message service you just created is localhost:3030/messages.

ProTip: If you notice any errors when running `npm start`, make sure you have the latest versions of Node and NPM, otherwise you might run into issues such as this.

What Just Happened

A lot of stuff just happened there very quickly. We automatically generated a basic application with a REST and real time API for both messages, users and local authentication. We now have full CRUD capability that uses a persistent datastore for both the `/messages` and `/users` endpoints. We also have local authentication fully configured and some basic authorization permissions on the user service.

If you look at your source code you'll see that the following folder structure was generated:

- `config` contains a `default.json` and `production.json` application configuration file for things like the database connection strings and other configuration options.
- `public` is the publicly hosted folder with the homepage
- `src` contains all the application source files
  - `hooks` will contain global hooks
  - `middleware` contains Express middleware
  - `services` has a folder for each service. A service has an `index.js` and a `hooks` folder for service specific hooks
  - `app.js` is the main application file which can be imported to test services
  - `index.js` imports `app.js` and starts the server on the ports set in the configuration file
- `test` contains test files for the app, services and hooks

Creating Our First Message

With the server running let's create our first message by sending a POST request to our message service at http://localhost:3030/messages. You can use any REST client like Postman or via CURL like this:

```
$ curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --data-bi
nary '{ "text": "Hello Feathers!" }'
```

> **ProTip:** If you are making requests from a browser or Postman you want to make sure you set the the `Accept` header to `application/json` otherwise you will get HTML errors back. For more information see the Error Handling section.

You can also connect to the real-time API using Socket.io. The easiest way to do so is using the Feathers client. You can learn more about using Feathers on the client in the Client chapter.

Add the following to `public/index.html` before the `</body>` tag:

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.js"></script>
<script src="socket.io/socket.io.js"></script>
<script type="text/javascript">
  // Establish a Socket.io connection to the local server
  var socket = io();
  // Create a client side Feathers application that uses the socket
  // for connecting to services
  var app = feathers();
  app.configure(feathers.socketio(socket));
  // Retrieve a connection to the /messages service on the server
  // This service will use websockets for all communication
  var messages = app.service('messages');

  // Listen for when a new message has been created
  messages.on('created', function(message) {
    console.log('Someone created a message', message);
  });

  // Create a new message on the service
  messages.create({ text: 'Hello from websocket!' });
</script>
```

Open the console, then go to localhost:3030 and you will see the new message. Those events also work for REST calls. Try running the command below and you will see a new message show up your browser's console.

```
$ curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --data-bi
nary '{ "text": "Hello again!" }'
```

This is basically how Feathers does real-time and you can learn more about it in the Real-Time chapter.

What's next?

In this part we generated our first Feathers app, created our first service, and created our first message. In the next chapter we will create a new user and restrict access to our Message service.

User Management

In the previous section we set up a Message service with a `/messages` endpoint. The app we generated also comes with a `/users` endpoint and local authentication. Now we're going learn how we can create users and authenticate them.

Creating and authenticating users

Although a new user can be created by POSTing to the `/users` endpoint, we're going to create a separate `/signup` endpoint. This will keep things explicit and also show how you can use regular Express middleware with Feathers.

In this example the fields required for creating a user are `email` and `password`. Feathers automatically hashes passwords using bcrypt.

Feathers Authentication uses JSON webtoken (JWT) for secure authentication between a client and server as opposed to cookies and sessions. After we create a user, we'll be able to login. All we'll need to do is POST the `email` and `password` to the `http://localhost:3030/auth/local` endpoint set up by Feathers authentication. The response will return the authenticated user and their token.

This token needs to be set in the `Authorization` header for any subsequent requests that require authentication. You can find more details in the authentication chapter.

When we create a front-end for our chat API this will all be done automatically for us when using the Feathers client by calling `app.authenticate()`.

Adding HTML pages

For our chat app we will create a static `signup.html` and `login.html` page that shows a form. The form in `signup.html` will POST to the `/signup` endpoint which we will create later and `login.html` will submit to `auth/local` which already exists.

Let's replace `public/index.html` with the following welcome page:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1, user-scalable=0"
>
    <title>Feathers Chat</title>
    <link rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/chat.css">
  </head>
  <body>
    <main class="home container">
      <div class="row">
        <div class="col-12 col-8-tablet push-2-tablet text-center">
          <img class="logo center-item"
            src="http://feathersjs.com/img/feathers-logo-wide.png"
            alt="Feathers Logo">
          <h3 class="title">Chat</h3>
        </div>
      </div>
      <div class="row">
        <div class="col-12 push-4-tablet col-4-tablet">
          <div class="row">
            <div class="col-12">
              <a href="login.html" class="button button-primary block login">
                Login
              </a>
            </div>
          </div>
          <div class="row">
            <div class="col-12">
              <a href="signup.html" class="button button-primary block signup">
                Signup
              </a>
            </div>
          </div>
        </div>
      </div>
    </main>
  </body>
</html>
```

`public/login.html` looks like this:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1, user-scalable=0"
>
    <title>Feathers Chat</title>
    <link rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/chat.css">
  </head>
  <body>
    <main class="login container">
      <div class="row">
        <div class="col-12 col-6-tablet push-3-tablet text-center">
          <h1 class="font-100">Welcome Back</h1>
        </div>
      </div>
      <div class="row">
        <div class="col-12 col-6-tablet push-3-tablet col-4-desktop push-4-desktop tex
t-center">
          <form class="form" method="post" action="/auth/local">
            <fieldset>
              <input class="block" type="email" name="email" placeholder="email">
            </fieldset>
            <fieldset>
              <input class="block" type="password" name="password" placeholder="passwo
rd">
            </fieldset>
            <button type="submit" class="button button-primary block login">
              Login
            </button>
          </form>
        </div>
      </div>
    </main>
  </body>
</html>
```

Finally, `public/signup.html` looks like this:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1, user-scalable=0"
>
    <title>Feathers Chat</title>

    <link rel="shortcut icon" href="img/favicon.png">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/chat.css">
  </head>
  <body>
    <main class="login container">
      <div class="row">
        <div class="col-12 col-6-tablet push-3-tablet text-center">
          <h1 class="font-100">Create an Account</h1>
        </div>
      </div>
      <div class="row">
        <div class="col-12 col-6-tablet push-3-tablet col-4-desktop push-4-desktop tex
t-center">
          <form class="form" method="post" action="/signup">
            <fieldset>
              <input class="block" type="email" name="email" placeholder="email">
            </fieldset>
            <fieldset>
              <input class="block" type="password" name="password" placeholder="passwo
rd">
            </fieldset>
            <button type="submit" class="button button-primary block signup">
              Signup
            </button>
          </form>
        </div>
      </div>
    </main>
  </body>
</html>
```

Signing up new users

Now we have a welcome, login and signup page and we can create a `signup` endpoint that creates a new user from the `signup.html` form submission and then redirects to `login.html`. Because Feathers works just like Express we can just create an Express middleware called `signup` that does that.

```
$ feathers generate middleware
```

```
[? What do you want to call your middleware? signup
   create src/middleware/signup.js
```

`src/middleware/signup.js` takes the `users` service and creates a new user from the submitted form data. Then redirects to `login.html`:

```
'use strict';

module.exports = function(app) {
  return function(req, res, next) {
    const body = req.body;

    // Get the user service and `create` a new user
    app.service('users').create({
      email: body.email,
      password: body.password
    })
    // Then redirect to the login page
    .then(user => res.redirect('/login.html'))
    // On errors, just call our error middleware
    .catch(next);
  };
};
```

Now we have to add the `/signup` POST route to `src/middleware/index.js`:

```
'use strict';

const signup = require('./signup');

const handler = require('feathers-errors/handler');
const notFound = require('./not-found-handler');
const logger = require('./logger');

module.exports = function() {
  // Add your custom middleware here. Remember, that
  // just like Express the order matters, so error
  // handling middleware should go last.
  const app = this;

  app.post('/signup', signup(app));
  app.use(notFound());
  app.use(logger(app));
  app.use(handler());
};
```

> **ProTip:** Just like Express, most middleware should be registered before the `notFound` middleware.

The last step is to change the standard success redirect to `/chat.html`. This page will contain the actual frontend for our chat application. Because there are so many different frameworks we won't create that page yet and instead implement it in the framework guides for your favourite framework after finishing this tutorial. We set up the redirect by adding `successRedirect` to the `auth` section in `config/default.json`:

```
{
  "host": "localhost",
  "port": 3030,
  "nedb": "../data/",
  "public": "../public/",
  "auth": {
    "token": {
      "secret": "<your secret>"
    },
    "local": {},
    "successRedirect": "/chat.html"
  }
}
```

After stopping (CTRL + C) and starting the server (`npm start`) again we can go to the signup page, sign up with an email and password which will redirect us to the login page. There we can log in with the information we used to sign up and will get redirected to `chat.html` which currently shows authentication success page with the JWT.

Authorization

Now that we can authenticate we want to restrict the Message service to only authenticated users. We could have done that already in the service generator by answering *"yes"* when asked if we need authentication but we can also easily add it manually by changing `src/services/message/hooks/index.js` to:

```
'use strict';

const globalHooks = require('../../../hooks');
const auth = require('feathers-authentication').hooks;

exports.before = {
  all: [
    auth.verifyToken(),
    auth.populateUser(),
    auth.restrictToAuthenticated()
  ],
  find: [],
  get: [],
  create: [],
  update: [],
  patch: [],
  remove: []
};

exports.after = {
  all: [],
  find: [],
  get: [],
  create: [],
  update: [],
  patch: [],
  remove: []
};
```

That's it for authentication! We now have a *home*, *login* and *signup* page. We can sign up as a new user and log in with their email and password. In the next section we will look at creating new messages and adding additional information to them using the authenticated user information.

Using Hooks and Manipulating Data

In the previous section we set up authentication, a signup and login page, and restricted the message service to only authenticated users. Now we can add additional information, like the user who sent it and the creation time to a new message.

Adding information with hooks

Adding information like the current user and the creation time can be done by creating our own `before` hook. Hooks are a powerful way to register composable middleware before and after a service method runs. You can learn more about it in the hooks chapter. To generate a new hook run:

```
$ feathers generate hook
```

Gravatar profile images

Our first hook will be a `before` hook for the `create` method on the `user` service called `gravatar`:

```
[? What do you want to call your hook? gravatar
? What type of hook do you need? before hook
[? What service is this hook for? user
? What method is this hook for? create
   create src/services/user/hooks/gravatar.js
   create test/services/user/hooks/gravatar.test.js
```

In this hook we will modify the data before sending it to the database and add the Gravatar image from a users email address when someone signs up. Change `src/services/user/hooks/gravatar.js` to:

```
'use strict';

// src/services/user/hooks/gravatar.js
//
// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/hooks/readme.html

// We need this to create the MD5 hash
const crypto = require('crypto');

// The Gravatar image service
const gravatarUrl = 'https://s.gravatar.com/avatar';
// The size query. Our chat needs 60px images
const query = `s=60`;

// Returns a full URL to a Gravatar image for a given email address
const gravatarImage = email => {
  // Gravatar uses MD5 hashes from an email address to get the image
  const hash = crypto.createHash('md5').update(email).digest('hex');

  return `${gravatarUrl}/${hash}?${query}`;
};

module.exports = function() {
  return function(hook) {
    // Assign the new data with the Gravatar image
    hook.data = Object.assign({}, hook.data, {
      avatar: gravatarImage(hook.data.email)
    });
  };
};
```

Processing messages

The next hook will also be a `before` hook for a `create` method but this time for the `message` service. We'll do two things:

1. Add the `_id` of the user that created the message as `userId`
2. Add a `createdAt` timestamp with the current time

Let's generate that hook!

```
[? What do you want to call your hook? process
? What type of hook do you need? before hook
[? What service is this hook for? message
? What method is this hook for? create
   create src/services/message/hooks/process.js
   create test/services/message/hooks/process.test.js
```

Now we can update `src/services/message/hooks/process.js` to:

```
'use strict';

// src/services/message/hooks/process.js
//
// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/hooks/readme.html

module.exports = function(options) {
  return function(hook) {
    // The authenticated user
    const user = hook.params.user;
    // The actual message text
    const text = hook.data.text
      // Messages can't be longer than 400 characters
      .substring(0, 400)
      // Do some basic HTML escaping
      .replace(/&/g,'&amp;').replace(/</g,'&lt;').replace(/>/g,'&gt;');

    // Override the original data
    hook.data = {
      text,
      // Set the user id
      userId: user._id,
      // Add the current time via `getTime`
      createdAt: new Date().getTime()
    };
  };
};
```

Serializing Messages

So far we've implemented a couple `before` hooks but we can also format our data using `after` hooks, which get called after a service method returns.

Let's say that we want to populate the sender of each message so that we can display them in our UI. Instead of creating one of your own hooks, this time we'll use some that come bundled with Feathers. Specifically, we'll use the `populate` and `remove` hooks that come bundled with `feathers-hooks`.

We need to make a small change to our `src/services/message/hooks/index.js` file so that it now looks like this:

```
'use strict';

const auth = require('feathers-authentication').hooks;
const hooks = require('feathers-hooks');

const process = require('./process');
const globalHooks = require('../../../hooks');
const populateSender = hooks.populate('sentBy', {
  service: 'users',
  field: 'userId'
});

exports.before = {
  all: [
    auth.verifyToken(),
    auth.populateUser(),
    auth.restrictToAuthenticated()
  ],
  find: [],
  get: [],
  create: [process()],
  update: [hooks.remove('sentBy')],
  patch: [hooks.remove('sentBy')],
  remove: []
};

exports.after = {
  all: [],
  find: [populateSender],
  get: [populateSender],
  create: [populateSender],
  update: [],
  patch: [],
  remove: []
};
```

This will take the ID stored at the `userId` attribute on our Message, query the `users` service to find a User with that ID, and set the User object on the `sentBy` attribute (replacing the ID).

As you can see, manipulating data is pretty easy with hooks. To improve portability, we could break our hooks up into multiple smaller hooks and chain them. A good candidate might be to move manipulating the `createdAt` attribute into it's own hook so that it can be shared across multiple services. For this tutorial, we will leave it as it is.

Message authorization

We've seen how Hooks can be used to manipulate data but they can also be used for permissions and validations. We need one last hook that makes sure that users can only `remove`, `update` and `patch` their own message (see the services chapter for more information about those methods).

We need a `restrict-to-sender` *before* hook for the `message` service that runs before those methods. Go ahead and use the generator to create it. Once you've generated the hook, change the file at `src/services/message/hooks/restrict-to-sender.js` to:

```
'use strict';

// src/services/message/hooks/restrict-to-sender.js
//
// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/hooks/readme.html

const errors = require('feathers-errors');

module.exports = function(options) {
  return function(hook) {
    const messageService = hook.app.service('messages');

    // First get the message that the user wants to access
    return messageService.get(hook.id, hook.params).then(message => {
      // Throw a not authenticated error if the message and user id don't match
      if (message.sentBy._id !== hook.params.user._id) {
        throw new errors.NotAuthenticated('Access not allowed');
      }

      // Otherwise just return the hook
      return hook;
    });
  };
};
```

That's it! We now have a fully-functional, real-time chat API complete with user signup, local authentication, and authorization. You've now had an introduction to services, hooks and middleware, which is almost everything there is to Feathers.

In the next chapter we will briefly talk about building a frontend for our Chat app before learning more about Feathers and diving into the specific framework guides.

Building a Front-end

So far in this chapter we created a REST and real-time API to send messages including authentication and user management. We also added a static login and signup HTML page. Now it is time to use that API in a frontend. Feathers works great with any client that can connect through HTTP(S) or websockets to a REST API. For more details on what commands to send and listen to, follow up in the client use chapter. Sometimes it is just a few lines of code to make an existing front-end turn real-time.

Feathers is also universally usable and provides a JavaScript client that works in the Browser, React Native and other NodeJS servers. This makes it very easy to integrate with most JavaScript libraries and client side frameworks. In the frameworks section we've put together a growing list of integration guides to build real-time front-ends for the chat API we created in this chapter:

- jQuery Feathers Chat
- React Feathers Chat
- Vue.js Feathers Chat
- Feathers and Angular 2

If you don't see your favourite framework, do not despair! We have more information on how to use a Feathers API on the client in the Client Use chapter and we're always open for suggestions and questions. If you think you can help write a guide, we'd love to hear from you!

Why Feathers

If you went through the tutorial you probably saw just how quick it is to get an application off the ground.

Using Feathers is quite simply the fastest way to build scalable, real-time web and mobile applications. You can literally build prototypes in minutes and production ready applications in days.

Feathers achieves this by being a thin wrapper over top of some amazing battle tested open source technologies and adding a few core pieces like Hooks and Services.

If you've decided that Feathers might be for you and you haven't tried the tutorial, feel free to dive right in and build your first app. If you're still unsure about what Feathers does and where it comes from you can learn more about the Feathers philosophy, see what Feathers offers, or check out how Feathers compares to others.

The Feathers Philosophy

We know! You're probably screaming *"Not another JavaScript framework!"*. We've also become frustrated with all the Rails clones and MVC frameworks that don't do anything different. Instead, a few years ago we started to explore a different approach to building web applications using services and cross cutting concerns while also being careful not to reinvent the wheel.

With this experimentation Feathers has grown into what it is today. Our core philosophy that guides Feathers is still the same as it was years ago:

> *"Monolithic apps tend to fall apart at scale, either because of performance or because there are too many people in the code. What if we could make it easy to build applications that can naturally become service oriented from day one, rather than having to start with a large application and painfully tease it apart?*
>
> *What if we could make a framework that grows with you and your business and makes it easy for you to transition to a series of microservices, or easily change databases without ripping our code apart?*
>
> *What if we could make real-time less intimidating rather than a hacky, complex after thought? What if we could remove the boilerplate needed for building REST APIs? Could we build a framework that provides enough structure to get going easily and add all the common pieces that modern apps need, but still keep everything flexible and optional?*
>
> *A framework itself should not be opinionated. It should be made up of small, reusable, optional components that do one thing well but are combined in an opinionated way. By keeping the components of your application small, flexible and optional you eliminate much of the engineering obstacles that prevent moving fast and scaling."*

We strongly believe that your UI, data and business logic are the core of any web or mobile application and your framework should take care of the rest so you can focus on the things that matter.

The services concept

Many web frameworks focus on things like rendering views, defining routes and handling HTTP requests and responses without providing a structure for implementing application logic separate from those secondary concerns. The result - even when using the MVC

pattern - are monolithic applications with messy controllers or fat models. Your actual application logic and how your data is accessed are all mixed up together.

Feathers brings 3 important concepts together that help to separate those concerns from how your application works and give you incredible flexibility while still keeping things DRY.

Services

A service layer which helps to decouple your application logic from how it is being accessed and represented. Besides also making things a lot easier to test - you just call your service methods instead of having to make fake HTTP requests - this is what allows Feathers to provide the same API through both HTTP REST and websockets. It can even be extended to use any other RPC protocol and you won't have to change any of your services.

Uniform Interfaces

Every Feathers service exposes a uniform interface modeled after REST. Where, just like one of the key constraints of REST, your action context is immediately apparent due to the naming convention. With REST you have the HTTP verbs (GET, POST, PUT, PATCH and DELETE). This translates naturally to a Feathers service object interface:

```
const myService = {
  // GET /path
  find(params, callback) {},
  // GET /path/<id>
  get(id, params, callback) {},
  // POST /path
  create(data, params, callback) {},
  // PUT /path/<id>
  update(id, data, params, callback) {},
  // PATCH /path/<id>
  patch(id, data, params, callback) {},
  // DELETE /path/<id>
  remove(id, params, callback) {}
}
```

This interface also makes it easier to "hook" into the execution of those methods and emit events when they return which can naturally be used to provide real-time functionality.

Hooks

Cross cutting concerns are an extremely powerful part of aspect oriented programming. They are a very good fit for web and mobile applications since the majority are primarily CRUD applications with lots of shared functionality. Keeping with the Unix philosophy we

believe that small modules that do one thing are better than large complex ones. That's why you can create `before` and `after` hooks and chain them together to create very complex processes while still maintaining modularity and flexibility.

Built on the Shoulders of Giants

Because we utilize some already proven modules, we spend less time re-inventing the wheel, are able to move incredibly fast, and have small well-tested, stable modules.

Here's how we use some of the tech under the hood:

- Feathers extends Express 4, the most popular web framework for NodeJS.
- Our CLI tool extends Vorpal, its generators are built with Yeoman.
- We wrap Socket.io or Primus as your websocket transport.
- Our service adapters typically wrap mature ORMs like mongoose, sequelize, knex, or waterline.
- npm for package management.
- passport for much of the feathers-authentication work.

Now that you know a bit about where Feathers comes from let's look at what Feathers provides.

Feathers Showcase

Below are some of the amazing things built with Feathers or for the Feathers ecosystem.

Applications

- Feathers Chat
- Feathers React Native Chat

Starter stacks, Examples and Tutorials

Submit yours by creating a pull request.

- A series of small, example apps
- Feathers + Apollo
- Feathers + React + Mobx
- Feathers + React + Webpack

Community Plug-ins

Submit yours by creating a pull request.

Authentication

- feathers-accounts - Token-Based User Account System for FeathersJS (configure)
- feathers-service-verify-reset - Adds user email verification and password reset capabilities to local feathers-authentication (service)

Communications

- feathers-batch - Batch multiple Feathers service calls into one (service)

Database

- amity-mongodb - Use various FeatherJS services to manage a MongoDB server with Amity.
- can-connect-feathers - Feathers client library for DoneJS and can-connect (feathers-

client)
- canjs-feathers - CanJS model implementation that connects to Feathers services through feathers-client. (feathers-client)
- feathers-blob - Feathers abstract blob store service (service)
- feathers-blueprints - Add some of the Sails.js blueprints functionality to Feathers. (configure)
- feathers-bookshelf - A bookshelf ORM service adapter. (service)
- feathers-connect - Feathers client library for DoneJS and can-connect (feathers-client)
- feathers-filemaker - Filemaker adapter for feathers.js
- feathers-linvodb - Create an LinvoDB Service for FeatherJS. (service)
- feathers-mongo-collections - MongoDB collections service for FeathersJS. (service)
- feathers-mongo-databases - Create a MongoDB database service for FeathersJS. (service)
- feathers-mongodb-revisions - This Feathers database adapter extends the basic MongoDB adapter, adding revision support. (service)
- feathers-mongoose-advanced - Create a flexible Mongoose Service for FeathersJS. (service)
- feathers-mongoose-service - Easily create a Mongoose Service for Featherjs. (service)
- feathers-nedb-dump - Middleware for Feathers.js - dumps and restores NeDB database for a given service (middleware)
- feathers-objection - A service adapter for Objection.js - A minimal SQL ORM built on top of Knex.
- feathers-orm-service - Easily create a Object Relational Mapping Service for Featherjs.
- feathers-rethinky - Thinky.js RethinkDB Adaptor for Feathers JS
- feathers-seeder - Straightforward data seeder for FeathersJS services.
- feathers-skypager - A skypager ORM service adapter (service)
- feathers-solr - Solr Adapter for Feathersjs

Documentation

- feathers-swagger - Add documentation to your Featherjs services and feed them to Swagger UI. (configure)

Multiple instances

- feathers-cluster - Easily take advantage of multi-core systems for Featherjs. (configure)
- feathers-sync - Synchronize service events between application instances using MongoDB publish/subscribe (configure)

Email

- feathers-mailer - Feathers mailer service using nodemailer (service)
- feathers-mailgun - A Mailgun Service for FeatherJS. (service)
- feathers-sendgrid - A SendGrid Service for FeatherJS. (service)

React, Redux

- feathers-action - use feathers services with redux (connector)
- feathers-action-creators - redux action creators for feathers services
- feathers-action-reducer - redux reducer for feathers service actions
- feathers-action-types - flux action types for feathers services (connector)
- feathers-react-redux - Unofficial Feathers bindings for React-Redux.
- feathers-reduxify-services - Wrap Feathers services so they work transparently and perfectly with Redux.

Testing

- feathers-tests-fake-app-users - Fake some feathers dependencies in service unit tests. Starter for your customized fakes (service)

Transformation

- feathers-populate-hook - Feathers hook to populate multiple fields with n:m, n:1 or 1:m relations. (hook)
- feathers-transform-hook - Feathers hook for transform hook.data parameters (hook)
- feathers-virtual-attribute-hook - Feathers hook for add virtual attributes to your service response (hook)

Utilities

- feathers-hooks-common - Useful hooks for use with Feathersjs services. (hooks)
- feathers-hooks-utils - Utility library for writing Feathersjs hooks. (hooks)

Validation

- feathers-hooks-validate-joi - Feathers hook utility for schema validation, sanitization and client notification using Joi. (hook)
- feathers-hook-validation-jsonschema - Validate Feathers resources using JSON Schema. (hook)
- feathers-tcomb - validate feathers services using tcomb (app.service)
- feathers-validate-hook - Feathers hook for validate json-schema with is-my-json-valid (hook)

- feathers-validator - A validator for Feathers services. (service)

View layer

- donejs-feathers - A generator to quickly add FeathersJS to your DoneJS project. Includes Auth! (generator)
- feathers-done-ssr - a set of Express middleware that allows Feathers JWT tokens to work with DoneJS's built-in SSR.
- feathers-mithril - Connect feathers.js to mithril.js (connector)
- feathers-reactive - Turns a Feathers service call into an RxJS observables that automatically updates on real-time events. (configure)
- ng-feathers - Feathers client for AngularJS. FeatherJS for plain old AngularJS (1.X)
- vue-syncers-feathjers - Synchronises feathers services with vue objects, updated in real time (connector)

Features

Feathers provides a lot of the things that you need for building modern web and mobile applications. Here are some of the things that you get out of the box with Feathers. All of them are optional so you can choose exactly what you need. No more, no less.

We like to think of Feathers as a *"batteries included but easily swappable"* framework.

Instant REST APIs	Feathers automatically provides REST APIs for all your services. This industry best practice makes it easy for mobile applications, a web front-end and other developers to communicate with your application.
Unparalleled Database Support	With Feathers service adapters you can connect to all of the most popular databases, and query them with a unified interface no matter which one you use. This makes it easy to swap databases and use entirely different DBs in the same app without changing your application code.
Real Time	Feathers services can notify clients when something has been created, updated or removed. To get even better performance, you can communicate with your services through websockets, by sending and receiving data directly.
Cross-Cutting Concerns	Using "hooks" you have an extremely flexible way to share common functionality or concerns. Keeping with the Unix philosophy, these hooks are small functions that do one thing and are easily tested but can be chained to create complex processes.
Universal Usage	Services and hooks are a powerful and flexible way to build full stack applications. In addition to the server, these constructs also work incredibly well on the client. That's why Feathers works the same in NodeJS, the browser and React Native.
Authentication	Almost every app needs authentication so Feathers comes with support for email/password, OAuth and Token (JWT) authentication out of the box.
API Versioning	As an application matures the API typically evolves to accommodate business needs or technology changes. With Feathers it's easy to version your API by mounting a sub application or spinning up an entirely new service or app.
Pagination	Today's applications are very data rich so most of the time you cannot load all the data for a resource all at once. Therefore, Feathers gives you pagination for every service from the start.
Rate Limiting	When an app goes to production you'll need to have some protection against denial of service attacks. With Feathers it's easy to add Express middleware to do rate limiting at a service level.
Error Handling	Feathers removes the pain of defining errors and handling them. Feathers services automatically return appropriate errors, including validation errors, and return them to the client in a easily consumable format.
Logging	Feathers comes with a very simplistic logger that has sane defaults for production. However, it is easily swappable to allow you to customize to your needs.

Feathers vs X

The following sections compare Feathers to other software choices that seem similar or may overlap with the use cases of Feathers.

Due to the bias of these comparisons being on the Feathers website, we attempt to only use facts. Below you can find a feature comparison table and in each section you can get more detailed comparisons.

If you find something invalid or out of date in the comparisons, please create an issue (or better yet, a pull request) and we'll address it as soon as possible.

Feature Comparison

> Due to the fact that ease of implementation is subjective and primarily related to a developer's skill-set and experience we only consider a feature supported if it is officially supported by the framework or platform, regardless of how easy it is to implement (aka. are there official plugins, guides or SDKs?).

Legend

: Officially supported with a guide or core module

: Not supported

: Community supported or left to developer

Feature	Feathers	Express	Meteor	Sails	Firebase
REST API					
Real Time From Server					
Real Time From Client			(DDP)		
Universal JavaScript			(sort of)		
React Native Support					
Client Agnostic					(SDKs)
Email/Password Auth					
Token Auth					
OAuth					
Self Hosted					
Hosting Support					
Pagination					
Databases	10+ databases. Multiple ORMs		MongoDB	10+ databases. 1 ORM	Unknown
Analytics					
Admin Dashboard					
Push Notifications					
Offline Mode					
Hot Code Push					

Feathers vs. Express + Socket.io

Express is a minimalist web application framework for NodeJS. It was originally inspired by Sinatra and currently does much of the heavy lifting behind Feathers; routing, content-negotiation, middleware support, etc. You can actually just replace Express with Feathers in any existing application and start adding new Feathers services and hooks. All the same middleware that works with Express works with Feathers.

Because Express is so minimalist you still have to write a lot of code yourself to get things like RESTful routes mapped to a database resource, validation, authentication, authorization, rate limiting, logging, hooking up websockets for real-time support, etc.

Feathers eliminates all of that common boilerplate. It provides Services that give you CRUD methods for the most common databases, instant REST APIs and real-time compatibility. It sets up all the real-time events for you and sends messages when CRUD actions are performed. There are also core plugins that provide things like authentication services and hooks that make things like authorization just a few lines of code.

In addition to the server, Feathers can also be used in the browser and React Native apps.

You can think of Express as an abstraction over top of core low level NodeJS functionality that makes it easier to build web applications. Feathers is another thin abstraction layer over top of Express that brings together engineering patterns from Aspect Oriented Programming and Service Oriented Architecture along with some of the most popular Express middleware to make building web and mobile apps even faster and easier.

Feathers vs. Meteor

Both Feathers and Meteor are open source real-time JavaScript platforms that provide front end and back end support. They both allow clients to send and receive messages over websockets. Feathers lets you choose which real-time transport(s) you want to use via socket.io or Primus, while Meteor relies on SockJS.

Feathers is community supported, whereas Meteor is venture backed and has raised $31.2 million to date.

Meteor only has official support for MongoDB but there are some community modules of various levels of quality that support other databases. Meteor has it's own package manager and package ecosystem. They have their own template engine called Blaze which is based off of Mustache along with their own build system, but also have guides for Angular and React.

Feathers has official support for many more databases and supports any front-end framework or view engine that you want. We have framework guides for integrating Feathers with many of the most popular.

Feathers uses the defacto JavaScript package manager npm. As a result you can utilize the hundreds of thousands of modules published to npm. Feathers lets you decide whether you want to use Gulp, Grunt, Browserify, Webpack or any other build tool.

Meteor has optimistic UI rendering and oplog tailing whereas currently Feathers leaves that up to the developer. However, we've found that being universal and utilizing websockets for both sending and receiving data alleviates the need for optimistic UI rendering and complex data diffing in most cases.

Both Meteor and Feathers provide support for email/password and OAuth authentication. Once authenticated Meteor uses sessions to maintain a logged in state, whereas Feathers keeps things stateless and uses JSON Web Tokens (JWT) to assess authentication state.

One big distinction is how Feathers and Meteor provide real-time across a cluster of apps. Feathers does it at the service layer or using another pub-sub service like Redis whereas Meteor relies on having access to and monitoring MongoDB operation logs as the central hub for real-time communication.

Feathers vs. Sails

From a feature standpoint Feathers and Sails are probably the closest. Both provide real-time REST API's, multiple DB support, and are client agnostic. Sails is bound to the server whereas Feathers can also be used in the browser and in React Native apps. Both frameworks use Express, with Feathers supporting the latest Express 4, while Sails supports Express 3.

Sails follows the MVC pattern while Feathers provides lightweight services to define your resources. Feathers uses hooks to define your business logic including validations, security policies and serialization in reusable, chainable modules, whereas with Sails these reside in more of a configuration file format.

Feathers supports multiple ORMs while Sails only supports their Waterline ORM.

Sails allows you to receive messages via websockets on the client but, unlike Feathers, does not directly support data being sent from the client to the server over websockets. Additionally, Sails uses Socket.io for it's websocket transport. Feathers also supports Socket.io but also many other socket implementations via Primus.

Even though the features are very similar, Feathers achieves this with much less code. Feathers also doesn't assume how you want to manage your assets or that you even have any (you might be making a JSON API). Instead of coming bundled with Grunt Feathers let's you use your build tool of choice.

Sails doesn't come with any built in authentication support. Instead they have guides on how to configure Passport. By contrast, Feathers supports an official authentication plugin that is a drop-in, minimal configuration, module that provides email/password, token, and OAuth authentication much more like Meteor. Using this you can authenticate using those providers over REST **and/or** sockets interchangeably.

Scaling a Sails app is as simple as deploying your large app multiple times behind a load balancer with some pub-sub mechanism like Redis. With Feathers you can do the same but you also have the option to mount sub-apps more like Express, spin up additional services in the same app, or split your services into small standalone microservice applications.

Feathers vs. Firebase

Firebase is a hosted platform for mobile or web applications. Just like Feathers, Firebase provides REST and real-time APIs but also includes CDN support. Feathers on the other hand leaves setting up a CDN and hosting your Feathers app up to the developer.

Firebase is a closed-source, paid hosted service starting at 5$/month with the next plan level starting at 49$/month. Feathers is open source and can run on any hosting platform like Heroku, Modulus or on your own servers like Amazon AWS, Microsoft Azure, Digital Ocean and your local machine. Because Firebase can't be run locally you typically need to pay for both a shared development environment on top of any production and testing environment.

Firebase has JavaScript and mobile clients and also provides framework specific bindings. Feathers currently focuses on universal usage in JavaScript environments and does not have any framework specific bindings. Mobile applications can use Feathers REST and websocket endpoints directly but at the moment there are no Feathers specific iOS and Android SDKs.

Firebase currently supports offline mode whereas that is currently left up to the developer with Feathers. We do however have a proposal for this feature.

Both Firebase and Feathers support email/password, token, and OAuth authentication. Firebase has not publicly disclosed the database technology they use to store your data behind their API but it seems to be an SQL variant. Feathers supports multiple databases, NoSQL and SQL alike.

For more technical details on the difference and how to potentially migrate an application you can read how to use Feathers as an open source alternative to Firebase.

Feathers vs. Parse

Like Firebase, Parse is a mobile backend as a service (MBaaS). It was acquired by Facebook in 2013 but recently (Jan. 2016) announced it was shutting down and it open sourced a Parse API compatible Express server.

Prior to being open sourced Parse was a hosted service and wasn't available locally. Now that it is open source, like Feathers, it can be hosted anywhere that NodeJS can be run. This is left up to the developer to manage.

Parse supports push notifications. Currently Feathers does not, although a plugin is proposed for this functionality. The Express Parse server expects to work with MongoDB whereas Feathers supports many more databases.

Parse is a server side technology so it is completely front end agnostic and has integration guides and SDK's for virtually every client platform. Feathers is also client agnostic but currently doesn't have the same number of guides and currently only has an official JavaScript SDK (Pull Requests welcome for new SDK's).

Parse is not a real-time framework. Although they provided a push notification service, user authentication, and analytics the only way to support real-time without polling (ie. websockets) is to use a third party service. Feathers, on the other hand supports real-time and user authentication but currently leaves analytics and push notifications up to the developer.

Regarding authentication, Parse supports email and password and facebook authentication whereas Feathers supports both those, token authentication and any other OAuth provider.

Services

Services are the heart of every Feathers application. They perform the application-level i/o, helping you get data into and out of your app.

A service is simply a JavaScript object that offers one or more of the available service methods:

- `find`
- `get`
- `create`
- `update`
- `patch`
- `remove`
- `setup`

Services can be used just like an Express middleware:

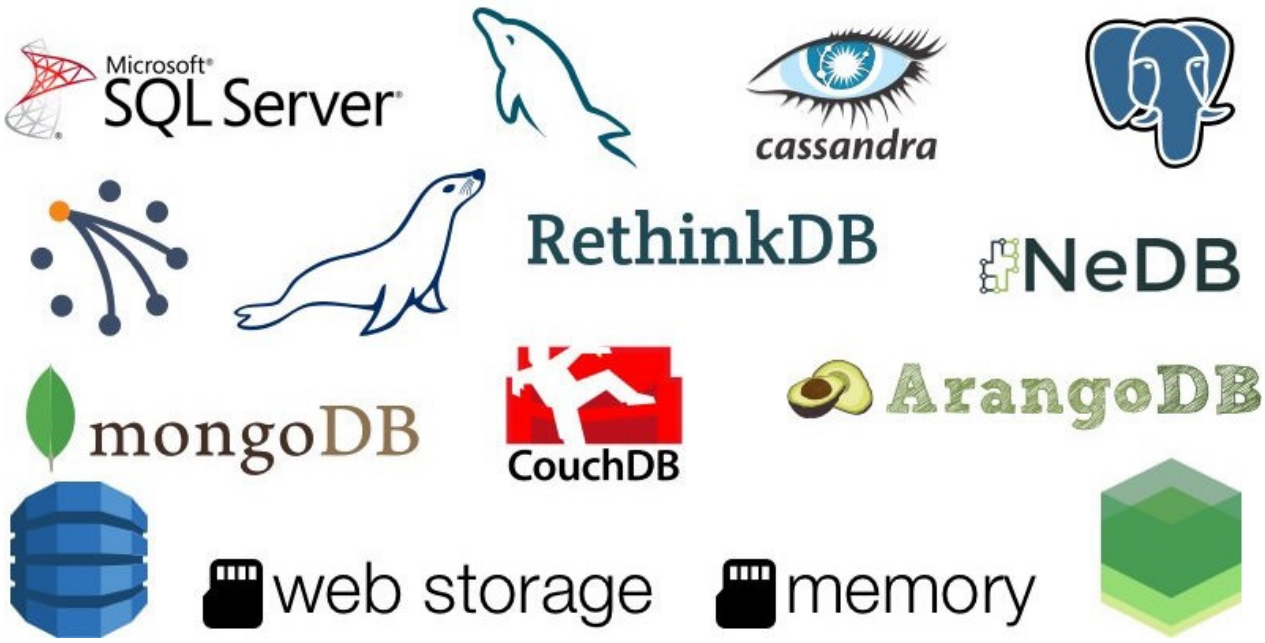
```
app.use('/path', serviceObject)
```

There are three common uses for services:

1. Facilitating Data Storage
2. Communicating with an External API
3. Real-time proxying a legacy server.

Services for Data Storage

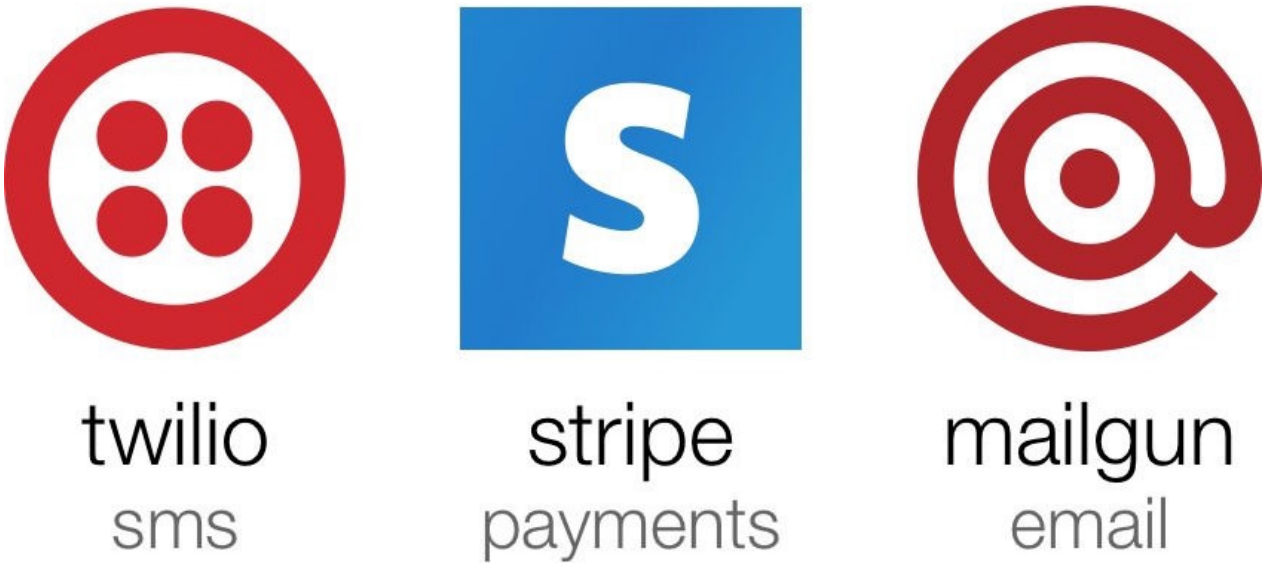
The most common use for a service is data storage. Thanks to its simple API and zealous community of developers, Feathers supports more data storage options than any other real-time framework. It also features a familiar query syntax for working with any of the database adapters. (See the section on Databases)



Services for External APIs

Services can also be used to communicate with other API providers. The community has published some great modules:

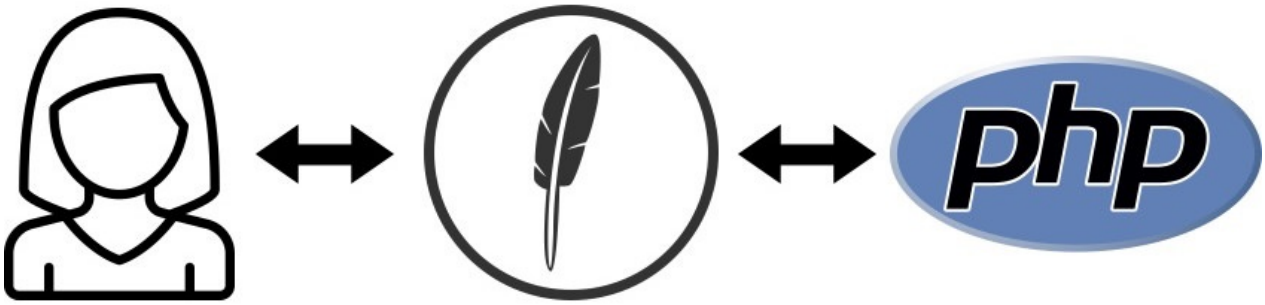
- feathers-twilio for handling calls and SMS messages over the Twilio API.
- feathers-stripe for processing financial transactions and managing customer data through the Stripe API.
- feathers-mailgun for sending transactional emails over the Mailgun API.



Services that Real-time Proxy a Legacy Server

Services provide an easy path for incrementally upgrading a legacy application to work with a modern API. This is usually done in three steps:

1. Logically map service methods to the existing legacy endpoints. This might involve using multiple services.
2. Set up each service method to make the request to the legacy API, acting as proxy for the user.
3. Update client-side applications to use the Feathers service's endpoints.



A Basic Example

A Feathers application with a very simple service and the REST provider set up can look like this:

```
// app.js
const feathers = require('feathers');
const rest = require('feathers-rest');
const app = feathers();

app.configure(rest());
app.use('/messages', {
  get(id, params) {
    return Promise.resolve({
      id,
      read: false,
      text: `Feathers is great!`,
      createdAt: new Date().getTime()
    });
  }
});

app.listen(3030);
```

After running

```
$ npm install feathers feathers-rest
$ node app.js
```

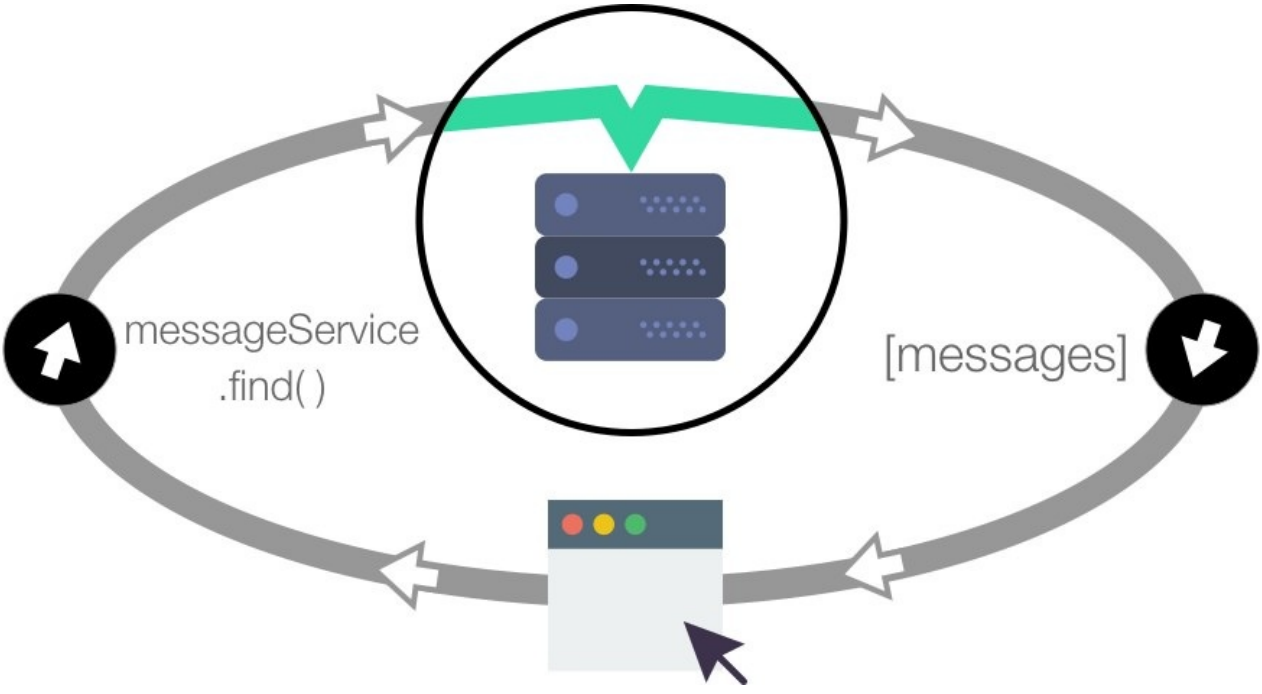
When going to localhost:3030/messages/1 you will see:

```
{
  "id": 1,
  "read": false,
  "text": "Feathers is great!",
  "createdAt": 1458490631911
}
```

Retrieving services

When registering a service with `app.use('/messages', messageService)` Feathers makes a shallow copy of that object and adds its own functionality. This means that to use Feathers functionality (like real-time events, hooks etc.) this object has to be used. It can be retrieved using `app.service` like this:

```
const messages = app.service('messages');
// also works with leading/trailing slashes
const messages = app.service('/messages/');

// Now we can use it on the server
messages.get(1).then(message => console.log(message.text));
```



Service methods

Below is a description of the complete interface for a Feathers service:

```
const myService = {
  find(params [, callback]) {},
  get(id, params [, callback]) {},
  create(data, params [, callback]) {},
  update(id, data, params [, callback]) {},
  patch(id, data, params [, callback]) {},
  remove(id, params [, callback]) {},
  setup(app, path) {}
}

app.use('/my-service', myService);
```

Or as an ES6 class:

```
'use strict';

class MyService {
  find(params [, callback]) {}
  get(id, params [, callback]) {}
  create(data, params [, callback]) {}
  update(id, data, params [, callback]) {}
  patch(id, data, params [, callback]) {}
  remove(id, params [, callback]) {}
  setup(app, path) {}
}

app.use('/my-service', new MyService());
```

> **ProTip:** Methods are optional, and if a method is not implemented Feathers will automatically emit a `NotImplemented` error.

Service methods should return a Promise and have the following parameters:

- `id` - the identifier for the resource. A resource is the data identified by a unique id.
- `data` - is the resource data.
- `params` - can contain any extra parameters, for example the authenticated user.
- `callback` - is an optional callback that can be called instead of returning a Promise. It is a Node-style callback function following the `function(error, result) {}` convention.

> **ProTip:** `params.query` contains the query parameters from the client (see the REST and real-time providers), `params.data` contains any data submitted in a request body, and `params.result` contains any data returned from a data store after a service method has been called.

These methods basically reflect a CRUD interface:

- `find(params [, callback])` - retrieves a list of all resources from the service. Provider parameters will be passed as `params.query`.
- `get(id, params [, callback])` - retrieves a single resource with the given `id` from the service.
- `create(data, params [, callback])` - creates a new resource with `data`. The method should return a Promise with the newly created data. `data` may also be an array which creates and returns a list of resources.
- `update(id, data, params [, callback])` - replaces the resource identified by `id` with `data`. The method should return a Promise with the complete updated resource data. `id` can also be `null` when updating multiple records.
- `patch(id, data, params [, callback])` - merges the existing data of the resource identified by `id` with the new `data`. `id` can also be `null` indicating that multiple resources should be patched. The method should return with the complete updated resource data. Implement `patch` additionally to `update` if you want to separate between partial and full updates and support the `PATCH` HTTP method.
- `remove(id, params [, callback])` - removes the resource with `id`. The method should return a Promise with the removed resource. `id` can also be `null` indicating to delete multiple resources.

methods mapped to restful actions

.find( )	GET	/todos
.get( )	GET	/todos/1
.create( )	POST	/todos
.update( )	PUT	/todos/1
.patch( )	PATCH	/todos/1
.delete( )	DELETE	/todos/1

The `setup` method

`setup(app, path)` is a special method that initializes the service, passing an instance of the Feathers application and the path it has been registered on. It is called automatically by Feathers when a service is registered.

`setup` is a great place to initialize your service with any special configuration or if connecting services that are very tightly coupled (see below), as opposed to using hooks.

```
// app.js
'use strict';

const feathers = require('feathers');
const rest = require('feathers-rest');

class MessageService {
  get(id, params) {
    return Promise.resolve({
      id,
      read: false,
      text: `Feathers is great!`,
      createdAt: new Date.getTime()
    });
  }
}

class MyService {
  setup(app) {
    this.app = app;
  }

  get(name, params) {
    const messages = this.app.service('messages');

    return messages.get(1)
      .then(message => {
        return { name, message };
      });
  }
}

const app = feathers()
  .configure(rest())
  .use('/messages', new MessageService())
  .use('/my-service', new MyService())

app.listen(3030);
```

You can see the combined response when going to localhost:3030/my-service/test.

Events

Any registered service will automatically turn into an event emitter that emits events when a resource has changed, that is a `create`, `update`, `patch` or `remove` service call returned successfully. For more information about events, please follow up in the real-time events chapter.

Protecting Service Methods

There are some times where you may want to use a service method inside your application or allow other servers in your cluster access to a method, but you don't want to expose a service method publicly. We've created a bundled hook that makes this really easy.

```
const hooks = require('feathers-hooks');

app.service('users').before({
  // Users can not be created by external access
  create: hooks.disable('external'),
});
```

Extending or Customizing Services

Services are really easy to create on their own but you can also customize an existing service by extending it in a few different ways. You can learn more by checking out the Extending Database Adapters section.

Providers

Providers are plugins that handle data transfer. They are the communication layer on top of which services operate. There are currently three providers available:

- feathers-rest - enables standard HTTP communication.
- feathers-socketio - enables real-time websocket communication using the Engine.io framework.
- feathers-primus - enables real-time websocket communication over eight possible frameworks.

Using providers together

The feathers-rest plugin does not include any real-time websocket functionality on its own. However, when you have one or more real-time providers enabled, all service events will be broadcast over them, by default. Read the section on real-time events to learn how they work.

Here are some examples of the default behavior that you could expect with various providers enabled:

- `feathers-rest` by itself - No real-time events will be available (they require websockets).
- `feathers-socketio` by itself - Real-time events will be sent to all connected clients.
- `feathers-rest` with `feathers-socketio` - Real-time events will be sent to all connected clients over the `feathers-socketio` provider, even if the original action was taken over the `feathers-rest` provider.
- `feathers-rest` with `feathers-socketio` and `feathers-primus` - Real-time events will be sent to all connected clients over both real-time providers.

Sending events across all connected providers by default has its advantages. Let's suppose that you have a situation where you want to use `feathers-socketio` to enable realtime for your main web application, but you decide that one of the other frameworks supported by `feathers-primus` works better in your React Native mobile application. Then you also decide that you want to enable Github and other services to communicate with your API using HTTP webhooks, so you enable the `feathers-rest` provider. With all providers

enabled, when a GitHub webhook sends a POST request to one of your services, that new data will propagate to the main web application across `feathers-socketio` and to the mobile app via `feathers-primus`.

> Pro Tip: The default behavior of sending events across all providers can be modified using event filters.

Creating your own provider

Because everything in the Feathers ecosystem is a plugin, you can create your own providers. For example, if you find that your application absolutely needs to use UDP, you could create a `feathers-udp` provider plugin modeled after the other providers. If you do create your own provider, please talk to us about adding mention of it here. Also, There is not currently a tutorial for creating an adapter, but pull requests are welcome! ☺

The REST API Provider

We have already seen in the services chapter that the feathers-rest module allows to expose services through a RESTful interface on the services path. This means that you can call a service method through the `GET`, `POST`, `PUT`, `PATCH` and `DELETE` HTTP methods:

```
const messageService = {
  // GET /messages
  find(params [, callback]) {},
  // GET /messages/<id>
  get(id, params [, callback]) {},
  // POST /messages
  create(data, params [, callback]) {},
  // PUT /messages[/<id>]
  update(id, data, params [, callback]) {},
  // PATCH /messages[/<id>]
  patch(id, data, params [, callback]) {},
  // DELETE /messages[/<id>]
  remove(id, params [, callback]) {},
  setup(app, path) {}
}

app.use('/messages', messageService);
```

A full overview of which HTTP method call belongs to which service method call and parameters can be found in the REST client use chapter. This chapter will talk about how to use and configure the provider module on the server.

Usage

Install the provider with:

```
$ npm install feathers-rest body-parser
```

We will have to provide our own body parser middleware (here the standard Express 4 body-parser) to make REST `.create`, `.update` and `.patch` calls parse the data in the HTTP body.

> **ProTip:** The body-parser middleware has to be registered *before* any service. Otherwise the service method will throw a `No data provided` or `First parameter for 'create' must be an object` error.

If you would like to add other middleware *before* the REST handler, simply call `app.use(middleware)` before registering any services. The following example creates a messages service that can save a new message and return all messages:

```
// app.js
'use strict';

const feathers = require('feathers');
const rest = require('feathers-rest');
const bodyParser = require('body-parser');

class MessageService {
  constructor() {
    this.messages = [];
  }

  find(params) {
    return Promise.resolve(this.messages);
  }

  create(data, params) {
    this.messages.push(data);

    return Promise.resolve(data);
  }
}

const app = feathers()
  // Enable the REST provider
  .configure(rest())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }));

app.use('/messages', new MessageService());

// Log newly created messages on the server
app.service('messages').on('created', message =>
  console.log('Created message', message)
);

app.listen(3030);
```

After starting the application with `node app.js`, we can now use CURL to create a new message:

```
curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --data-bina
ry '{ "text": "Learning Feathers!" }'
```

And should see the created message logged on the console. When going to localhost:3030/messages/ we see the newly created message.

Query, route and middleware parameters

URL query parameters will be parsed and passed to the service as `params.query`. For example:

```
GET /messages?read=true&$sort[createdAt]=-1
```

Will set `params.query` to

```
{
  "read": "true",
  "$sort": { "createdAt": "-1" }
}
```

> **ProTip:** Since the URL is just a string, there will be **no type conversion**. This can be done manually in a hook.

> **ProTip:** For REST calls, `params.provider` will be set to `rest` so you know which provider the service call came in on.

> **ProTip:** It is also possible to add information directly to the `params` object by registering an Express middleware that modifies the `req.feathers` property. It must be registered **before** your services are.

> **ProTip:** Route params will automatically be added to the `params` object.

```
const feathers = require('feathers');
const rest = require('feathers-rest');

const app = feathers();

app.configure(rest())
  .use(function(req, res, next) {
    req.feathers.fromMiddleware = 'Hello world';
    next();
  });

app.use('/users/:userId/messages', {
  get(id, params) {
    console.log(params.query); // -> ?query
    console.log(params.provider); // -> 'rest'
    console.log(params.fromMiddleware); // -> 'Hello world'
    console.log(params.userId); // will be `1` for GET /users/1/messages

    return Promise.resolve({
      id,
      params,
      read: false,
      text: `Feathers is great!`,
      createdAt: new Date.getTime()
    });
  }
});

app.listen(3030);
```

You can see all the passed parameters by going to something like `localhost:3030/users/213/messages/23?read=false&$sort[createdAt]=-1`. More information on how services play with Express middleware, routing and versioning can be found in the middleware chapter.

Customizing The Response Format

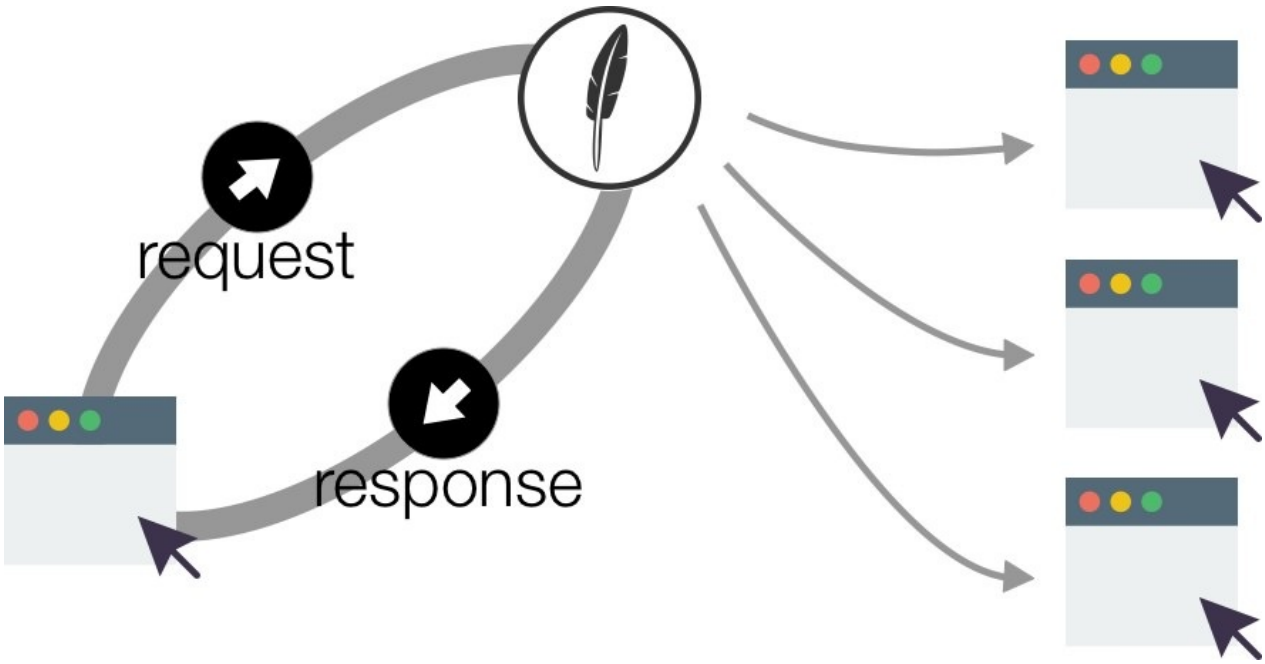
The default REST response formatter is a middleware that formats the data retrieved by the service as JSON. If you would like to configure your own `formatter` middleware just pass it to `rest(formatter)`. This middleware will have access to `res.data` which is the data returned by the service. res.format can be used for content negotiation. For example, a middleware that just renders plain text with the Message text:

```
const feathers = require('feathers');
const rest = require('feathers-rest');

const app = feathers();

function restFormatter(req, res) {
  res.format({
    'text/plain': function() {
      res.end(`The Message is: "${res.data.text}"`);
    }
  });
}

app.configure(rest(restFormatter))
  .use('/messages', {
    get(id, params) {
      return Promise.resolve({
        text: `Feathers is great!`
      });
    }
  });

app.listen(3030);
```

Now going to localhost:3030/messages/1 will print the plain text `The message is: "Feathers is great!"` .

realtime

In Feathers, realtime means that services automatically send `created`, `updated`, `patched` and `removed` events when a `create`, `update`, `patch` or `remove` service method is complete. Clients can listen for these events and then react accordingly.



With Feathers websockets aren't just used for sending events from server to client. It is also possible to call service methods and send data over sockets, either from server-to-server or client-to-server. This is often much faster than going through the REST API and results in a snappier app.

Currently Feathers supports two websocket transport libraries:

- Socket.io - Probably the most commonly used real-time library for NodeJS. It works on every platform, browser or device, focusing equally on reliability and speed.
- Primus - Is a universal wrapper for real-time frameworks that supports Engine.IO, WebSockets, Faye, BrowserChannel, SockJS and Socket.IO

In this chapter we will look at how to use Service events, how to configure the Socket.io and Primus real-time libraries and about how to restrict sending events to specific clients.

The Socket.io Provider

The feathers-socketio provider adds support for Socket.io which enables real-time bi-directional, event-based communication. It works on every platform, browser or device, focusing equally on reliability and speed.

Server Side Usage

Install the provider module with:

```
$ npm install feathers-socketio
```

Then import the module and pass it to `app.configure`. The following example will start a server on port 3030 and also set up Socket.io:

```
const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers().configure(socketio());

app.listen(3030);
```

Configuration

Once the server has been started with `app.listen()` the Socket.io object is available as `app.io`. It is also possible to pass a function that gets called with the initialized `io` server object (for more details see the Socket.io server documentation). This is a good place to listen to custom events or add authorization:

```
const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers()
  .configure(socketio(function(io) {
    io.on('connection', function(socket) {
      socket.emit('news', { text: 'A client connected!' });
      socket.on('my other event', function (data) {
        console.log(data);
      });
    });

    // Registering Socket.io middleware
    io.use(function (socket, next) {
      // Exposing a request property to services and hooks
      socket.feathers.referrer = socket.request.referrer;
      next();
    });
  }));

app.listen(3030);
```

It is also possible to additionally pass a Socket.io options object. This can be used to e.g. configure the path where Socket.io is initialize (`socket.io/` by default). The following changes the path to `ws/` :

```
const feathers = require('feathers');
const socketio = require('feathers-socketio');

const app = feathers()
  .configure(socketio({
    path: '/ws/'
  }, function(io) {
    // Do something here
    // This function is optional
  }));

app.listen(3030);
```

Middleware and service parameters

Similar to REST provider middleware, Socket.io middleware can modify the `feathers` property on the `socket` which will then be used as the service parameters:

```
app.configure(socketio(function(io) {
  io.use(function (socket, next) {
    socket.feathers.user = { name: 'David' };
    next();
  });
}));

app.use('messages', {
  create(data, params, callback) {
    // When called via SocketIO:
    params.provider // -> socketio
    params.user // -> { name: 'David' }
  }
});
```

Client Side Usage

A detailed description of the usage on a client can be found in Feathers Socket.io client chapter.

The Primus Provider

Primus is a universal wrapper for real-time frameworks that supports Engine.IO, WebSockets, Faye, BrowserChannel, SockJS and Socket.IO.

Server Side Usage

Install the provider module with:

```
$ npm install feathers-primus ws
```

> **ProTip:** Here we also installed the `ws` module which will let us use plain websockets. Typically you need to install your transport module in addition to primus. See the Primus docs for more details.

Now import the module and pass `primus(configuration [, fn])` to `app.configure`.

The following example will start a server on port 3030 and also set up Primus using the `ws` websocket module.

```
const feathers = require('feathers');
const primus = require('feathers-primus');

const app = feathers().configure(primus({
  transformer: 'websockets'
}));

app.listen(3030);
```

Configuration

The second parameter to the configuration function can be a callback that gets called with the Primus server instance that can be used to register Primus Middleware or Plugins

```
// Set up Primus with SockJS
app.configure(feathers.primus({
  transformer: 'sockjs'
}, function(primus) {
  // Do something with primus object
}));
```

Middleware and service parameters

Just like REST and SocketIO, the Primus request object has a `feathers` property that can be extended with additional service `params` during authorization:

```
app.configure(primus({
  transformer: 'sockjs'
}, function(primus) {
  // Do something with primus
  primus.before('todos::create', function(socket, done){
    // Exposing a request property to services and hooks
    socket.request.feathers.referrer = socket.request.referrer;
    done();
  });
}));

app.use('messages', {
  create(data, params, callback) {
    // When called via Primus:
    params.provider // -> primus
    params.user // -> { name: 'David' }
  }
});
```

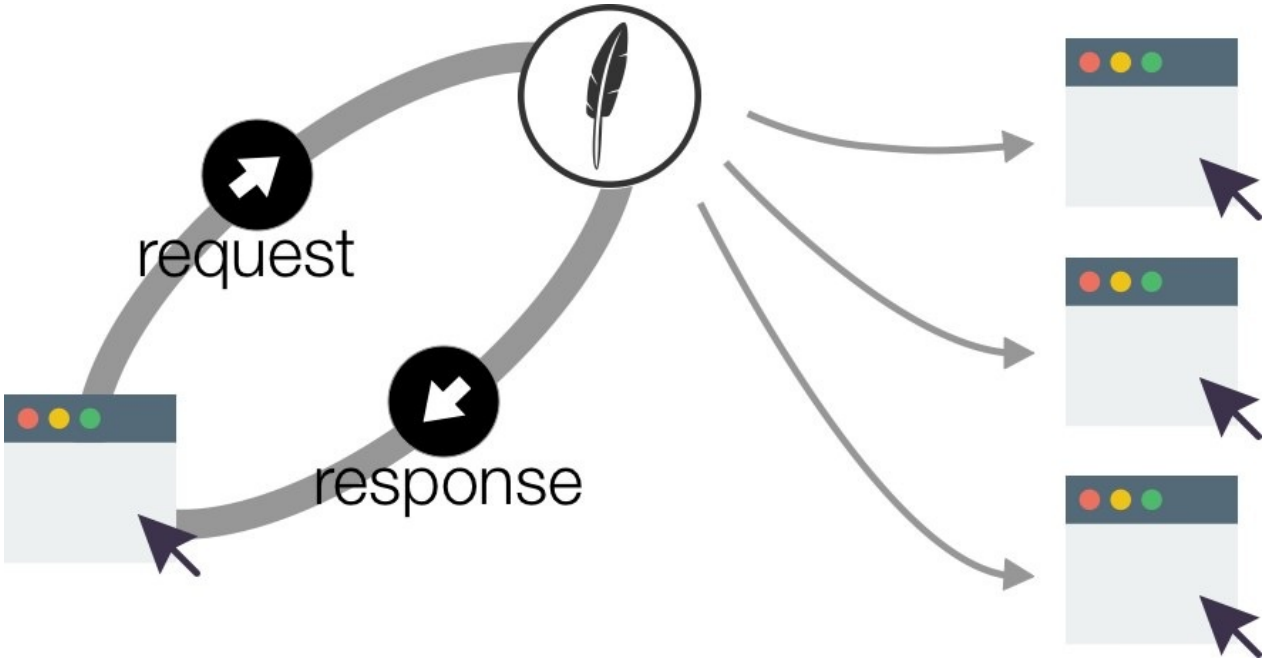
Client Side Usage

A detailed description of the usage on a client can be found in the Primus Feathers client chapter.

Service Events

Once registered through `app.use`, a Feathers service gets turned into an EventEmitter that sends `created`, `updated`, `patched` and `removed` events when the respective service method returns successfully. On the server and with the Feathers client you can listen to them by getting the service object with `app.service('<servicepath>')` and using it like a normal event emitter. Event behavior can also be customized or disabled using event filters.

> **ProTip:** Events are not fired until all of your *after* hooks have executed.



There are two types of events: Standard and Custom.

Standard Events

Standard events are built in to every service and are enabled by default. A standard event exists for each service method that affects data:

- `created`
- `updated`
- `patched`
- `removed`

created

The `created` event will fire with the result data when a service `create` returns successfully.

```
const feathers = require('feathers');
const app = feathers();

app.use('/messages', {
  create(data, params) {
    return Promise.resolve(data);
  }
});

// Retrieve the wrapped service object which will be an event emitter
const messages = app.service('messages');

messages.on('created', message => console.log('created', message));

messages.create({
  text: 'We have to do something!'
});
```

updated, patched

The `updated` and `patched` events will fire with the callback data when a service `update` or `patch` method calls back successfully.

```
const feathers = require('feathers');
const app = feathers();

app.use('/my/messages/', {
  update(id, data) {
    return Promise.resolve(data);
  },

  patch(id, data) {
    return Promise.resolve(data);
  }
});

const messages = app.service('my/messages');

messages.on('updated', message => console.log('updated', message));
messages.on('patched', message => console.log('patched', message));

messages.update(0, {
  text: 'updated message'
});

messages.patch(0, {
  text: 'patched message'
});
```

removed

The `removed` event will fire with the callback data when a service `remove` calls back successfully.

```
const feathers = require('feathers');
const app = feathers();

app.use('/messages', {
  remove(id, params) {
    return Promise.resolve({ id });
  }
});

const messages = app.service('messages');

messages.on('removed', messages => console.log('removed', messages));
messages.remove(1);
```

Custom events

By default, real-time clients will only receive the standard service events. However, it is possible to define a list of custom events on a service as `service.events` that should also be passed. For example, a payment service that sends status events to the client while processing a payment could look like this:

```
class PaymentService {
  constructor() {
    this.events = ['status'];
  },

  create(data, params) {
    createStripeCustomer(params.user).then(customer => {
      this.emit('status', { status: 'created' });
      return createPayment(data).then(result => {
        this.emit('status', { status: 'completed' });
      });
    });
    createPayment(data)
  }
}
```

Now clients can listen to the `<servicepath> status` event. Custom events can be filtered just like standard events.

Listening For Events

It is easy to listen for these events on the client or the server. Depending on the socket library you are using it is a bit different so refer to either the Socket.io or Primus docs.

Hooks vs Events

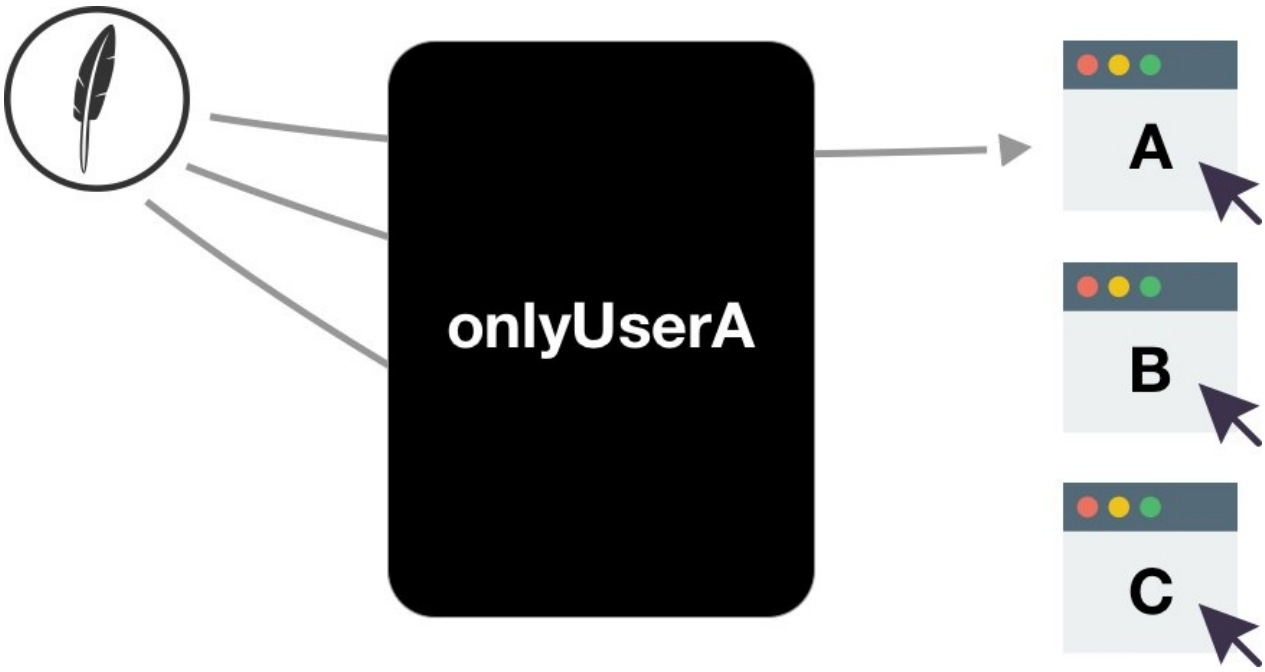
Binding to service events is great for logging or updating internal state. However, things like sending an email when creating a new user should be implemented through hooks.

The reason is that if you have multiple application instances of the same app, they will **all** be listening for, in this example, a `user created` event. All of their event handlers would be triggered and each app would send an email. Sending multiple emails to your user when they sign up (N X # of apps) is definitely not intended and not very scalable.

Furthermore, with hooks it is much easier to give feedback to the user when an error happened.

Event Filters

By default all service events will be sent to **all** connected clients. In many cases you probably want to be able to only send events to certain clients, say maybe only ones that are authenticated.



Both, the Socket.io and Primus provider add a `.filter()` service method which can be used to filter events. A filter is a `function(data, connection, hook)` which gets passed

- `data` - the data to dispatch.
- `connection` - the connected socket for which the data is being filtered. This is the `feathers` property from the Socket.io and Primus middleware and usually contains information like the connected user.
- `hook` - the hook object from the original method call.

It either returns the data to dispatch or `false` if the event should not be dispatched to this client. Returning a Promise that resolves accordingly is also supported so that you can chain filters, just like hooks.

> **ProTip:** Filter functions run for every connected client on every event and should be optimized for speed and chained by granularity. That means that general and quick filters should run first to narrow down the connected clients to then run more involved checks if necessary.

Registering filters

There are several ways filter functions can be registered, very similar to how hooks can be registered.

```
const todos = app.service('todos');

// Register a filter for all events
todos.filter(function(data, connection, hook) {});

// Register a filter for the `created` event
todos.filter('created', function(data, connection, hook) {});

// Register a filter for the `created` and `updated` event
todos.filter({
  created(data, connection, hook) {},
  updated(data, connection, hook) {}
});

// Register a filter chain the `created` and `removed` event
todos.filter({
  created: [ filterA, filterB ],
  removed: [ filterA, filterB ]
});
```

Filter examples

The following example filters all events on the `messages` service if the connection does not have an authenticated user:

```
const messages = app.service('messages');

messages.filter(function(data, connection) {
  if(!connection.user) {
    return false;
  }

  return data;
});
```

As mentioned, filters can be chained. So once the previous filter passes (the connection has an authenticated user) we can now filter all connections where the data and the user do not belong to the same company:

```
// Blanket filter out all connections that don't belong to the same company
messages.filter(function(data, connection) {
  if(data.company_id !== connection.user.company_id) {
    return false;
  }

  return data;
});
```

Now that we know the connection has an authenticated user and the data and the user belong to the same company, we can filter the `created` event to only be sent if the connections user and the user that created the Message are friends with each other:

```
// After that, filter messages, if the user that created it
// and the connected user aren't friends
messages.filter('created', function(data, connection, hook) {
  // The id of the user that created the todo
  const messageUserId = hook.params.user._id;
  // The a list of ids of the connection's user friends
  const currentUserFriends = connection.user.friends;

  if(currentUserFriends.indexOf(messageUserId) === -1) {
    return false;
  }

  return data;
});
```

Filtering Custom Events

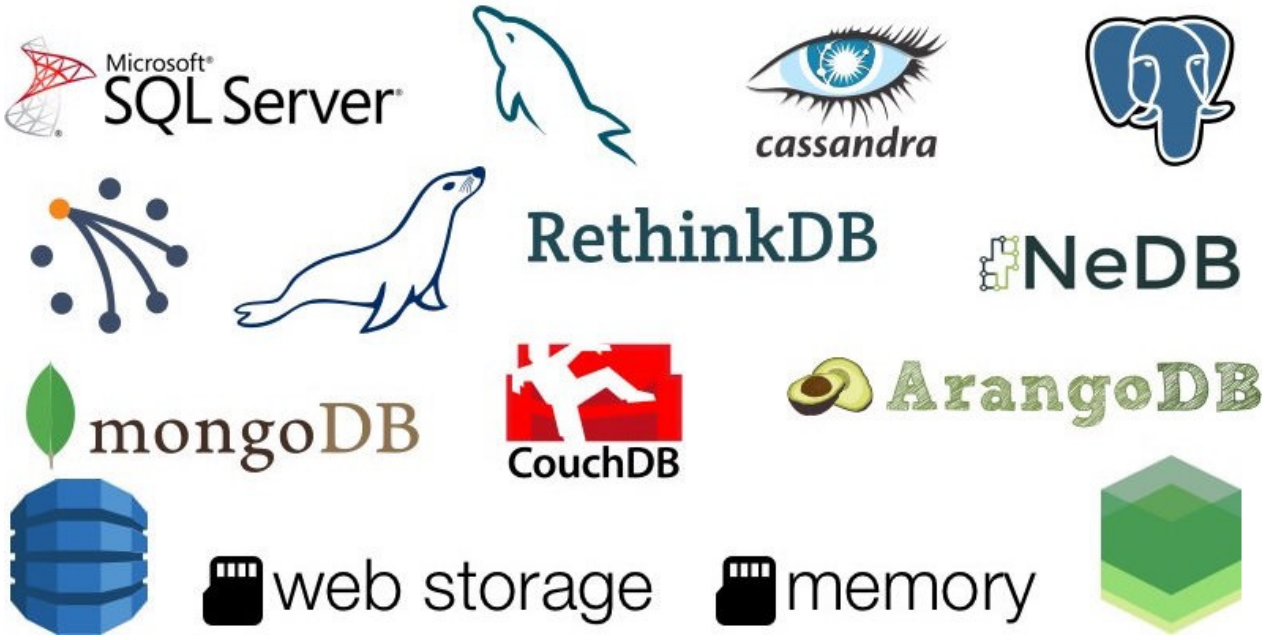
Custom events can be filtered the same way:

```
app.service('payments').filter('status', function(data, connection, hook) {

});
```

Databases

The service interface makes it easy to implement a wrapper that connects to a database. Which is exactly what we have done with the Feathers database adapters. Using those adapters it is possible to create a database-backed REST and real-time API endpoint, including validation in a few minutes!

Instead of coming up with our own ORM and validation system our official database adapters simply wrap many of the great ORM/ODM solutions that already exist. Feathers currently supports Mongoose, Sequelize, KnexJS, Waterline and LevelUP as well as standalone adapters for in-memory and NeDB. This allows you to use the following databases, among many others:



- **AsyncStorage** - feathers-localstorage
- **localStorage** - feathers-localstorage
- **Memory** - feathers-memory
- **MongoDB**
  - feathers-mongoose
  - feathers-mongodb
- **NeDB** - feathers-nedb
- **RethinkDB** - feathers-rethinkdb
- **PostgreSQL, MySQL, MariaDB, and SQLite**
  - feathers-knex
  - feathers-sequelize
- **Oracle** - feathers-knex

- **Microsoft SQL Server** - feathers-sequelize
- **Waterline** - feathers-waterline adds support for the following data stores (among others):
  - Redis
  - Riak
  - Neo4j
  - OrientDB
  - ArangoDB
  - Apache Cassandra
  - GraphQL
- **LevelUP** - feathers-levelup adds support for many backing stores including:
  - LevelDB
  - Amazon DynamoDB
  - Windows Azure Table Storage
  - Redis
  - Riak
  - Google Sheets

Every database adapter supports a common syntax for pagination, sorting and selecting and advanced querying out of the box and can be easily extended with custom functionality. Errors from the adapters (like ORM validation errors) will be passed seamlessly to clients.

This allows you to swap databases whenever the need arises **without having to change any of your querying code or validation hooks** and you can even use multiple databases within the same app!

Pagination and Sorting

All official database adapters support a common way for sorting, limiting and paginating `find` method calls as part of `params.query`.

Pagination

When initializing an adapter you can set the following pagination options in the `paginate` object:

- `default` - Sets the default number of items
- `max` - Sets the maximum allowed number of items per page (even if the `$limit` query parameter is set higher)

When `paginate.default` is set, `find` will return an object (instead of the normal array) in the following form:

```
{
  "total": "<total number of records>",
  "limit": "<max number of items per page>",
  "skip": "<number of skipped items (offset)>",
  "data": [/* data */]
}
```

The pagination options can be set as follows:

```
const service = require('feathers-<db-name>');

// Set the `paginate` option during initialization
app.use('/todos', service({
  paginate: {
    default: 5,
    max: 25
  }
}));

// override pagination in `params.paginate`
app.service('todos').find({
  paginate: {
    default: 100,
    max: 200
  }
});
// disable pagination for this call
app.service('todos').find({
  paginate: false
});
```

Sorting, limiting and selecting

The `find` API allows the use of `$limit`, `$skip`, `$sort`, and `$select` in the query. These special parameters can be passed directly inside the query object:

```
// Find all recipes that include salt, limit to 10, only include name field.
{"ingredients":"salt", "$limit":10, "$select": ["name"] } } // JSON

GET /?ingredients=salt&$limit=10&$select[]=name // HTTP
```

> **ProTip:** As a result of allowing these to be put directly into the query string, you won't want to use `$limit`, `$skip`, `$sort`, or `$select` as field names for documents in your database.

`$limit`

`$limit` will return only the number of results you specify:

```
// Retrieves the first two records found where age is 37.
query: {
  age: 37,
  $limit: 2
}
```

`$skip`

`$skip` will skip the specified number of results:

```
// Retrieves all except the first two records found where age is 37.
query: {
  age: 37,
  $skip: 2
}
```

`$sort`

`$sort` will sort based on the object you provide:

```
// Retrieves all where age is 37, sorted ascending alphabetically by name.
query: {
  age: 37,
  $sort: { name: 1 }
}

// Retrieves all where age is 37, sorted descending alphabetically by name.
query: {
  age: 37,
  $sort: { name: -1}
}
```

`$select`

`$select` support in a query allows you to pick which fields to include in the results.

```
// Only retrieve `name` and `id`
query: {
  name: 'Alice',
  $select: ['id', 'name']
}
```

To exclude fields from a result the remove hook can be used.

Querying

In addition to sorting and pagination, data can also be filtered by criteria. Standard criteria can just be added to the query. For example, the following finds all users with the name `Alice`:

```
query: {
  name: 'Alice'
}
```

Additionally, the following advanced criteria are supported for each property.

> **ProTip:** Just like with sorting and pagination you won't want to use these special attributes field names in your documents in your database.

$in, $nin

Find all records where the property does (`$in`) or does not (`$nin`) contain the given values. For example, the following query finds every user with the name of `Alice` or `Bob`:

```
query: {
  name: {
    $in: ['Alice', 'Bob']
  }
}
```

$lt, $lte

Find all records where the value is less (`$lt`) or less and equal (`$lte`) to a given value. The following query retrieves all users 25 or younger:

```
query: {
  age: {
    $lte: 25
  }
}
```

$gt, $gte

Find all records where the value is more (`$gt`) or more and equal (`$gte`) to a given value. The following query retrieves all users older than 25:

```
query: {
  age: {
    $gt: 25
  }
}
```

$ne

Find all records that do not contain the given property value, for example anybody not age 25:

```
query: {
  age: {
    $ne: 25
  }
}
```

$or

Find all records that match any of the given objects. For example, find all users name Bob or Alice:

```
query: {
  $or: [
    { name: 'Alice' },
    { name: 'Bob' }
  ]
}
```

Extending Database Adapters

Now that we talked about pagination, sorting and querying we can look at different ways you can extend the functionality of the existing database adapters.

> **ProTip:** Keep in mind that calling the original service methods will return a Promise that resolves with the value.

Hooks

The most flexible option is weaving in functionality through hooks. For a more detailed explanation about hook, go to the hooks chapter. For example, `createdAt` and `updatedAt` timestamps could be added like this:

```
const feathers = require('feathers');
const hooks = require('feathers-hooks');

// Import the database adapter of choice
const service = require('feathers-<adapter>');

const app = feathers()
  .configure(hooks())
  .use('/todos', service({
    paginate: {
      default: 2,
      max: 4
    }
  }));

app.service('todos').before({
  create(hook) {
    hook.data.createdAt = new Date();
  },

  update(hook) {
    hook.data.updatedAt = new Date();
  }
});

app.listen(3030);
```

Another important hook that you will probably use eventually is limiting the query for the current user (which is set in `params.user` by authentication):

```
app.service('todos').before({
  // You can create a single hook like this
  find: function(hook) {
    const query = hook.params.query;

    // Limit the entire query to the current user
    query.user_id = hook.params.user.id;
  }
});
```

Classes (ES6)

All modules also export an ES6 class as `Service` that can be directly extended like this:

```
'use strict';

const Service = require( 'feathers-<database>').Service;

class MyService extends Service {
  create(data, params) {
    data.created_at = new Date();

    return super.create(data, params);
  }

  update(id, data, params) {
    data.updated_at = new Date();

    return super.update(id, data, params);
  }
}

app.use('/todos', new MyService({
  paginate: {
    default: 2,
    max: 4
  }
}));
```

Uberproto (ES5)

You can also use `.extend` on a service instance (extension is provided by Uberproto):

```
const myService = memory({
  paginate: {
    default: 2,
    max: 4
  }
}).extend({
  create(data) {
    data.created_at = new Date();
    return this._super.apply(this, arguments);
  }
});

app.use('/todos', myService);
```

> **Note:** this is more for backwards compatibility. We recommend the usage of classes or hooks as they are easier to test, easier to maintain and are more flexible.

In Memory

feathers-memory is a service adapters that stores its data in-memory. It can be used for temporary data that doesn't need to be persisted and testing purposes. It also works great with client-side Feathers applications.

```
$ npm install --save feathers-memory
```

Getting Started

You can create an in-memory service with no options:

```
const memory = require('feathers-memory');

app.use('/messages', memory());
```

This will create a `messages` datastore with the default configuration.

Options

The following options can be passed when creating a new memory service:

- `idField` (default: 'id') [optional] - The name of the id field property.
- `startId` (default: 0) [optional] - An id number to start with that will be incremented for new record.
- `store` [optional] - An object with id to item assignments to pre-initialize the data store
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

Here is an example of a Feathers server with a `messages` in-memory service that supports pagination:

```
$ npm install feathers body-parser feathers-rest feathers-socketio feathers-memory
```

```
// app.js
const feathers = require('feathers');
const bodyParser = require('body-parser');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const memory = require('feathers-memory');

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable REST services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }));

// Create an in-memory Feathers service with a default page size of 2 items
// and a maximum size of 4
app.use('/messages', memory({
  paginate: {
    default: 2,
    max: 4
  }
}));

// Create a dummy Message
app.service('messages').create({
  text: 'Server message',
  complete: false
}).then(function(message) {
  console.log('Created message', message);
});

// Start the server.
const port = 3030;

app.listen(port, function() {
  console.log(`Feathers server listening on port ${port}`);
});
```

Run the example with `npm start` and go to localhost:3030/messages. You will see the test Message that we created at the end of that file.

KnexJS

feathers-knex is a database adapter for KnexJS, an SQL query builder for Postgres, MySQL, MariaDB, SQLite3, and Oracle designed to be flexible, portable, and fun to use.

```
npm install --save mysql knex feathers-knex
```

> **ProTip:** You also need to install the database driver for the DB you want to use. If you used the Feathers generator then this was already done for you.

Getting Started

You can create a SQL Knex service like this:

```
const knex = require('knex');
const service = require('feathers-knex');

const db = knex({
  client: 'sqlite3',
  connection: {
    filename: './db.sqlite'
  }
});

// Create the schema
db.schema.createTable('messages', table => {
  table.increments('id');
  table.string('text');
  table.boolean('read');
});

app.use('/messages', service({
  Model: db,
  name: 'messages'
}));
```

This will create a `messages` endpoint and connect to a local `messages` table on an SQLite database in `data.db`.

Options

The following options can be passed when creating a Knex service:

- `Model` (**required**) - The KnexJS database instance
- `name` (**required**) - The name of the table
- `id` (default: `id`) [optional] - The name of the id property.
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

Here's a complete example of a Feathers server with a `messages` SQLite service. We are using the Knex schema builder

```
$ npm install feathers feathers-rest feathers-socketio body-parser feathers-knex knex
sqlite3
```

```
// app.js
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const service = require('feathers-knex');
const knex = require('knex');

const db = knex({
  client: 'sqlite3',
  connection: {
    filename: './db.sqlite'
  }
});

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }));

// Create Knex Feathers service with a default page size of 2 items
// and a maximum size of 4
app.use('/messages', service({
  Model: db,
  name: 'messages',
```

```
  paginate: {
    default: 2,
    max: 4
  }
}));

// Clean up our data. This is optional and is here
// because of our integration tests
db.schema.dropTableIfExists('messages').then(function() {
  console.log('Dropped messages table');

  // Initialize your table
  return db.schema.createTable('messages', function(table) {
    console.log('Creating messages table');
    table.increments('id');
    table.string('text');
    table.boolean('complete');
  });
}).then(function() {
  // Create a dummy Message
  app.service('messages').create({
    text: 'Server message',
    complete: false
  }).then(function(message) {
    console.log('Created message', message);
  });
});

// Start the server.
const port = 3030;

app.listen(port, function() {
  console.log(`Feathers server listening on port ${port}`);
});
```

LocalStorage or AsyncStorage

feathers-localstorage is a service adapter that stores its data in localstorage in the browser or NodeJS (using a shim) or AsyncStorage in React Native. It is great for storing temporary data and can be used server side as well as in client-side Feathers applications.

```
$ npm install --save feathers-localstorage
```

Getting Started

You can easily create an localstorage service with no options:

```
const localstorage = require('feathers-localstorage');
app.use('/messages', localstorage({ storage: window.localStorage }));
```

This will create a `messages` datastore with the default configuration.

Options

The following options can be passed when creating a new localstorage service:

- `idField` (default: 'id') [optional] - The name of the id field property.
- `startId` (default: 0) [optional] - An id number to start with that will be incremented for new record.
- `name` (default: 'feathers') [optional] - The key to store data under in local or async storage.
- `storage` (**required**) - The local storage engine. You can pass in a server side localstorage module, the browser's `localStorage` or AsyncStorage on React Native.
- `store` [optional] - An object with id to item assignments to pre-initialize the data store
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Browser Usage

```
<script type="text/javascript" src="socket.io/socket.io.js"></script>
<script type="text/javascript" src="node_modules/feathers-client/dist/feathers.js"></s
cript>
<script type="text/javascript" src="node_modules/feathers-localstorage/dist/localstora
ge.js"></script>
<script type="text/javascript">
  var socket = io('http://api.my-feathers-server.com');
  var app = feathers()
    .configure(feathers.hooks())
    .configure(feathers.socketio(socket))
    .use('messages', feathers.localstorage({ storage: window.localStorage }));

  var localMessageService = app.service('messages');

  localMessageService.on('created', function(message) {
    console.log('Someone created a message', message);
  });

  localMessageService.create({
    text: 'Message from client'
  });
</script>
```

Server Usage

Here is an example of a Feathers server with a `messages` localstorage service that supports pagination:

```
$ npm install feathers body-parser feathers-rest feathers-socketio feathers-localstora
ge localstorage-memory
```

```
var feathers = require('feathers');
var bodyParser = require('body-parser');
var rest = require('feathers-rest');
var socketio = require('feathers-socketio');
var localstorage = require('feathers-localstorage');
var storage = require('localstorage-memory');

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }));

// Create an in-memory localstorage Feathers service with a default page size of 2 ite
ms and a maximum size of 4
app.use('/messages', localstorage({
  storage: storage,
  paginate: {
    default: 2,
    max: 4
  }
}));

// Start the server.
var port = 3030;

app.listen(port, function() {
  console.log(`Feathers server listening on port ${port}`);
});
```

Run the example with `npm start` and go to localhost:3030/messages. You will see the test Message that we created at the end of that file.

React Native Usage

```
$ npm install feathers feathers-socketio feathers-hooks feathers-localstorage socket.i
o-client
```

```
import React from 'react-native';
import localstorage from 'feathers-localstorage';
import feathers from 'feathers';
import hooks from 'feathers-hooks';
import localstorage from 'feathers-localstorage';
import {client as socketio} from 'feathers-socketio';
import {socket.io as io} from 'socket.io-client';

let { AsyncStorage } = React;

// A hack so that you can still debug. Required because react native debugger runs in
a web worker, which doesn't have a window.navigator attribute.
if (window.navigator && Object.keys(window.navigator).length == 0) {
  window = Object.assign(window, { navigator: { userAgent: 'ReactNative' }});
}

const socket = io('http://api.feathersjs.com', { transports: ['websocket'] });
const app = feathers()
  .configure(feathers.hooks())
  .configure(socketio(socket))
  .use('messages', localstorage({ storage: AsyncStorage }));

const localMessageService = app.service('messages');

localMessageService.on('created', function(message) {
  console.log('Someone created a message', message);
});

localMessageService.create({
  text: 'Message from client'
});
```

Mongoose

feathers-mongoose is a database adapter for Mongoose, an object modeling tool for MongoDB. Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more.

```
$ npm install --save mongoose feathers-mongoose
```

Getting Started

We can create Mongoose services like this:

```
const mongoose = require('mongoose');
const service = require('feathers-mongoose');

// A module that exports your Mongoose model
const Message = require('./models/message');

// Make Mongoose use the ES6 promise
mongoose.Promise = global.Promise;

// Connect to a local database called `feathers`
mongoose.connect('mongodb://localhost:27017/feathers');

app.use('/messages', service({ Model: Message }));
```

> **Important:** To avoid odd error handling behaviour, always set `mongoose.Promise = global.Promise`. If not available already, Feathers comes with a polyfill for native Promises.

See the Mongoose Guide for more information on defining your model.

Options

The following options can be passed when creating a new Mongoose service:

- `Model` (**required**) - The Mongoose model definition
- `id` (default: `_id`) [optional] - The name of the id property
- `paginate` - A pagination object containing a `default` and `max` page size (see the

Pagination chapter)

- `lean` (default: `false`) [optional] - When set to true runs queries faster by returning plain mongodb objects instead of mongoose models.
- `overwrite` (default: `true`) [optional] - Updates completely replace existing documents.

Complete Example

Here's a complete example of a Feathers server with a `messages` Mongoose service.

```
$ npm install feathers feathers-rest body-parser mongoose feathers-mongoose
```

In `message-model.js`:

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;
const MessageSchema = new Schema({
  text: {
    type: String,
    required: true
  },
  read: {
    type: Boolean,
    default: false
  }
});
const Model = mongoose.model('Message', MessageSchema);

module.exports = Model;
```

Then in `app.js`:

```
const feathers = require('feathers');
const rest = require('feathers-rest');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const service = require('feathers-mongoose');

const Model = require('./message-model');

// Tell mongoose to use native promises
// See http://mongoosejs.com/docs/promises.html
mongoose.Promise = global.Promise;

// Connect to your MongoDB instance(s)
mongoose.connect('mongodb://localhost:27017/feathers');

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}))

// Connect to the db, create and register a Feathers service.
app.use('/messages', service({
  Model,
  paginate: {
    default: 2,
    max: 4
  }
}));

// Create a dummy Message
app.service('messages').create({
  text: 'Server message'
}).then(function(message) {
  console.log('Created message', message);
});

// Start the server.
const port = 3030;
app.listen(port, function() {
    console.log(`Feathers server listening on port ${port}`);
});
```

You can run this example by using `npm start` and going to localhost:3030/messages. You should see a paginated object with the message that we created on the server.

Migrating

Version 3 of this adapter no longer brings its own Mongoose dependency, only accepts mongoose models and doesn't set up a database connection for you anymore. This means that you now need to make your own mongoose database connection and you need to pass in mongoose models changing something like

```
var MySchema = require('./models/mymodel')
var mongooseService = require('feathers-mongoose');
app.use('messages', mongooseService('message', MySchema, options));
```

To

```
var mongoose = require('mongoose');
var MongooseModel = require('./models/mymodel')
var mongooseService = require('feathers-mongoose');

mongoose.Promise = global.Promise;
mongoose.connect('mongodb://localhost:27017/feathers');

app.use('/messages', mongooseService({
  Model: MongooseModel
}));
```

Validation

Mongoose by default gives you the ability to add validations at the model level. Using an error handler like the one comes with Feathers your validation errors will be formatted nicely right out of the box!

For more complex validations you really have two options. You can combine Mongoose's validation mechanism with a validation library like validator.js or you can do your validations at the service level using hooks.

With Validator.js

Here's an example of doing more complex validations at the model level with the validator.js validation library.

```
const validator = require('validator');
const mongoose = require('mongoose');

const Schema = mongoose.Schema;
const userSchema = new Schema({
  email: {
    type: String,
    validate: {
      validator: validator.isEmail,
      message: '{VALUE} is not a valid email!'
    }
  },
  phone: {
    type: String,
    validate: {
      validator: function(v) {
        return /d{3}-d{3}-d{4}/.test(v);
      },
      message: '{VALUE} is not a valid phone number!'
    }
  }
});

const User = mongoose.model('user', userSchema);
```

Modifying results with the `toObject` hook

Unless you passed `lean: true` when initializing your service, the records returned from a query are Mongoose documents, so they can't be modified directly (You won't be able to delete properties from them).

To get around this, you can use the included `toObject` hook to convert the Mongoose documents into plain objects. Let's modify the after hook's setup in the feathers-hooks example, above, to this:

```
app.service('messages').after({
  all: [service.hooks.toObject({})]
});
```

The `toObject` hook must be called as a function and accepts a configuration object with any of the options supported by Mongoose's toObject method. Additionally, a second parameter can be specified that determines which attribute of `hook.result` will have its Mongoose documents converted to plain objects (defaults to `data`).

MongoDB Native

feathers-mongodb is a database adapter for MongoDB. Unlike Mongoose this is does not have an ORM. You deal with the database directly.

```
$ npm install --save mongodb feathers-mongodb
```

Getting Started

The following example will create a `messages` endpoint and connect to a local `messages` collection on the `feathers` database.

```
var MongoClient = require('mongodb').MongoClient;
var service = require('feathers-mongodb');
var app = feathers();

MongoClient.connect('mongodb://localhost:27017/feathers').then(function(db){
  app.use('/messages', service({
    Model: db.collection('messages')
  }));

  app.listen(3030);
});
```

Options

The following options can be passed when creating a new NeDB service:

- `Model` (**required**) - The MongoDB collection instance
- `id` (default: '_id') [optional] - The id field for your documents for this service.
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

To run the complete MongoDB example we need to install

```
$ npm install feathers feathers-rest feathers-socketio feathers-mongodb mongodb body-p
arser
```

Then add the following into `app.js`:

```
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const errors = require('feathers-errors');
const bodyParser = require('body-parser');
var MongoClient = require('mongodb').MongoClient;
const service = require('feathers-mongodb');

// Create a feathers instance.
const app = feathers()
  // Enable Socket.io
  .configure(socketio())
  // Enable REST services
  .configure(rest())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}));

  // Connect to your MongoDB instance(s)
MongoClient.connect('mongodb://localhost:27017/feathers').then(function(db){
  // Connect to the db, create and register a Feathers service.
  app.use('/messages', service({
    Model: db.collection('messages'),
    paginate: {
      default: 2,
      max: 4
    }
  }));

  // A basic error handler, just like Express
  app.use(errors.handler());

  // Start the server
  var server = app.listen(3030);
  server.on('listening', function() {
    console.log('Feathers Message MongoDB service running on 127.0.0.1:3030');
  });
}).catch(function(error){
  console.error(error);
});
```

You can run this example from the GitHub repository with `npm start` and going to localhost:3030/messages. You should see an empty array. That's because you don't have any messages yet but you now have full CRUD for your new messages service.

NeDB

feathers-nedb is a database adapter for NeDB, an embedded datastore with a MongoDB like API. By default NeDB persists data locally to a file. This is very useful if you do not want to run a separate database server. To use the adapter we have to install both, `feathers-nedb` and the `nedb` package itself:

```
$ npm install --save nedb feathers-nedb
```

Getting Started

The following example creates a NeDB `messages` service. It will create a `messages.db` datastore file in the `db-data` directory and automatically load it. If you delete that file, the data will be deleted. For a list of all the available options when creating an NeDB instance check out the NeDB documentation.

```
const NeDB = require('nedb');
const service = require('feathers-nedb');

// Create a NeDB instance
const db = new NeDB({
  filename: './data/messages.db',
  autoload: true
});

app.use('/messages', service({
  // Use it as the service `Model`
  Model: db,
  // Enable pagination
  paginate: {
    default: 2,
    max: 4
  }
}));
```

Options

The following options can be passed when creating a new NeDB service:

- `Model` (**required**) - The NeDB database instance

- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

To run the complete NeDB example we need to install

```
$ npm install feathers feathers-rest feathers-socketio feathers-nedb nedb body-parser
```

Then add the following into `app.js` :

```
const NeDB = require('nedb');
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const service = require('feathers-nedb');

const db = new NeDB({
  filename: './db-data/messages',
  autoload: true
});

// Create a feathers instance.
var app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}));

// Connect to the db, create and register a Feathers service.
app.use('/messages', service({
  Model: db,
  paginate: {
    default: 2,
    max: 4
  }
}));

// Create a dummy Message
app.service('messages').create({
  text: 'Oh hai!',
  complete: false
}).then(function(message) {
  console.log('Created message', message);
});

// Start the server.
const port = 3030;

app.listen(port, function() {
  console.log(`Feathers server listening on port ${port}`);
});
```

You can run this example from the GitHub repository with `npm start` and going to localhost:3030/messages. You should see an empty array. That's because you don't have any messages yet but you now have full CRUD for your new messages service.

Sequelize

feathers-sequelize is a database adapter for Sequelize, an ORM for Node.js. It supports the PostgreSQL, MySQL, MariaDB, SQLite and MSSQL dialects and features solid transaction support, relations, read replication and more.

```
npm install --save feathers-sequelize
```

And one of the following:

```
npm install --save pg pg-hstore
npm install --save mysql // For both mysql and mariadb dialects
npm install --save sqlite3
npm install --save tedious // MSSQL
```

> **ProTip:** If you are using the Feathers CLI generator this has already been done for you. For a full list of available drivers, check out Sequelize documentation.

Getting Started

`feathers-sequelize` hooks a Sequelize Model up as a service. For more information about models and general Sequelize usage, follow up in the Sequelize documentation.

```
const Model = require('./models/mymodel');
const sequelize = require('feathers-sequelize');

app.use('/messages', sequelize({ Model }));
```

Options

Creating a new Sequelize service currently offers the following options:

- `Model` (**required**) - The Sequelize model definition
- `id` (default: `id`) [optional] - The name of the id property
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

Here is an example of a Feathers server with a `messages` SQLite Sequelize Model:

```
$ npm install feathers feathers-rest feathers-socketio body-parser sequelize feathers-sequelize sqlite3
```

```
// app.js
const path = require('path');
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const Sequelize = require('sequelize');
const service = require('feathers-sequelize');

const sequelize = new Sequelize('sequelize', '', '', {
  dialect: 'sqlite',
  storage: path.join(__dirname, 'db.sqlite'),
  logging: false
});
const Message = sequelize.define('message', {
  text: {
    type: Sequelize.STRING,
    allowNull: false
  },
  read: {
    type: Sequelize.BOOLEAN,
    defaultValue: false
  }
}, {
  freezeTableName: true
});

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }));

// Removes all database content
Message.sync({ force: true });

// Create an in-memory Feathers service with a default page size of 2 items
// and a maximum size of 4
app.use('/messages', service({
  Model: Message,
  paginate: {
```

```
    default: 2,
    max: 4
  }
}));

// This clears the database
Message.sync({ force: true }).then(() => {
  // Create a dummy Message
  app.service('messages').create({
    text: 'Server message',
    read: false
  }).then(function(message) {
    console.log('Created message', message.toJSON());
  });
});

// Start the server
const port = 3030;

app.listen(port, function() {
  console.log(`Feathers server listening on port ${port}`);
});
```

Now there is an SQLite messages API running at `http://localhost:3030/messages`, including validation according to the model definition.

Validation

Sequelize by default gives you the ability to add validations at the model level. Using an error handler like the one that comes with Feathers your validation errors will be formatted nicely right out of the box!

Migrations

Migrations with feathers and sequelize are quite simple and we have provided some sample code to get you started. This guide will follow the directory structure used by the sample code, but you are free to rearrange things as you see fit. The following assumes you have a `migrations` folder in the root of your app.

Initial Setup: one-time tasks

1. Install the sequelize CLI:

```
npm install sequelize-cli --save
```

2. Create a `.sequelizerc` file in your project root with the following content:

```
const path = require('path');

module.exports = {
  'config': path.resolve('migrations/config/config.js'),
  'migrations-path': path.resolve('migrations'),
  'seeders-path': path.resolve('migrations/seeders'),
  'models-path': path.resolve('migrations/models')
};
```

3. Create the migrations config in `migrations/config/config.js`:

```js const app = require('../../src/app'); const env = process.env.NODE_ENV || 'development';

module.exports = {
    [env]: {
        url: app.get('db_url'),
        dialect: app.get('db_dialect'),
        migrationStorageTableName: '_migrations'
    }
};
```

1. Register your models. The following assumes you have defined your models using the method described here.

```
const Sequelize = require('sequelize');
const app = require('../../src/app');
const models = app.get('models');
const sequelize = app.get('sequelize');

// The export object must be a dictionary of model names -> models
// It must also include sequelize (instance) and Sequelize (constructor) properti
es
module.exports = Object.assign({
    Sequelize,
    sequelize
}, models);
```

Migrations workflow

The migration commands will load your application and it is therefore required that you define the same environment variables as when running you application. For example, many applications will define the database connection string in the startup command:

```
DATABASE_URL=postgres://user:pass@host:port/dbname npm start
```

All of the following commands assume that you have defined the same environment variables used by your application.

> **ProTip:** To save typing, you can export environment variables for your current bash/terminal session:
>
> ```
> export DATABASE_URL=postgres://user:pass@host:port/db
> ```

Create a new migration

To create a new migration file, run the following command and provide a meaningful name:

```
sequelize migration:create --name="meaningful-name"
```

This will create a new file in the migrations folder. All migration file names will be prefixed with a sortable data/time string: `20160421135254-meaninful-name.js`. This prefix is crucial for making sure your migrations are executed in the proper order.

> **NOTE:** The order of your migrations is determined by the alphabetical order of the migration scripts in the file system. The file names generated by the CLI tools will always ensure that the most recent migration comes last.

Add the up/down scripts:

Open the newly created migration file and write the code to both apply and undo the migration. Please refer to the sequelize migration functions for available operations. **Do not be lazy - write the down script too and test!** Here is an example of converting a `NOT NULL` column accept null values:

```
'use strict';

module.exports = {
  up: function (queryInterface, Sequelize) {
    return queryInterface.changeColumn('tableName', 'columnName', {
      type: Sequelize.STRING,
      allowNull: true
    });
  },

  down: function (queryInterface, Sequelize) {
    return queryInterface.changeColumn('tableName', 'columnName', {
      type: Sequelize.STRING,
      allowNull: false
    });
  }
};
```

> **ProTip:** As of this writing, if you use the `changeColumn` method you must **always** specify the `type`, even if the type is not changing.
>
> **ProTip:** Down scripts are typically easy to create and should be nearly identical to the up script except with inverted logic and inverse method calls.

Keeping your app code in sync with migrations

The application code should always be up to date with the migrations. This allows the app to be freshly installed with everything up-to-date without running the migration scripts. Your migrations should also never break a freshly installed app. This often times requires that you perform any necessary checks before executing a task. For example, if you update a model to include a new field, your migration should first check to make sure that new field does not exist:

```
'use strict';

module.exports = {
  up: function (queryInterface, Sequelize) {
    return queryInterface.describeTable('tableName').then(attributes => {
      if ( !attributes.columnName ) {
        return queryInterface.addColumn('tableName', 'columnName', {
          type: Sequelize.INTEGER,
          defaultValue: 0
        });
      }
    })
  },

  down: function (queryInterface, Sequelize) {
    return queryInterface.describeTable('tableName').then(attributes => {
      if ( attributes.columnName ) {
        return queryInterface.removeColumn('tableName', 'columnName');
      }
    });
  }
};
```

Apply a migration

The CLI tools will always run your migrations in the correct order and will keep track of which migrations have been applied and which have not. This data is stored in the database under the `_migrations` table. To ensure you are up to date, simply run the following:

```
sequelize db:migrate
```

> **ProTip:** You can add the migrations script to your application startup command to ensure that all migrations have run every time your app is started. Try updating your package.json `scripts` attribute and run `npm start`:
>
> ```
> scripts: {
>     start: "sequelize db:migrate && node src/"
> }
> ```

Undo the previous migration

To undo the last migration, run the following command:

```
sequelize db:migrate:undo
```

Continue running the command to undo each migration one at a time - the migrations will be undone in the proper order.

> **Note:** - You shouldn't really have to undo a migration unless you are the one developing a new migration and you want to test that it works. Applications rarely have to revert to a previous state, but when they do you will be glad you took the time to write and test your `down` scripts!

Reverting your app to a previous state

In the unfortunate case where you must revert your app to a previous state, it is important to take your time and plan your method of attack. Every application is different and there is no one-size-fits-all strategy for rewinding an application. However, most applications should be able to follow these steps (order is important):

1. Stop your application (kill the process)
2. Find the last stable version of your app
3. Count the number of migrations which have been added since that version
4. Undo your migrations one at a time until the db is in the correct state
5. Revert your code back to the previous state
6. Start your app

Waterline

feathers-waterline is a database adapter for the Waterline ORM, the ORM used by SailsJS. For detailed Waterline documentation, see the waterline-docs repository. Currently Waterline supports the following data stores:

- PostgreSQL - *0.9+ compatible*
- MySQL - *0.9+ compatible*
- MongoDB - *0.9+ compatible*
- Memory - *0.9+ compatible*
- Disk - *0.9+ compatible*
- Microsoft SQL Server
- Redis
- Riak
- IRC
- Twitter
- JSDom
- Neo4j
- OrientDB
- ArangoDB
- Apache Cassandra
- GraphQL
- Solr

Installation

```
npm install --save sails-postgresql waterline feathers-waterline
```

> **ProTip:** You also need to install the waterline database adapter for the DB you want to use.

Getting Started

`feathers-waterline` hooks a Waterline Model up to a configured data store as a feathers service.

```
const Message = require('./models/message');
const config = require('./config/waterline');
const Waterline = require('waterline');
const service = require('feathers-waterline');

const ORM = new Waterline();

ORM.loadCollection(Message);
ORM.initialize(config, function(error, data) {
    app.use('/messages', waterlineService({
      Model: data.collections.message
    }));
});
```

Options

Creating a new Waterline service currently offers the following options:

- `Model` (**required**) - The Waterline model definition
- `id` (default: `id`) [optional] - The name of the id property
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

Here is an example of a Feathers server with a `messages` Waterline Model using the Disk store:

```
$ npm install feathers feathers-rest feathers-socketio body-parser waterline sails-dis
k feathers-waterline
```

```
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const Waterline = require('waterline');
const diskAdapter = require('sails-disk');
const service = require('feathers-waterline');

const ORM = new Waterline();
const config = {
  adapters: {
    'default': diskAdapter,
    disk: diskAdapter
```

```
  },
  connections: {
    myLocalDisk: {
      adapter: 'disk'
    }
  },
  defaults: {
    migrate: 'alter'
  }
};
const Message = Waterline.Collection.extend({
  identity: 'message',
  schema: true,
  connection: 'myLocalDisk',
  attributes: {
    text: {
      type: 'string',
      required: true
    },

    complete: {
      type: 'boolean'
    }
  }
});

// Create a feathers instance.
const app = feathers()
  // Enable REST services
  .configure(rest())
  // Enable Socket.io services
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({ extended: true }));

ORM.loadCollection(Message);
ORM.initialize(config, (error, data) => {
  if (error) {
    console.error(error);
  }

  // Create a Waterline Feathers service with a default page size of 2 items
  // and a maximum size of 4
  app.use('/messages', service({
    Model: data.collections.message,
    paginate: {
      default: 2,
      max: 4
    }
  }));
```

```
  app.use(function(error, req, res, next){
    res.json(error);
  });

  // Create a dummy Message
  app.service('messages').create({
    text: 'Server message',
    complete: false
  }).then(function(message) {
    console.log('Created message', message.toJSON());
  });

  // Start the server
  const server = app.listen(3030);
  server.on('listening', function() {
    console.log('Feathers Message waterline service running on 127.0.0.1:3030');
    resolve(server);
  });
});
```

LevelUP

feathers-levelup is a database adapter for the fast and simple LevelDB, though many other backing stores are supported by supplying options to your LevelUP instances.

LevelUP backing store → LevelUP instance → Feathers service

LevelDB is a key-value database that stores its keys in lexicographical order, allowing for efficient range queries. Because LevelDB can only stream results in the order they are stored, avoid using `$sort` in your `find` calls, as the entire keyspace may be loaded for an in-memory sort. Read further to learn about ordering your LevelDB keys and designing your application around efficient range queries.

LevelUP currently supports the following backing stores:

- LevelDB
- Amazon DynamoDB
- AsyncStorage
- Basho's LevelDB Fork
- Google Sheets
- localStorage
- In-memory LRU Cache
- IndexedDB
- JSON Files
- Knex (sqlite3, postgres, mysql, websql)
- Medea
- Memory
- MongoDB
- MySQL
- Redis
- Riak
- RocksDB
- Windows Azure Table Storage

Installation

```
npm install levelup leveldown feathers-levelup --save
```

Getting Started

Creating a LevelUP service:

```
const levelup = require('levelup');
const levelupService = require('feathers-levelup');

// this will create a database on disk under ./todos
const db = levelup('./todos', { valueEncoding: 'json' });

app.use('/todos', levelupService({ db: db }));
```

See the LevelUP Guide for more information on configuring your database, including selecting a backing store.

Key Order

By default, LevelDB stores records on disk sorted by key. Storing sorted keys is one of the distinguishing features of LevelDB, and the LevelUP interface is designed around it.

When records are created, a key is generated based on a the value of `options.sortField`, plus a uuid. By default, `_createdAt` is automatically set on each record and its value is prepended to the key (id).

Change the `sortField` option to the field of your choice to configure your database's key ordering:

```
app.use('todos', service({
  db: db,
  sortField: '_createdAt' // this field value will be prepended to the db key
  paginate: {
    default: 2,
    max: 4
  }
}));

const todos = app.service('todos');

todos
  .create({task: 'Buy groceries'})
  .then(console.log);
```

```
{ task: 'Buy groceries',
  _createdAt: 1457923734510,
  id: '1457923734510:0:d06afc7e-f4cf-4381-a9f9-9013a6955562' }
```

Range Queries

Avoid memory-hungry `_find` calls that load the entire key set for processing by not specifying `$sort`, or by setting it to the same field as `options.sortField`. This way `_find` can take advantage of the natural sort order of the keys in the database to traverse the fewest rows.

Use `$gt`, `$gte`, `$lt`, `$lte` and `$limit` to perform fast range queries over your data.

```
app.use('todos', service({
  db: db,
  sortField: '_createdAt' // db keys are sorted by this field value
  paginate: {
    default: 2,
    max: 4
  }
}));

const todos = app.service('todos');

todos
  .find({
    query: {
      _createdAt: {
        $gt: '1457923734510'    // keys starting with this _createdAt
      },
      $limit: 10,               // load the first ten
      $sort: {
        _createdAt: 1           // sort by options.sortField (or don't pass $sort at a
ll)
      }
    }
  })
```

Complete Example

Here's a complete example of a Feathers server with a `message` levelup service.

```
const service = require('./lib');
const levelup = require('levelup');
const feathers = require('feathers');
const rest = require('feathers-rest');
const bodyParser = require('body-parser');
const socketio = require('feathers-socketio');

// Create a feathers instance.
const app = feathers()
  // Enable Socket.io
  .configure(socketio())
  // Enable REST services
  .configure(rest())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}));

// Connect to the db, create and register a Feathers service.
app.use('messages', service({
  db: levelup('./messages', { valueEncoding: 'json' }),
  paginate: {
    default: 2,
    max: 4
  }
}));

app.listen(3030);
console.log('Feathers Message levelup service running on 127.0.0.1:3030');
```

RethinkDB

feathers-rethinkdb is a database adapter for RethinkDB, the open-source database for the real-time web. To use the adapter we have to install both, `feathers-rethinkdb` and the `rethinkdbdash` package:

```
$ npm install --save rethinkdbdash feathers-rethinkdb
```

Getting Started

The following example creates a RethinkDB `messages` service. After connecting to the database server, it creates a database named `feathers` and a table named `messages`. For a list of all the available options for connecting to a RethinkDB server, check out the rethinkdbdash documentation.

```
const r = require('rethinkdbdash')({
  db: 'feathers'
});
const service = require('feathers-rethinkdb');

app.use('/messages', service({
  Model: r,
  name: 'messages',
  // Enable pagination
  paginate: {
    default: 2,
    max: 4
  }
}));
```

Options

The following options can be passed when creating a new RethinkDB service:

- `Model` (**required**) - The `rethinkdbdash` instance, already initialized with a configuration object. see options here
- `name` (**required**) - The name of the database table.
- `paginate` [optional] - A pagination object containing a `default` and `max` page size (see the Pagination chapter)

Complete Example

To run the complete RethinkDB example we need to install

```
$ npm install feathers feathers-rest feathers-socketio feathers-rethinkdb rethinkdbdas
h body-parser
```

We also need access to a RethinkDB server. You can install a local server on your local development machine by downloading one of the packages from the RethinkDB website. It might also be helpful to review their docs on starting a RethinkDB server.

Then add the following into `app.js`:

```
const rethink = require('rethinkdbdash');
const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const bodyParser = require('body-parser');
const service = require('../lib');

// Connect to a local RethinkDB server.
const r = rethink({
  db: 'feathers'
});

// Create a feathers instance.
var app = feathers()
  // Enable the REST provider for services.
  .configure(rest())
  // Enable the socketio provider for services.
  .configure(socketio())
  // Turn on JSON parser for REST services
  .use(bodyParser.json())
  // Turn on URL-encoded parser for REST services
  .use(bodyParser.urlencoded({extended: true}));

// Create your database if it doesn't exist.
r.dbList().contains('feathers')
  .do(dbExists => r.branch(dbExists, {created: 0}, r.dbCreate('feathers'))).run()

  // Create the table if it doesn't exist.
  .then(() => {
    return r.db('feathers').tableList().contains('messages')
      .do(tableExists => r.branch( tableExists, {created: 0}, r.dbCreate('messages')))
.run();
  })

  // Create and register a Feathers service.
  .then(() => {
```

```
    app.use('messages', service({
      Model: r,
      name: 'messages',
      paginate: {
        default: 10,
        max: 50
      }
    }));

    // Create a dummy Message
    app.service('messages').create({
      text: 'Oh hai!',
      complete: false
    }).then(message => console.log('Created message', message));
  })
  .catch(err => console.log(err));

const port = 3030;
app.listen(port, function() {
  console.log(`Feathers server listening on port ${port}`);
});
```

You can run this example from the GitHub repository with `npm start` and going to localhost:3030/messages. You should see an empty array. That's because you don't have any messages yet but you now have full CRUD for your new messages service.

Feathers Hooks

Hooks are just small middleware functions that get applied **before** and **after** a service method executes. The concept comes from Aspect Oriented Programming and they are construct for implementing Cross Cutting Concerns.

A hook is *provider independent*, which means it does not matter if it has been called through REST, Socket.io, Primus or any other provider Feathers may support in the future.

They are also service agnostic, meaning they can be used with **any** service regardless of whether they have a model or not. This keeps services lighter weight and only focused on the CRUD part of an application. An added benefit of not having things like validation baked into your services is flexibility.

Using hooks allows you to easily decouple the actual service logic from things like authorization, data pre-processing (sanitization and validation), data post processing (serialization), or sending notifications like emails or text messages after something happened.

That way you can swap databases or ORMs with minimal application code changes. You can also share validations for multiple databases in the same app, across multiple apps, and with your client. If hooks weren't completely independent of the service this would be extremely difficult to accomplish.

If you would like to learn more about the design patterns behind hooks read up on API service composition with hooks. In this chapter we will look at the usage of hooks, some examples and the built-in hooks.

Getting Started

To install from NPM run:

```
$ npm install feathers-hooks
```

Then, to use the plugin in a Feathers app:

```
const feathers = require('feathers');
const hooks = require('feathers-hooks');

const app = feathers().configure(hooks());
```

Now we can register a hook for a service:

```
const service = require('feathers-memory');

// Initialize our service
app.use('/users', service());

// Get our initialized service so that we can bind hooks
const userService = app.service('/users');

// Set up our before hook
userService.before({
  find(hook) {
    console.log('My custom hook ran');
  }
});
```

Using hooks

You can add as many `before` and `after` hooks as you like to any of the Feathers service methods:

- `get`
- `find`
- `create`
- `update`
- `patch`
- `remove`
- `all` (all service methods)

Hooks will be executed in the order they have been registered. There are two ways to use hooks. Either after registering the service by calling `service.before(beforeHooks)` or `service.after(afterHooks)` or by adding a `before` or `after` object with your hooks to the service.

Lets assume a Feathers application initialized like this:

```
const feathers = require('feathers');
const memory = require('feathers-memory');
const hooks = require('feathers-hooks');

const app = feathers()
  .configure(feathers.rest())
  .configure(hooks())
  .use('/todos', memory());

app.listen(8000);

// Get the wrapped service object which will be used in the other examples
const todoService = app.service('todos');
```

Before hooks

`service.before(beforeHooks)` hooks allow you to pre-process service call parameters. They will be called with the hook object and a callback which should be called with any errors or no arguments or `null` and the modified hook object. The hook object contains information about the intercepted method and for `before` hooks can have the following properties:

- `method` - The method name

- `type` - The hook type (`before` or `after`)
- `callback` - The original callback (can be replaced but shouldn't be called in your hook)
- `params` - The service method parameters
- `data` - The request data (for `create` , `update` and `patch`)
- `app` - The `app` object
- `id` - The id (for `get` , `remove` , `update` and `patch`)

All properties of the hook object can be modified and the modified data will be used for the actual service method call. This is very helpful for pre-processing parameters and massaging data when creating or updating.

`before` hooks can set `hook.result` which will skip the original service method. This can be used to override methods (see the soft-delete example).

The following example adds an authorization check (if a user has been provided in the params) to *all* service methods and also adds a `createdAt` property to a newly created todo:

```
todoService.before({
  all(hook) {
    if(!hook.params.user) {
      throw new Error('You need to be logged in');
    }
  },

  create(hook, next) {
    hook.data.createdAt = new Date();
  }
});
```

> **Note:** `all` hooks will be registered before specific hooks in that object. For the above example that means that the `all` hook will be added to the `create` service method and then the specific hook.

After hooks

`service.after(afterHooks)` hooks will be called with a similar hook object than `before` hooks but additionally contain a `result` property with the service call results:

- `method` - The method name
- `type` - The hook type (`before` or `after`)
- `result` - The service call result data
- `callback` - The original callback (can be replaced but shouldn't be called in your hook)
- `params` - The service method parameters

- `data` - The request data (for `create`, `update` and `patch`)
- `app` - The `app` object
- `id` - The id (for `get`, `remove`, `update` and `patch`)

In any `after` hook, only modifications to the `result` object will have any effect. This is a good place to filter or post-process the data retrieved by a service.

The following example filters the data returned by a `find` service call based on a users company id and checks if the current user is allowed to retrieve the data returned by `get` (that is, they have the same company id):

```
todoService.after({
  find(hook) {
    // Manually filter the find results
    hook.result = hook.result.filter(current =>
      current.companyId === params.user.companyId
    );
  },

  get(hook) {
    if (hook.result.companyId !== hook.params.user.companyId) {
      throw new Error('You are not authorized to access this information');
    }
  }
});
```

After hooks also support the `all` property to register a hook for every service method.

> **ProTip:** `all` hooks will be registered after specific hooks in that object.

> **ProTip:** The context for `this` in a hook function is the service it currently runs on.

As service properties

You can also add `before` and `after` hooks to your initial service object right away by setting the `before` and `after` properties to the hook object. The following example has the same effect as the previous examples:

```
const TodoService = {
  todos: [],

  get(id, params) {
    for (var i = 0; i < this.todos.length; i++) {
      if (this.todos[i].id === id) {
        return Promise.resolve(this.todos[i]);
      }
```

```
    }

    return Promise.reject(new Error('Todo not found'));
  },

  // Return all todos from this service
  find(params, callback) {
    return Promise.resolve(this.todos);
  },

  // Create a new Todo with the given data
  create(data, params, callback) {
    data.id = this.todos.length;
    this.todos.push(data);

    return Promise.resolve(data);
  },

  before: {
    find(hook) {
      if (!hook.params.user) {
        throw new Error('You are not logged in');
      }
    },

    create(hook) {
      hook.data.createdAt = new Date();
    }
  },

  after: {
    find(hook) {
      // Manually filter the find results
      hook.result = hook.result.filter(current =>
        current.companyId === hook.params.user.companyId
      );
    },

    get(hook) {
      if (hook.result.companyId !== hook.params.user.companyId) {
        throw new Error('You are not authorized to access this information');
      }
    }
  }
}
```

Asynchronous hooks

Hooks also allow asynchronous processing either by returning a Promise or by calling a callback.

Promises

All hooks can return a Promise object for asynchronous operations:

```
todoService.before({
  find(hook) {
    return new Promise((resolve, reject) => {

    });
  }
});
```

If you want to change the hook object just chain the returned promise using `.then`:

```
todoService.before({
  find(hook) {
    return this.get().then(data => {
      hook.params.message = 'Ran through promise hook';
      // Always return the hook object
      return hook;
    });
  }
});
```

> **ProTip:** If a promise fails, the error will be propagated immediately and will exit out of the promise chain.

Continuation passing

Another way is to pass `next` callback as the second parameter that has to be called with `(error, data)`.

```
todoService.before({
  find(hook, next) {
    this.find().then(data => {
      hook.params.message = 'Ran through promise hook';
      hook.data.result = data;
      // With no error
      next();
      // or to change the hook object
      next(null, hook);
    });
  }
});
```

Chaining / Registering Multiple Hooks

If you want to register more than one `before` or `after` hook for the same method, there are 2 ways to do this.

Dynamic Registrations

If you register a `before` or `after` hook for a certain method in one place and then register another `before` or `after` hook for the same method, `feathers-hooks` will automatically execute them in a chained fashion **in the order that they were registered**.

> **Pro Tip:** *This works well if you have more dynamic or conditional hooks.*

```
const app = feathers().use('/users', userService);

// We need to retrieve the wrapped service object from app which has the added hook fu
nctionality
const userService = app.service('users');

userService.before({
    ...
});

// Somewhere else
userService.before({
    ...
});
```

Defining Arrays

You can also register multiple hooks at the same time, in the order that you want them executed, when you are registering your service.

> **Pro Tip:** This is the preferred method because it is bit cleaner and execution order is more apparent. As your app gets bigger it is much easier to trace and debug program execution.

```
const hooks = require('your-hooks');
const app = feathers().use('/users', userService);

// We need to retrieve the wrapped service object from app which has the added hook fu
nctionality
const userService = app.service('users');

userService.before({
  // Auth is required.  No exceptions
  create : [hooks.requireAuth(), hooks.setUserID(), hooks.setCreatedAt()]
});
```

Communicating with other services

Hooks make it convenient to work with other services. You can use the `hook.app` object to lookup the services you need to use like this:

```
/**
 * Check if provided account already exists.
 */
const myHook = function(hook) {
  // Get a reference to the accounts service.
  const accounts = hook.app.service('accounts');

  // do something
}
```

Now that you know a bit about hooks work. Feel free to check out some examples or some of the bundled hooks that we've already written for you to for common use cases.

Customizing Built In Hooks

Sometimes you will only want to run a hook in certain circumstances or you wan to modify the functionality of the output of the hook without re-writing it. Since hooks are chainable you can simply wrap it in your own hook.

```
import { hooks } from 'feathers-authentication';

// Your custom hashPassword hook
exports.hashPassword = function(options) {
  // Add any custom options

  return function(hook) {
    return new Promise((resolve, reject) => {
      if (myCondition !== true) {
        return resolve(hook);
      }

      // call the original hook
      hooks.hashPassword(options)(hook)
        .then(hook => {
          // do something custom
          resolve(hook);
        })
        .catch(error => {
          // do any custom error handling
          error.message = 'my custom message';
          reject(error);
        });
    });
  };
}
```

Then simply use it like you normally would:

```
import hashPassword from './hooks/myHashPassword';

userService.before({
  create : [hashPassword()]
});
```

Hook examples

There are many ways in which hooks can be used. Below are some examples that show how to quickly add useful functionality to a service. The authentication chapter has more examples on how to use hooks for user authorization.

Setting Timestamps

If the database adapter does not support it already, timestamps can be easily added as a *before* hook like this:

```
app.service('todos').before({
  create(hook) {
    hook.data.created_at = new Date();
  },

  update(hook) {
    hook.data.updated_at = new Date();
  },

  patch(hook) {
    hook.data.updated_at = new Date();
  }
})
```

Fetching Related Items

Hooks can also be used to fetch related items from other services. The following hook checks for a `related` query parameter in `get` and if it exists, includes all todos for the user in the response:

```
app.service('users').after({
  get(hook) {
    // The user id
    const id = hook.result.id;

    if(hook.params.query.related) {
      return hook.app.service('todos').find({
        query: { user_id: id }
      }).then(todos => {
        // Set the todos on the user property
        hook.result.todos = todos;
        // Always return the hook object or `undefined`
        return hook;
      });
    }
  }
});
```

Validation

For a production app you need to do validation. With hooks it's actually pretty easy and validation methods can easily be reused.

```
app.service('users').before({
  create(hook) {
    // Don't create a user unless they accept our terms
    if (!hook.data.acceptedTerms) {
      throw new errors.BadRequest(`Invalid request`, {
        errors: [
          {
            path: 'acceptedTerms',
            value: hook.data.acceptedTerms,
            message: `You must accept the terms`
          }
        ]
      });
    }
  }
});
```

Sanitization

You might also need to correct or sanitize data that is sent to your app. Also pretty easy and you can check out some of our bundled hooks that cover some common use cases.

```
app.service('users').before({
  update(hook) {
    // Convert the user's age to an integer
    const sanitizedAge = parseInt(hook.data.age, 10);

    if (isNaN(age)) {
      throw new errors.BadRequest(`Invalid 'age' value ${hook.data.age}`, {
        errors: [
          {
            path: 'age',
            value: hook.data.age,
            message: `Invalid 'age' value ${hook.data.age}`
          }
        ]
      });
    }

    hook.data.age = sanitizedAge;
  }
});
```

Soft Delete

Sometimes you might not want to actually delete items in the database but just mark them as deleted. We can do this by adding a `remove` *before* hook, marking the todo as deleted and then setting `hook.result`. That way we can completely skip the original service method.

```
app.service('todos').before({
  remove(hook) {
    // Instead of calling the service remote, call `patch` and set deleted to `true`
    return this.patch(hook.id, { deleted: true }, hook.params).then(data => {
      // Set the result from `patch` as the method call result
      hook.result = data;
      // Always return the hook or `undefined`
      return hook;
    });
  },

  find(hook) {
    // Only query items that have not been marked as deleted
    hook.params.query.deleted = { $ne: true };
  }
});
```

> **ProTip:** Setting `hook.result` will only skip the actual method. *after* hooks will still run in the order they have been registered.

Built-in hooks

When it makes sense to do so, some plug-ins include their own hooks. The following plug-ins come bundled with useful hooks:

- `feathers-hooks` (see below)
- `feathers-mongoose`
- `feathers-authentication`

There are a few hooks included in the `feathers-hooks` module that are available on the hooks module.

populate

`populate(fieldName, { service: service, field: sourceField })`

The `populate` hook uses a property from the result (or every item if it is a list) to retrieve a single related object from a service and add it to the original object. It is meant to be used as an **after** hook on any service method.

```
const hooks = require('feathers-hooks');

// Given a `user_id` in a message, retrieve the user and
// add it in the `user` field.
app.service('messages').after(hooks.populate('user', {
  service: 'users',
  field: 'user_id'
}));
```

Options

- `fieldName` (**required**) - The field name you want to populate the related object on to.
- `service` (**required**) - The service you want to populate the object from.
- `field` (default: 'fieldName') [optional] - The field you want to look up the related object by from the `service`. By default it is the same as the target `fieldName`.

disable

Disables access to a service method completely or for a specific provider. All providers (REST, Socket.io and Primus) set the `params.provider` property which is what `disable` checks for. There are several ways to use the disable hook:

```
const hooks = require('feathers-hooks');

app.service('users').before({
  // Users can not be created by external access
  create: hooks.disable('external'),
  // A user can not be deleted through the REST provider
  remove: hooks.disable('rest'),
  // Disable calling `update` completely (e.g. to only support `patch`)
  update: hooks.disable(),
  // Disable the remove hook if the user is not an admin
  remove: hooks.disable(function(hook) {
    return !hook.params.user.isAdmin
  })
});
```

> **ProTip:** Service methods that are not implemented do not need to be disabled.

Options

- `providers` (default: *disables everything*) [optional] - The transports that you want to disable this service method for. Options are:
  - `socketio` - will disable the method for the Socket.IO provider
  - `primus` - will disable the method for the Primus provider
  - `rest` - will disable the method for the REST provider
  - `external` - will disable access from all providers making a service method only usable internally.
- `callback` (default: *runs when not called internally*) [optional] - A function that receives the `hook` object where you can put your own logic to determine whether this hook should run. Returns either `true` or `false`.

remove

Remove the given fields either from the data submitted (as a `before` hook for `create`, `update` or `patch`) or from the result (as an `after` hook). If the data is an array or a paginated `find` result the hook will remove the field for every item.

```
const hooks = require('feathers-hooks');

// Remove the hashed `password` and `salt` field after all method calls
app.service('users').after(hooks.remove('password', 'salt'));

// Remove _id for `create`, `update` and `patch`
app.service('users').before({
  create: hooks.remove('_id'),
  update: hooks.remove('_id'),
  patch: hooks.remove('_id')
})

// remove `email` field for all methods unless the
// requesting user is an admin.
app.service('users').after({
  all: hooks.remove('email', function(hook){
    return !hook.params.user.isAdmin;
  })
})
```

> **ProTip:** This hook will only fire when `params.provider` has a value (ie. when it is an external request over REST or Sockets.) or if you have passed your own custom condition function.

Options

- `fields` (**required**) - The fields that you want to remove from the object(s).
- `callback` (default: *runs when not called internally*) [optional] - A function that receives the `hook` object where you can put your own logic to determine whether this hook should run. Returns either `true` or `false`.

removeQuery

Remove the given fields from the query params. Can be used as a **before** hook for any service method.

```
const hooks = require('feathers-hooks');

// Remove _id from the query for all service methods
app.service('users').before({
  all: hooks.removeQuery('_id')
});

// remove `email` field for all methods unless the
// requesting user is an admin.
app.service('users').after({
  all: hooks.removeQuery('email', function(hook){
    return !hook.params.user.isAdmin;
  })
})
```

> **ProTip:** This hook will only fire when `params.provider` has a value (ie. when it is an external request over REST or Sockets.) or if you have passed your own custom condition function.

Options

- `fields` (**required**) - The fields that you want to remove from the query object.
- `callback` (default: *runs when not called internally*) [optional] - A function that receives the `hook` object where you can put your own logic to determine whether this hook should run. Returns either `true` or `false`.

pluck

Discard all other fields except for the provided fields either from the data submitted (as a `before` hook for `create`, `update` or `patch`) or from the result (as an `after` hook). If the data is an array or a paginated `find` result the hook will remove the field for every item.

```
const hooks = require('feathers-hooks');

// Only retain the hashed `password` and `salt` field after all method calls
app.service('users').after(hooks.pluck('password', 'salt'));

// Only keep the _id for `create`, `update` and `patch`
app.service('users').before({
  create: hooks.pluck('_id'),
  update: hooks.pluck('_id'),
  patch: hooks.pluck('_id')
})

// Only keep the `email` field for all methods unless the
// requesting user is an admin.
app.service('users').after({
  all: hooks.pluck('email', function(hook){
    return !hook.params.user.isAdmin;
  })
})
```

> **ProTip:** This hook will only fire when `params.provider` has a value (ie. when it is an external request over REST or Sockets.) or if you have passed your own custom condition function.

Options

- `fields` (**required**) - The fields that you want to retain from the object(s). All other fields will be discarded.
- `callback` (default: *runs when not called internally*) [optional] - A function that receives the `hook` object where you can put your own logic to determine whether this hook should run. Returns either `true` or `false`.

pluckQuery

Discard all other fields except for the given fields from the query params. Can be used as a **before** hook for any service method.

```
const hooks = require('feathers-hooks');

// Discard all other fields except for _id from the query
// for all service methods
app.service('users').before({
  all: hooks.pluckQuery('_id')
});

// only retain the `email` field for all methods unless the
// requesting user is an admin.
app.service('users').after({
  all: hooks.pluckQuery('email', function(hook){
    return !hook.params.user.isAdmin;
  })
})
```

> **ProTip:** This hook will only fire when `params.provider` has a value (ie. when it is an external request over REST or Sockets.) or if you have passed your own custom condition function.

Options

- `fields` (**required**) - The fields that you want to retain from the query object. All other fields will be discarded.
- `callback` (default: *runs when not called internally*) [optional] - A function that receives the `hook` object where you can put your own logic to determine whether this hook should run. Returns either `true` or `false`.

lowerCase

Lowercases the given fields either in the data submitted (as a `before` hook for `create`, `update` or `patch`) or in the result (as an `after` hook). If the data is an array or a paginated `find` result the hook will lowercase the field for every item.

```
const hooks = require('feathers-hooks');

// lowercase the `email` and `password` field before a user is created
app.service('users').before({
  create: hooks.lowerCase('email', 'username')
});
```

Options

- `fields` (**required**) - The fields that you want to lowercase from the retrieved object(s).

Connecting to a Feathers Server

Now that we went over how to set up REST and real-time APIs on the server we can look at how you can interact with a Feathers server from various clients.

Feathers as the client

Feathers itself can be used as a universal (isomorphic) client. That means that you can easily connect to remote services and register your own client side services and hooks. The Feathers client works in the browser with any front-end framework or in non-browser JavaScript environments like React Native and other NodeJS servers.

In the Feathers client chapter we will talk about how to use Feathers as a client in different environments and get it to connect to a Feathers server via REST, Socket.io and Primus.

Direct communication

In order to communicate with a Feathers API you don't need to use Feathers as the client. A Feathers server works great with any client that can connect through HTTP(S) to a REST API or - to also get real-time events - websockets. In the chapter for direct communication we will look at how to directly communicate with a Feathers API via REST, Socket.io and Primus.

Framework support

Because it is easy to integrate, Feathers does not currently have any official framework specific bindings. Work on iOS and Android SDKs are in progress.

To give you a better idea of how the Feathers client plays with other frameworks we've written some guides in the frameworks chapter. We are adding new ones all the time! If you are having any trouble with your framework of choice, create an issue and we'll try our best to help out.

Caveats

One important thing to know about Feathers is that it only exposes the official service methods to clients. While you can add and use any service method on the server, it is **not** possible to expose those custom methods to clients.

In the Why Feathers chapter we discussed how the *uniform interface* of services naturally translates into a REST API and also makes it easy to hook into the execution of known methods and emit events when they return. Adding support for custom methods would add a new level of complexity defining how to describe, expose and secure custom methods. This does not go well with Feathers approach of adding services as a small and well defined concept.

In general, almost anything that may require custom methods can also be done by creating other services. For example, a `userService.resetPassword` method can also be implemented as a password service that resets the password in the `create`:

```
class PasswordService {
  create(data) {
    const userId = data.user_id;
    const userService = this.app.service('user');

    userService.resetPassword(userId).then(user => {
      // Send an email with the new password
      return sendEmail(user);
    })
  }

  setup(app) {
    this.app = app;
  }
}
```

Universal Feathers

The Feathers client (`feathers/client`) module provides Feathers core functionality (registering and retrieving services, events etc.) without relying on Express. This makes it possible to use Feathers in any JavaScript environment like the browser, React Native or other NodeJS servers and to transparently connect to and use services from a Feathers API server.

If they are not universally usable already (like feathers-hooks), many plug-ins provide their own client modules:

- `feathers-socketio/client`
- `feathers-primus/client`
- `feathers-authentication/client` with support for:
  - Local authentication (username/password)
  - Token authentication (JWT)
  - OAuth2 (Facebook, LinkedIn, etc)
- `feathers-rest/client` with support for:
  - jQuery
  - Superagent
  - request
  - Fetch: works in supported browsers, React Native or modules like node-fetch.

> **Important:** The Feathers client libraries come transpiled to ES5 but require ES6 shims either through the babel-polyfill module or by including core.js in older browsers e.g. via `<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>`

Usage in NodeJS and client Module Loaders

For module loaders that support NPM like Browserify, Webpack or StealJS the Feathers client modules can be loaded individually. The following example sets up a Feathers client that uses a local Socket.io connection to communicate with remote services:

```
$ npm install feathers feathers-socketio feathers-hooks socket.io-client
```

```
const feathers = require('feathers/client')
const socketio = require('feathers-socketio/client');
const hooks = require('feathers-hooks');
const io = require('socket.io-client');

const socket = io('http://api.my-feathers-server.com');
const app = feathers()
  .configure(hooks())
  .configure(socketio(socket));

const messageService = app.service('messages');

messageService.on('created', message => console.log('Created a message', message));

// Use the messages service from the server
messageService.create({
  text: 'Message from client'
});
```

Usage with React Native

React Native currently requires a workaround due to an issue in Socket.io.

```
$ npm install feathers feathers-socketio feathers-hooks socket.io-client babel-polyfil
l
```

Create a `user-agent.js` with the following content:

```
window.navigator.userAgent = 'react-native';
```

Then in the main application file:

```
import 'babel-polyfill';
import './user-agent';
import io from 'socket.io-client';
import feathers from 'feathers/client';
import socketio from 'feathers-socketio/client';
import hooks from 'feathers-hooks';

const socket = io('http://api.my-feathers-server.com');
const app = feathers()
  .configure(hooks())
  .configure(socketio(socket));

const messageService = app.service('messages');

messageService.on('created', message => console.log('Created a message', message));

// Use the messages service from the server
messageService.create({
  text: 'Message from client'
});
```

feathers-client

feathers-client consolidates a standard set of client plugins into a single distributable that can be used standalone in the browser or with other module loaders (like RequireJS) that don't support NPM. The following modules are included:

- *feathers* as `feathers` (or the default module export)
- *feathers-hooks* as `feathers.hooks`
- *feathers-rest* as `feathers.rest`
- *feathers-socketio* as `feathers.socketio`
- *feathers-primus* as `feathers.primus`
- *feathers-authentication* as `feathers.authentication`

In the browser a client that connects to the local server via websockets can be initialized like this:

```
<script type="text/javascript" src="socket.io/socket.io.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.
js"></script>
<script type="text/javascript">
  var socket = io('http://api.my-feathers-server.com');
  var app = feathers()
    .configure(feathers.hooks())
    .configure(feathers.socketio(socket));

  var messageService = app.service('messages');

  messageService.on('created', function(message) {
    console.log('Someone created a message', message);
  });

  messageService.create({
    text: 'Message from client'
  });
</script>
```

In the following chapters we will discuss the different ways to connect to a Feathers server via REST, Socket.io and Primus.

Feathers Client + REST

Once set up on the server, there are several ways to connect to the REST API of a Feathers service. Keep in mind that while clients connected via websockets will receive real-time events from other REST API calls, you can't get real-time updates over the HTTP API without resorting to polling.

Using the Feathers client, the `feathers-rest/client` module allows you to connect to a Feathers service via a REST API using jQuery, request, Superagent or Fetch as the client library.

> **ProTip:** REST client services do emit `created`, `updated`, `patched` and `removed` events but only *locally for their own instance*. Real-time events from other clients can only be received by using a websocket connection.

> **ProTip:** The base URL is relative from where services are registered. That means that a service at `http://api.feathersjs.com/api/v1/messages` with a base URL of `http://api.feathersjs.com` would be available as `app.service('api/v1/messages')`. With a base URL of `http://api.feathersjs.com/api/v1` it would be `app.service('messages')`.

> **ProTip:** Notice how the REST client wrapper is always initialized using a base URL: `.configure(rest('http://api.feathersjs.com').superagent(superagent));`

jQuery

jQuery $.ajax requires an instance of jQuery passed to `feathers.jquery`. In most cases the global `jQuery`:

```
const feathers = require('feathers/client');
const rest = require('feathers-rest/client');
const host = 'http://api.feathersjs.com';
const app = feathers()
  .configure(rest(host).jquery(window.jQuery));
```

Request

The request object needs to be passed explicitly to `feathers.request`. Using request.defaults - which creates a new request object - is a great way to set things like default headers or authentication information:

```
const feathers = require('feathers/client');
const rest = require('feathers-rest/client');
const request = require('request');
const host = 'http://api.feathersjs.com';
const client = request.defaults({
  'auth': {
    'user': 'username',
    'pass': 'password',
    'sendImmediately': false
  }
});

const app = feathers()
  .configure(rest(host).request(client));
```

Superagent

Superagent currently works with a default configuration:

```
const feathers = require('feathers/client');
const rest = require('feathers-rest/client');
const superagent = require('superagent');
const host = 'http://api.feathersjs.com';
const app = feathers()
  .configure(rest(host).superagent(superagent));
```

Fetch

Fetch also uses a default configuration:

```
const feathers = require('feathers/client');
const rest = require('feathers-rest/client');
const host = 'http://api.feathersjs.com';
const fetch = require('node-fetch');
const app = feathers()
  .configure(rest(host).fetch(fetch));
```

Browser Usage

Using the Feathers client, the `feathers-rest/client` module can be configured to used for any Ajax requests:

```
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/superagent/1.2.0/superagent.min.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.js"></script>
<script type="text/javascript">
  const rest = feathers.rest('http://api.feathersjs.com');
  const app = feathers()
    .configure(feathers.hooks())
    .configure(rest.superagent(superagent));

  var messageService = app.service('messages');

  messageService.create({ text: 'Oh hai!'}).then(result => {
    console.log(result);
  }).catch(error => {
    console.error(error);
  });
</script>
```

Server Usage

Here's how to use the Feathers REST client in NodeJS.

```
$ npm install feathers feathers-rest feathers-hooks superagent
```

```
const feathers = require('feathers');
const superagent = require('superagent');
const client = require('feathers-rest/client');
const rest = client('http://my-feathers-server.com');

const app = feathers()
  .configure(hooks())
  .configure(rest.superagent(superagent));

// This will now connect to the http://my-feathers-server.com/messages API
const messageService = app.service('messages');

messageService.create({ text: 'Oh hai!'}).then(result => {
  console.log(result);
}).catch(error => {
  console.error(error);
});
```

React Native Usage

Here's how you can use Feathers client with the Fetch provider in React Native.

```
$ npm install feathers feathers-rest feathers-hooks
```

```
import React from 'react-native';
import hooks from 'feathers-hooks';
import feathers from 'feathers/client';
import rest from 'feathers-rest/client';

const app = feathers()
  .configure(feathers.hooks())
  .configure(rest('http://my-feathers-server.com').fetch(fetch));

// Get the message service that uses a REST connection
const messageService = app.service('messages');

messageService.create({ text: 'Oh hai!'})
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Feathers Client + Socket.io

With Socket.io configured on the server service methods and events will be available through a websocket connection. While using the REST API and just listening to real-time events on a socket is possible, Feathers also allows to call service methods through a websocket which, in most cases will be faster than REST HTTP.

Establishing the connection

Feathers sets up a normal Socket.io server that you can connect to using the Socket.io client either by loading the `socket.io-client` module or `/socket.io/socket.io.js` from the server. Unlike HTTP calls, websockets do not have a cross-origin restriction in the browser so it is possible to connect to any Feathers server. See below for platform specific examples.

> **ProTip**: The socket connection URL has to point to the server root which is where Feathers will set up Socket.io.

Client options

The Socket.io configuration (`socketio(socket [, options])`) can take settings which currently support:

- `timeout` (default: 5000ms) - The time after which a method call fails and times out. This usually happens when calling a service or service method that does not exist.

Browser Usage

Using the Feathers client, the `feathers-socketio/client` module can be configured to use that socket as the connection:

```
<script type="text/javascript" src="socket.io/socket.io.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.
js"></script>
<script type="text/javascript">
  var socket = io('http://api.feathersjs.com');
  var app = feathers()
    .configure(feathers.hooks())
    .configure(feathers.socketio(socket));

  var messageService = app.service('messages');

  messageService.on('created', function(message) {
    console.log('Someone created a message', message);
  });

  messageService.create({
    text: 'Message from client'
  });
</script>
```

Server Usage

Here's how to use the Feathers socket.io client in NodeJS. A great use case would be workers that need to update the server or broadcast to all connected clients.

```
$ npm install feathers feathers-socketio feathers-hooks socket.io-client
```

```
const feathers = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');

const socket = io('http://api.feathersjs.com');
const app = feathers().configure(socketio(socket));

// Get the message service that uses a websocket connection
const messageService = app.service('messages');

messageService.on('created', message => console.log('Someone created a message', messa
ge));
```

React Native Usage

Here's how you can use Feathers client with websockets in React Native.

```
$ npm install feathers feathers-socketio feathers-hooks socket.io-client
```

```
import React from 'react-native';
import hooks from 'feathers-hooks';
import feathers from 'feathers/client';
import socketio from 'feathers-socketio/client';
import io from 'socket.io-client';

// A hack so that you can still debug. Required because react native debugger runs in
a web worker, which doesn't have a window.navigator attribute.
if (window.navigator && Object.keys(window.navigator).length == 0) {
  window = Object.assign(window, { navigator: { userAgent: 'ReactNative' }});
}

const socket = io('http://api.feathersjs.com', { transports: ['websocket'] });
const app = feathers()
  .configure(feathers.hooks())
  .configure(socketio(socket));

// Get the message service that uses a websocket connection
const messageService = app.service('messages');

messageService.on('created', message => console.log('Someone created a message', messa
ge));
```

Feathers Client + Primus

Primus works very similar to Socket.io but supports a number of different real-time libraries. Once configured on the server service methods and events will be available through a Primus socket connection.

Establishing the connection

In the browser, the connection can be established by loading the client from `primus/primus.js` and instantiating a new `Primus` instance. Unlike HTTP calls, websockets do not have a cross-origin restriction in the browser so it is possible to connect to any Feathers server. See below for platform specific examples.

> **ProTip**: The socket connection URL has to point to the server root which is where Feathers will set up Primus.

Client options

The Primus configuration (`primus(connection [, options])`) can take settings which currently support:

- `timeout` (default: 5000ms) - The time after which a method call fails and times out. This usually happens when calling a service or service method that does not exist.

Browser Usage

Using the Feathers client, the `feathers-primus/client` module can be configured to use the Primus connection:

```
<script type="text/javascript" src="primus/primus.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.
js"></script>
<script type="text/javascript">
  var primus = new Primus('http://api.my-feathers-server.com');
  var app = feathers()
    .configure(feathers.hooks())
    .configure(feathers.primus(primus));

  var messageService = app.service('messages');

  messageService.on('created', function(message) {
    console.log('Someone created a message', message);
  });

  messageService.create({
    text: 'Message from client'
  });
</script>
```

Server Usage

This sets up Primus as a stand alone client in NodeJS using the websocket transport. If you are integrating with a separate server instance you can also connect without needing to require `primus-emitter`. You can read more in the Primus documentation.

```
$ npm install feathers feathers-primus feathers-hooks primus primus-emitter ws
```

```
const feathers = require('feathers');
const primus = require('feathers-primus/client');
const Primus = require('primus');
const Emitter = require('primus-emitter');
const hooks = require('feathers-hooks');
const Socket = Primus.createSocket({
  transformer: 'websockets',
  plugin: {
    'emitter': Emitter
  }
});
const socket = new Socket('http://api.feathersjs.com');
const app = feathers()
  .configure(hooks())
  // Configure and change the default timeout to one second
  .configure(primus(socket, { timeout: 1000 }));

// Get the message service that uses a websocket connection
const messageService = app.service('messages');

messageService.on('created', message => console.log('Someone created a message', messa
ge));
```

React Native Usage

TODO (EK): Add some of the specific React Native things we needed to change to properly support websockets. I'm pretty sure this doesn't completely work so it's still a WIP. PR's welcome!

```
$ npm install feathers feathers-primus feathers-hooks primus
```

```
import React from 'react-native';
import hooks from 'feathers-hooks';
import {client as feathers} from 'feathers';
import {client as primus} from 'feathers-primus';

let Socket = primus.socket;

// A hack so that you can still debug. Required because react native debugger runs in
a web worker, which doesn't have a window.navigator attribute.
if (window.navigator && Object.keys(window.navigator).length === 0) {
  window.navigator.userAgent = 'ReactNative';
}

const socket = new Socket('http://api.feathersjs.com');
const app = feathers()
  .configure(hooks())
  .configure(primus(socket));

// Get the message service that uses a websocket connection
const messageService = app.service('messages');

messageService.on('created', message => console.log('Someone created a message', messa
ge));
```

Direct connection

Although we do recommend using Feathers as the client in JavaScript environments communicating with a Feathers API server can be done with any client in any language that supports HTTP(S) or - to also get real-time events - websockets.

In this chapter we will go over:

- Using the REST API directly
- Connecting to and using Socket.io events
- Using Primus

Vanilla REST

Once set up on the server, there are several ways to connect to the REST API of a Feathers service. Keep in mind that while clients connected via websockets will receive real-time events from other REST API calls, you can't get real-time updates over the HTTP API without resorting to polling.

You can communicate with a Feathers server using any HTTP REST client. The following section describes what HTTP method, body and query parameters belong to which service method call.

All query parameters in a URL will be set as `params.query` on the server. Other service parameters can be set through hooks and Express middleware. URL query parameter values will always be strings. Conversion (e.g. the string `'true'` to boolean `true`) can be done in a hook as well.

The body type for `POST`, `PUT` and `PATCH` requests is determined by the Express body-parser middleware which has to be registered *before* any service. You should also make sure you are setting your `Accepts` header to `application/json`.

find

Retrieves a list of all matching resources from the service

```
GET /messages?status=read&user=10
```

Will call `messages.find({ query: { status: 'read', user: '10' } })` on the server.

get

Retrieve a single resource from the service.

```
GET /messages/1
```

Will call `messages.get(1, {})` on the server.

```
GET /messages/1?fetch=all
```

Will call `messages.get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
POST /messages
{ "text": "I really have to iron" }
```

Will call `messages.create({ "text": "I really have to iron" }, {})` on the server.

```
POST /messages
[
  { "text": "I really have to iron" },
  { "text": "Do laundry" }
]
```

update

Completely replace a single or multiple resources.

```
PUT /messages/2
{ "text": "I really have to do laundry" }
```

Will call `messages.update(2, { "text": "I really have to do laundry" }, {})` on the server. When no `id` is given by sending the request directly to the endpoint something like:

```
PUT /messages?complete=false
{ "complete": true }
```

Will call `messages.update(null, { "complete": true }, { query: { complete: 'false' } })` on the server.

> **ProTip:** `update` is normally expected to replace an entire resource which is why the database adapters only support `patch` for multiple records.

patch

Merge the existing data of a single or multiple resources with the new `data`.

```
PATCH /messages/2
{ "read": true }
```

Will call `messages.patch(2, { "read": true }, {})` on the server. When no `id` is given by sending the request directly to the endpoint something like:

```
PATCH /messages?complete=false
{ "complete": true }
```

Will call `messages.patch(null, { complete: true }, { query: { complete: 'false' } })` on the server to change the status for all read messages.

This is supported out of the box by the Feathers database adapters

remove

Remove a single or multiple resources:

```
DELETE /messages/2?cascade=true
```

Will call `messages.remove(2, { query: { cascade: 'true' } })`. When no `id` is given by sending the request directly to the endpoint something like:

```
DELETE /messages?read=true
```

Will call `messages.remove(null, { query: { read: 'true' } })` to delete all read messages.

Vanilla Socket.io

With Socket.io configured on the server service methods and events will be available through a websocket connection. While using the REST API and just listening to real-time events on a socket is possible, Feathers also allows to call service methods through a websocket which, in most cases will be faster than REST HTTP.

Establishing the connection

Feathers sets up a normal Socket.io server that you can connect to using the Socket.io client either by loading the `socket.io-client` module or `/socket.io/socket.io.js` from the server. Unlike HTTP calls, websockets do not have a cross-origin restriction in the browser so it is possible to connect to any Feathers server. See below for platform specific examples.

> **ProTip**: The socket connection URL has to point to the server root which is where Feathers will set up Socket.io.

Calling service methods

Service methods can be called by emitting a `<servicepath>::<methodname>` event with the method parameters. `servicepath` is the name the service has been registered with (in `app.use`) without leading or trailing slashes. An optional callback following the `function(error, data)` Node convention will be called with the result of the method call or any errors that might have occurred.

`params` will be set as `params.query` in the service method call. Other service parameters can be set through a Socket.io middleware.

`find`

Retrieves a list of all matching resources from the service

```
socket.emit('messages::find', { status: 'read', user: 10 }, (error, data) => {
  console.log('Found all messages', data);
});
```

Will call `messages.find({ query: { status: 'read', user: 10 } })` on the server.

get

Retrieve a single resource from the service.

```
socket.emit('messages::get', 1, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, {})` on the server.

```
socket.emit('messages::get', 1, { fetch: 'all' }, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
socket.emit('messages::create', {
  "text": "I really have to iron"
}, (error, message) => {
  console.log('Todo created', message);
});
```

Will call `messages.create({ "text": "I really have to iron" }, {})` on the server.

```
socket.emit('messages::create', [
  { "text": "I really have to iron" },
  { "text": "Do laundry" }
]);
```

Will call `messages.create` with the array.

update

Completely replace a single or multiple resources.

```
socket.emit('messages::update', 2, {
  "text": "I really have to do laundry"
}, (error, message) => {
  console.log('Todo updated', message);
});
```

Will call `messages.update(2, { "text": "I really have to do laundry" }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
socket.emit('messages::update', null, {
  complete: true
}, { complete: false });
```

Will call `messages.update(null, { "complete": true }, { query: { complete: 'false' } })` on the server.

> **ProTip:** `update` is normally expected to replace an entire resource which is why the database adapters only support `patch` for multiple records.

`patch`

Merge the existing data of a single or multiple resources with the new `data`.

```
socket.emit('messages::patch', 2, {
  read: true
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `messages.patch(2, { "read": true }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
socket.emit('messages::patch', null, {
  complete: true
}, {
  complete: false
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `messages.patch(null, { complete: true }, { query: { complete: false } })` on the server to change the status for all read messages.

This is supported out of the box by the Feathers database adapters

`remove`

Remove a single or multiple resources:

```
socket.emit('messages::remove', 2, { cascade: true }, (error, message) => {
  console.log('Removed a message', message);
});
```

Will call `messages.remove(2, { query: { cascade: true } })` on the server. The `id` can also be `null` to remove multiple resources:

```
socket.emit('messages::remove', null, { read: true });
```

Will call `messages.remove(null, { query: { read: 'true' } })` on the server to delete all read messages.

Listening to events

Listening to service events allows real-time behaviour in an application. Service events are sent to the socket in the form of `servicepath eventname` .

created

The `created` event will be published with the callback data when a service `create` returns successfully.

```
<script>
  var socket = io('http://localhost:3030/');

  socket.on('messages created', function(message) {
    console.log('Got a new Todo!', message);
  });
</script>
```

updated, patched

The `updated` and `patched` events will be published with the callback data when a service `update` or `patch` method calls back successfully.

```
<script>
  var socket = io('http://localhost:3030/');

  socket.on('my/messages updated', function(message) {
    console.log('Got an updated Todo!', message);
  });

  socket.emit('my/messages::update', 1, {
    text: 'Updated text'
  }, {}, function(error, callback) {
   // Do something here
  });
</script>
```

removed

The `removed` event will be published with the callback data when a service `remove` calls back successfully.

```
<script>
  var socket = io('http://localhost:3030/');

  socket.on('messages removed', function(message) {
    // Remove element showing the Todo from the page
    $('#message-' + message.id).remove();
  });
</script>
```

Vanilla Primus

Primus works very similar to Socket.io but supports a number of different real-time libraries. Once configured on the server service methods and events will be available through a Primus socket connection.

Establishing the connection

In the browser, the connection can be established by loading the client from `primus/primus.js` and instantiating a new `Primus` instance. Unlike HTTP calls, websockets do not have a cross-origin restriction in the browser so it is possible to connect to any Feathers server.

See the Primus docs for more details.

> **ProTip**: The socket connection URL has to point to the server root which is where Feathers will set up Primus.

Calling service methods

Service methods can be called by emitting a `<servicepath>::<methodname>` event with the method parameters. `servicepath` is the name the service has been registered with (in `app.use`) without leading or trailing slashes. An optional callback following the `function(error, data)` Node convention will be called with the result of the method call or any errors that might have occurred.

`params` will be set as `params.query` in the service method call. Other service parameters can be set through a Primus middleware.

`find`

Retrieves a list of all matching resources from the service

```
primus.send('messages::find', { status: 'read', user: 10 }, (error, data) => {
  console.log('Found all messages', data);
});
```

Will call `messages.find({ query: { status: 'read', user: 10 } })` on the server.

get

Retrieve a single resource from the service.

```
primus.send('messages::get', 1, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, {})` on the server.

```
primus.send('messages::get', 1, { fetch: 'all' }, (error, message) => {
  console.log('Found message', message);
});
```

Will call `messages.get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
primus.send('messages::create', {
  "text": "I really have to iron"
}, (error, message) => {
  console.log('Message created', message);
});
```

Will call `messages.create({ "text": "I really have to iron" }, {})` on the server.

```
primus.send('messages::create', [
  { "text": "I really have to iron" },
  { "text": "Do laundry" }
]);
```

Will call `messages.create` on the server with the array.

update

Completely replace a single or multiple resources.

```
primus.send('messages::update', 2, {
  "text": "I really have to do laundry"
}, (error, message) => {
  console.log('Message updated', message);
});
```

Will call `messages.update(2, { "text": "I really have to do laundry" }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
primus.send('messages::update', null, {
  complete: true
}, { complete: false });
```

Will call `messages.update(null, { "complete": true }, { query: { complete: 'false' } })` on the server.

> **ProTip:** `update` is normally expected to replace an entire resource which is why the database adapters only support `patch` for multiple records.

`patch`

Merge the existing data of a single or multiple resources with the new `data`.

```
primus.send('messages::patch', 2, {
  read: true
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `messages.patch(2, { "read": true }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
primus.send('messages::patch', null, {
  complete: true
}, {
  complete: false
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `messages.patch(null, { complete: true }, { query: { complete: false } })` on the server to change the status for all read messages.

This is supported out of the box by the Feathers database adapters

`remove`

Remove a single or multiple resources:

```
primus.send('messages::remove', 2, { cascade: true }, (error, message) => {
  console.log('Removed a message', message);
});
```

Will call `messages.remove(2, { query: { cascade: true } })` on the server. The `id` can also be `null` to remove multiple resources:

```
primus.send('messages::remove', null, { read: true });
```

Will call `messages.remove(null, { query: { read: 'true' } })` on the server to delete all read messages.

Listening to events

Listening to service events allows real-time behaviour in an application. Service events are sent to the socket in the form of `servicepath eventname` .

created

The `created` event will be published with the callback data when a service `create` returns successfully.

```
<script>
  primus.on('messages created', function(message) {
    console.log('Got a new Message!', message);
  });
</script>
```

updated, patched

The `updated` and `patched` events will be published with the callback data when a service `update` or `patch` method calls back successfully.

```
<script>
  primus.on('my/messages updated', function(message) {
    console.log('Got an updated Message!', message);
  });

  primus.send('my/messages::update', 1, {
    text: 'Updated text'
  }, {}, function(error, callback) {
   // Do something here
  });
</script>
```

removed

The `removed` event will be published with the callback data when a service `remove` calls back successfully.

```
<script>
  primus.on('messages removed', function(message) {
    // Remove element showing the Message from the page
    $('#message-' + message.id).remove();
  });
</script>
```

Frameworks

Because it is easy to integrate on the client, Feathers currently does not have any official framework specific bindings and iOs and Android clients are in the works.

In this chapter we will show some guides how to integrate Feathers with other JavaScript frameworks. We are adding new ones all the time! If you are having any trouble with your framework of choice, create an issue and we'll try our best to help out.

All guides will show how to create a frontend for the API built in the Your First App section. If you haven't done so you'll want to go through that tutorial or you can find a working example here.

Feathers + React

React is *a JavaScript library for building user interfaces*, which makes it ideal for building real-time applications with Feathers. Let React take care of the view rendering and let Feathers client do the heavy lifting for communicating with your server and managing your data.

In this guide we will create a plain React web front-end for the chat API built in the Your First App section.

If you haven't done so you'll want to go through that tutorial or you can find a working example here.

Setting up the HTML page

React and the Feathers client modules can be loaded individually via npm through a module loader like Webpack. To get up and running more quickly for this guide though, let's use the following HTML page as `public/chat.html`:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scal
e=1, user-scalable=0" />
    <title>Feathers Chat</title>
    <link rel="shortcut icon" href="img/favicon.png">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/chat.css">

    <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4
/core.min.js"></script>
    <script src="//fb.me/react-0.14.7.min.js"></script>
    <script src="//fb.me/react-dom-0.14.7.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></
script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/moment.js/2.12.0/moment.js"></script>

    <script src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script type="text/babel" src="app.jsx"></script>
  </head>
  <body>
  </body>
</html>
```

> **ProTip:** This setup is not recommended for a production environment because Babel will transpile our `public/app.jsx` every time on the fly in the browser. Find out how to use React with NPM in the React documentation and how to use the Feathers client modules in the client chapter.

Application Bootstrap

All of the code examples that follow should be appended to the `public/app.jsx` file, which the HTML file already loads. The first step is to add a placeholder user in the event that a user image or other user information doesn't exist:

```
// A placeholder image if the user does not have one
const PLACEHOLDER = 'https://placeimg.com/60/60/people';
// An anonymous user if the message does not have that information
const dummyUser = {
  avatar: PLACEHOLDER,
  email: 'Anonymous'
};
```

Now we can add this code to set up the Socket.io connection and the Feathers app:

```
// Establish a Socket.io connection
const socket = io();
// Initialize our Feathers client application through Socket.io
// with hooks and authentication.
const app = feathers()
  .configure(feathers.socketio(socket))
  .configure(feathers.hooks())
  // Use localStorage to store our login token
  .configure(feathers.authentication({
    storage: window.localStorage
  }));
```

Adding components

Each of your components will use the Feathers application we initialized above. The first component shows a form that when submitted creates a new message on the `messages` service:

```
const ComposeMessage = React.createClass({
  getInitialState() {
    return { text: '' };
  },

  updateText(ev) {
    this.setState({ text: ev.target.value });
  },

  sendMessage(ev) {
    // Get the messages service
    const messageService = app.service('messages');
    // Create a new message with the text from the input field
    messageService.create({
      text: this.state.text
    }).then(() => this.setState({ text: '' }));

    ev.preventDefault();
  },

  render() {
    return <form className="flex flex-row flex-space-between"
        onSubmit={this.sendMessage}>
      <input type="text" name="text" className="flex flex-1"
        value={this.state.text} onChange={this.updateText} />
      <button className="button-primary" type="submit">Send</button>
    </form>;
  }
});
```

The next component shows a list of user and allows to log out of the application;

```
const UserList = React.createClass({
  logout() {
    app.logout().then(() => window.location.href = '/index.html');
  },

  render() {
    const users = this.props.users;

    return <aside className="sidebar col col-3 flex flex-column flex-space-between">
      <header className="flex flex-row flex-center">
        <h4 className="font-300 text-center">
          <span className="font-600 online-count">{users.length}</span> users
        </h4>
      </header>

      <ul className="flex flex-column flex-1 list-unstyled user-list">
        {users.map(user =>
          <li>
            <a className="block relative" href="#">
              <img src={user.avatar || PLACEHOLDER} className="avatar" />
              <span className="absolute username">{user.email}</span>
            </a>
          </li>
        )}
      </ul>
      <footer className="flex flex-row flex-center">
        <a href="#" className="logout button button-primary" onClick={this.logout}>
          Sign Out
        </a>
      </footer>
    </aside>;
  }
});
```

Now want to display a list of messages:

```
const MessageList = React.createClass({
  // Render a single message
  renderMessage(message) {
    const sender = message.sentBy || dummyUser;

    return <div className="message flex flex-row">
      <img src={sender.avatar || PLACEHOLDER} alt={sender.email} className="avatar" />
      <div className="message-wrapper">
        <p className="message-header">
          <span className="username font-600">{sender.email}</span>
          <span className="sent-date font-300">
            {moment(message.createdAt).format('MMM Do, hh:mm:ss')}
          </span>
        </p>
        <p className="message-content font-300">
          {message.text}
        </p>
      </div>
    </div>;
  },

  render() {
    return <main className="chat flex flex-column flex-1 clear">
      {this.props.messages.map(this.renderMessage)}
    </main>;
  }
});
```

Finally we need a main component that retrieves all messages and users, listens to real-time events and passes the data to the components we previously created:

```
const ChatApp = React.createClass({
  getInitialState() {
    return {
      users: [],
      messages: []
    };
  },

  componentDidUpdate: function() {
    // Whenever something happened, scroll to the bottom of the chat window
    const node = this.getDOMNode().querySelector('.chat');
    node.scrollTop = node.scrollHeight - node.clientHeight;
  },

  componentDidMount() {
    const userService = app.service('users');
    const messageService = app.service('messages');

    // Find all users initially
    userService.find().then(page => this.setState({ users: page.data }));
    // Listen to new users so we can show them in real-time
    userService.on('created', user => this.setState({
      users: this.state.users.concat(user)
    }));

    // Find the last 10 messages
    messageService.find({
      query: {
        $sort: { createdAt: -1 },
        $limit: this.props.limit || 10
      }
    }).then(page => this.setState({ messages: page.data.reverse() }));
    // Listen to newly created messages
    messageService.on('created', message => this.setState({
      messages: this.state.messages.concat(message)
    }));
  },

  render() {
    return <div className="flex flex-row flex-1 clear">
      <UserList users={this.state.users} />
      <div className="flex flex-column col col-9">
        <MessageList users={this.state.users} messages={this.state.messages} />
        <ComposeMessage />
      </div>
    </div>
  }
});
```

Rendering and authenticating

The chat application is set up to redirect from `login.html` to our `chat.html` page on successful login. The Feathers server has already authenticated the user with email and password and put a JWT in a cookie. Feathers client detects this token for us so we just have to call app.authenticate to authenticate that user using the token. Once authenticated successfully we render the main layout with the `ChatApp` component:

```
app.authenticate().then(() => {
  ReactDOM.render(<div id="app" className="flex flex-column">
    <header className="title-bar flex flex-row flex-center">
      <div className="title-wrapper block center-element">
        <img className="logo" src="http://feathersjs.com/img/feathers-logo-wide.png"
          alt="Feathers Logo" />
        <span className="title">Chat</span>
      </div>
    </header>

    <ChatApp />
  </div>, document.body);
}).catch(error => {
  if(error.code === 401) {
    window.location.href = '/login.html'
  }

  console.error(error);
});
```

That's it. We now have a real-time chat front-end built with React and Feathers.

Feathers + React Native

We haven't had time to publish this guide yet. If you feel like you could contribute one feel free to open a pull request.

We do however, have a working React Native Chat App that you can peruse in the mean time.

Feathers + jQuery

You don't always need a full on framework. Feathers and the Feathers client also work great to add real-time capability to a vanilla JavaScript or jQuery application. In this guide we will create a jQuery front-end for the chat API built in the Your First App section.

If you haven't done so you'll want to go through that tutorial or you can find a working example here.

> **ProTip:** This guide uses ES6 syntax which is only available in newer browsers and IE edge. If you want to use the app in older browsers you need to include a transpiler like Babel.

Setting up the HTML page

The first step is getting the HTML skeleton for the chat application up. You can do so by pasting the following HTML into `public/chat.html` (which is the page the guide app redirects to after a successful login):

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1, user-scalable=0"
 />
    <title>Feathers Chat</title>
    <link rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/publ
ic/chat.css">
  </head>
  <body>
    <div id="app" class="flex flex-column">
      <header class="title-bar flex flex-row flex-center">
        <div class="title-wrapper block center-element">
          <img class="logo" src="http://feathersjs.com/img/feathers-logo-wide.png"
            alt="Feathers Logo">
          <span class="title">Chat</span>
        </div>
      </header>

      <div class="flex flex-row flex-1 clear">
        <aside class="sidebar col col-3 flex flex-column flex-space-between">
```

```
          <header class="flex flex-row flex-center">
            <h4 class="font-300 text-center">
              <span class="font-600 online-count">0</span> users
            </h4>
          </header>

          <ul class="flex flex-column flex-1 list-unstyled user-list"></ul>
          <footer class="flex flex-row flex-center">
            <a href="/login.html" class="logout button button-primary">
              Sign Out
            </a>
          </footer>
        </aside>

        <div class="flex flex-column col col-9">
          <main class="chat flex flex-column flex-1 clear"></main>

          <form class="flex flex-row flex-space-between" id="send-message">
            <input type="text" name="text" class="flex flex-1">
            <button class="button-primary" type="submit">Send</button>
          </form>
        </div>
      </div>
    </div>
    <script src="//cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></
script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/moment.js/2.12.0/moment.js"></script>

    <script src="//code.jquery.com/jquery-2.2.1.js"></script>
    <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4
/core.min.js"></script>
    <script src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.js">
    </script>
    <script src="/socket.io/socket.io.js"></script>
    <script type="text/babel" src="app.js"></script>
  </body>
</html>
```

This sets everything up we need including some styles, the Feathers client, jQuery and MomentJS (to format dates).

jQuery code

Our chat functionality will live in `public/app.js`. First, let's create some functions that use jQuery (and ES6 strings) to render a single user and message:

```
'use strict';

// A placeholder image if the user does not have one
const PLACEHOLDER = 'https://placeimg.com/60/60/people';
// An anonymous user if the message does not have that information
const dummyUser = {
  image: PLACEHOLDER,
  email: 'Anonymous'
};
// The total number of users
let userCount = 0;

function addUser(user) {
  // Update the number of users
  $('.online-count').html(++userCount);
  // Add the user to the list
  $('.user-list').append(`<li>
    <a class="block relative" href="#">
      <img src="${user.avatar || PLACEHOLDER}" alt="" class="avatar">
      <span class="absolute username">${user.email}</span>
    </a>
  </li>`);
}

// Renders a new message and finds the user that belongs to the message
function addMessage(message) {
  // Find the user belonging to this message or use the anonymous user if not found
  const sender = message.sentBy || dummyUser;
  const chat = $('.chat');

  chat.append(`<div class="message flex flex-row">
    <img src="${sender.avatar || PLACEHOLDER}" alt="${sender.email}" class="avatar">
    <div class="message-wrapper">
      <p class="message-header">
        <span class="username font-600">${sender.email}</span>
        <span class="sent-date font-300">${moment(message.createdAt).format('MMM Do, h
h:mm:ss')}</span>
      </p>
      <p class="message-content font-300">${message.text}</p>
    </div>
  </div>`);

  chat.scrollTop(chat[0].scrollHeight - chat[0].clientHeight);
}
```

Now we can set up the Feathers client. Because we also want real-time we will use a Socket.io connection:

```
// Establish a Socket.io connection
const socket = io();
// Initialize our Feathers client application through Socket.io
// with hooks and authentication.
const app = feathers()
  .configure(feathers.socketio(socket))
  .configure(feathers.hooks())
  // Use localStorage to store our login token
  .configure(feathers.authentication({
    storage: window.localStorage
  }));

// Get the Feathers services we want to use
const userService = app.service('users');
const messageService = app.service('messages');
```

Next, we set up event handlers for logout and when someone submits the message form:

```
$('#send-message').on('submit', function(ev) {
  // This is the message text input field
  const input = $(this).find('[name="text"]');

  // Create a new message and then clear the input field
  messageService.create({
    text: input.val()
  }).then(message => input.val(''));

  ev.preventDefault();
});

$('.logout').on('click', function() {
  app.logout().then(() => window.location.href = '/index.html');
});
```

When submitting the form, we create a new message with the text from the input field. When clicking the logout button we will call `app.logout()` and then redirect back to the login page.

The chat application is set up to redirect from `login.html` to our `chat.html` page on successful login. This means that we already know what user is logged in so we just have to call app.authenticate to authenticate that user (and redirect back to the login page if it fails). Then we retrieve the 25 newest messages, all the users and listen to events to make real-time updates:

```
app.authenticate().then(() => {
  // Find the latest 10 messages. They will come with the newest first
  // which is why we have to reverse before adding them
  messageService.find({
    query: {
      $sort: { createdAt: -1 },
      $limit: 25
    }
  }).then(page => page.data.reverse().forEach(addMessage));

  // Listen to created events and add the new message in real-time
  messageService.on('created', addMessage);

  // Find all users
  userService.find().then(page => {
    const users = page.data;

    // Add every user to the list
    users.forEach(addUser);
  });

  // We will also see when new users get created in real-time
  userService.on('created', addUser);
})
// On unauthorized errors we just redirect back to the login page
.catch(error => {
  if(error.code === 401) {
    window.location.href = '/login.html'
  }
});
```

That's it. We now have a real-time chat application front-end built in jQuery.

Feathers + CanJS

We haven't had time to publish this guide yet. If you feel like you could contribute one feel free to open a pull request.

Feathers + Angular

We haven't had time to publish this guide yet. If you feel like you could contribute one feel free to open a pull request.

Feathers + Angular 2

Angular 2 and Feathers work together wonderfully through Angular's services. Go ahead and read up on them if you aren't familiar with services or how they differ in Angular 2.x from 1.x.

We'll be setting up the basic service structure for an application that uses `feathers-socketio` but you should find this structure is flexible enough to accommodate other setups.

Using TypeScript

If you're using TypeScript to develop with Angular 2, Feathers and its related modules should be loaded into your app through npm packages. `npm install` the ones you'll need:

```
npm install feathers feathers-hooks feathers-rest feathers-socketio socket.io-client f
eathers-authentication
```

You'll also need a module loader, a commonly used one is Webpack. If you've gone through the Angular 2 quickstart you should be good to go.

Finally you'll need to load the feathers modules you need in a TypeScript file. Make a new file (I called mine `feathers.service.ts`) and include the following.

```
const feathers = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');
const localstorage = require('feathers-localstorage');
const hooks = require('feathers-hooks');
const rest = require('feathers-rest/client');
const authentication = require('feathers-authentication/client');
```

Notice that we **aren't using the ES6 import syntax**. As of the time of writing this, Feathers does not have TypeScript definitions, so TypeScript is unable to load the packages in that manner. As an alternative, we can use the standard CommonJS `require` syntax. It'll work the same, we just won't have the helpful intellisense that some IDE's provide.

Using ES5

Feathers has a single convenient file for the client which you can include in script tags called feathers-client.

```
<script src="/path/to/feathers-client.js"></script>
```

Setup

Now let's jump right in! We'll be using TypeScript from this point forward but all of this has a JS equivalent, which you can find on the Angular 2 JavaScript docs.

Make a `feathers.service.ts` file if you haven't already. This will contain the setup for our application to interact with Feathers.

REST

Let's create a new Angular service that uses REST to communicate with our server. In addition to the packages we previously installed, we also have to include a client REST library. In this guide, we'll be using Superagent.

```
import { Injectable } from 'angular2/core';
const superagent = require('superagent');

const HOST = 'http://localhost:3000'; // Your base server URL here
@Injectable()
export class RestService {
  private _app: any;
  constructor() {
    this._app = feathers() // Initialize feathers
      .configure(rest(HOST).superagent(superagent)) // Fire up rest
      .configure(hooks()) // Configure feathers-hooks
  }
}
```

Notice the `@Injectable()` decorator at the top of our service. This is important to make sure this service can be used by the rest of our app correctly.

Socket.io

Our socket.io app setup doesn't look much different!

```
@Injectable()
export class SocketService extends Service {
  public socket: SocketIOClient.Socket;
  private _app: any;

  constructor() {
    this.socket = io(HOST);
    this._app = feathers()
      .configure(socketio(this.socket))
      .configure(hooks())
  }
}
```

Remember, the `io` is the `socket.io-client` package that we included earlier in the guide. I also recommend keeping `socket` public as I've done in the above code example so you can use socket.io outside of the Feathers service in case you need to.

If we so choose, we can leave both the socket.io and REST app setups in the same file and use them both equally. That way, we can selectively use either one depending on the action we're doing!

Creating our first resource

That's all the setup we need to start interacting with our Feathers API. For the purpose of demonstration, lets make a `message.service.ts`. It will serve as a template for other services.

```
import { RestService, SocketService } from './feathers.service';

@Injectable()
export class MessageService {
  private _socket;
  private _rest;

  constructor(
    private _socketService: SocketService,
    private _restService: RestService
  ) {
    // Let's get both the socket.io and REST feathers services for messages!
    this._rest = _restService.getService('messages');
    this._socket = _socketService.getService('messages');
  }
}
```

As you can see, we imported the `feathers.service.ts` code we previously wrote, and set class instance variables to the services we want to interact with.

> **ProTip:** Remember that funny-looking `@Injectable()` decorator we used earlier? Angular 2 uses it to emit metadata about a service. Without it, we wouldn't be able to inject one service into another, like we just did.

Now let's write methods that will abstractly allow for our components to access our API!

```
  find(query: any) {
    return this._rest.find(query);
  }

  get(id: string, query: any) {
    return this._rest.get(id, query);
  }

  create(message: any) {
    return this._rest.create(message);
  }

  remove(id: string, query: any) {
    return this._socket.remove(id, query);
  }
```

You can write similar functions for other REST methods (PATCH, UPDATE) as well.

Notice that in these methods we're essentially just returning the promise that feathers gives us. This is a necessary step of abstraction, however, for a couple reasons.

1) We can switch between socket.io and REST whenever we want, and component code doesn't need to change. Look at our remove method! We made it use socket.io, but our component doesn't need to care! 2) We can create more complex behaviors whenever a method is called (e.g. a user is logged in after they are created, custom error handling, etc.)

Think you're done? **Not so fast!** If we run our code now, we're going to get an error. In Angular, we have to instantiate our services somewhere, and right now we have nothing creating our RestService or SocketService (if you're wondering about MessageService, sit tight)! Add them to your bootstrap call.

```
bootstrap(YourMainComponent, [
  SocketService,
  RestService
])
```

We'll explain more later, but believe it or not, that's all we need for our components to start handling our data!

Using a service in a component

Let's go to our `app.component.ts` which you should have from Angular 2 getting started. First things first, we import our newly created service.

```
import {MessageService} from '../services/message.service';
```

Providers

Now let's add it as a *provider* to the Angular 2 App Component.

```
import {Component} from 'angular2/core';
@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1>',
    providers: [ MessageService ]
})
export class AppComponent { }
```

So what does `providers` do? Every time you add MessageService as a provider to a component, **a new instance of the service is created.** In our application, in all likelihood we only ever need one instance of MessageService, so this should be **the only time we provide it**. If you want to make this service accessible globally throughout our application, rather than tying the service to a specific component, we can *bootstrap* it in our `main.ts` .

```
bootstrap(AppComponent, [
    SocketService,
    RestService,
    MessageService
])
```

That's what we did earlier! We created a global instance of RestService and SocketService, since we won't be providing them directly in any component (they're just for MessageService).

TL;DR Either *provide* your service on a single top-level component or *bootstrap* it. Don't do both, and don't `provide` to a component's children.

Injecting our Service

Now that we've created an instance of our service, let's give our component access to it. Simply providing it is not enough - you'll need to add it to the component's constructor.

```
  constructor (
    private _messageService: MessageService
  ) { }
```

Adding that type information, `: MessageService` gives Angular the information it needs to inject our service into the component!

Declaring our reference to the service instance private does a couple things. First, TypeScript's compiler will throw an error if we're trying to access this reference to the service from outside our component. Second, it is equivalent to writing the following

```
constructor (_messageService: MessageService) {
   this._messageService = _messageService;
}
```

And we all need a little less of `this` in our lives.

Using our service

Now we can access the service methods from our component. Let's get a list of all of our messages.

```
export class AppComponent {
  private _messages: any[] = [];
  // Called once when the component is early in its creation
  ngOnInit() {
    this._messageService.find().then(messages => {
       this._messages = messages;
    });
  }
}
```

And that's it! Our component now has access to our messages via `_messages`. From here we can do all sorts of fun Angular-y things. Let's display our Messages in a list.

```
@Component({
    ...
    template: `
      <div class="message" *ngFor="#message of _messages">
         <h1 class="title">{{message.title}}</h1>
         <div class="description">{{message.description}}</div>
      </div>
    `,
})
```

There are nicer ways of doing this, but you get the idea. What if we want to remove a message when a user clicks it?

```
@Component({
    ...
    template: `
      <div class="message" *ngFor="#message of _messages" (click)="removeMessage(messa
ge)">
         ...
      </div>
    `,
})
export class AppComponent () {
  removeMessage (message) {
    this._messageService.remove(message.id);
  }
}
```

Wrapping up

Structuring our interactions with Feathers in this way is very powerful. It allows us to create new services easily with only the methods and behaviors that we need, and inject services only where they are needed. This code is also completely isomorphic, so we can run it on the server or in the client!

Feathers + Vue.js

Vue.js allows you to easily build *reactive components for modern web Interfaces*, which makes it ideal for building real-time applications with Feathers. Let Vue.js take care of the client and let Feathers client do the heavy lifting for communicating with your server and managing your data.

In this guide we will create a plain Vue.js web front-end for the chat API built in the Your First App section.

If you haven't done so you'll want to go through that tutorial or you can find a working example here.

Setting up the HTML page

The Vue.js and Feathers client modules can be loaded individually via npm through a module loader like Webpack. To get up and running more quickly for this guide though, let's use the following HTML page as `public/chat.html`:

```
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=
1, user-scalable=0"/>
  <title>Feathers Chat</title>
  <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/public
/base.css">
  <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.1.0/public
/chat.css">
</head>
<body>


<chat-app v-if="user.authenticated">
  <user-list></user-list>
  <message-list>
    <compose-message></compose-message>
  </message-list>
</chat-app>


<template id="chat-app-template">
  <div id="app" class="flex flex-column">
    <header class="title-bar flex flex-row flex-center">
```

```
      <div class="title-wrapper block center-element">
        <img class="logo" src="http://feathersjs.com/img/feathers-logo-wide.png" alt="
Feathers Logo">
        <span class="title">Chat</span>
      </div>
    </header>
    <div class="flex flex-row flex-1 clear">

      <!-- Slots/transclusion (Angular). See http://vuejs.org/guide/components.html#Si
ngle-Slot -->
      <slot></slot>

    </div>
  </div>
</template>


<template id="user-list-template">
  <aside class="sidebar col col-3 flex flex-column flex-space-between">
    <header class="flex flex-row flex-center">
      <h4 class="font-300 text-center"><span class="font-600 online-count" v-cloak>{{
users.length }}</span> users</h4>
    </header>
    <ul class="flex flex-column flex-1 list-unstyled user-list">
      <li v-for="user in users" track-by="$index" v-cloak>
        <a class="block relative" href="#">
          <img :src="user.avatar || dummyUser.avatar" alt="" class="avatar">
          <span class="absolute username">{{ user.email || dummyUser.email }}</span>
        </a>
      </li>
    </ul>
    <footer class="flex flex-row flex-center">
      <a href="#" class="logout button button-primary" @click="logout">Sign Out</a>
    </footer>
  </aside>
</template>


<template id="message-list-template">
  <div class="flex flex-column col col-9">
    <main class="chat flex flex-column flex-1 clear">
      <div class="message flex flex-row" v-for="message in messages" track-by="$index"
v-cloak>
        <message :message=message></message>
      </div>
    </main>

    <slot></slot>

  </div>
</template>
```

```
<template id="message-template">
  <img :src="message.sentBy.avatar || placeholder" alt="{{ message.sentBy.email }}" cl
ass="avatar">
  <div class="message-wrapper">
    <p class="message-header">
      <span class="username font-600">{{ message.email }}</span>
      <span class="sent-date font-300">{{ message.createdAt | moment }}</span>
    </p>
    <p class="message-content font-300">{{ message.text }}</p>
  </div>
  </div>
</template>


<template id="compose-message-template">
  <form class="flex flex-row flex-space-between" id="send-message" v-on:submit.prevent>

    <input type="text" name="text" class="flex flex-1" v-model="newMessage">
    <button class="button-primary" type="submit" @click="addMessage">Send</button>
  </form>
</template>


<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.21/vue.js"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/moment.js/2.12.0/moment.js"></script>
<script src="//cdn.rawgit.com/feathersjs/feathers-client/v1.0.0/dist/feathers.js"></sc
ript>
<script src="/socket.io/socket.io.js"></script>
<script src="app.js"></script>
</body>
</html>
```

> **ProTip:** This setup is not recommended for a production environment. Instead, it is recommended that you use Webpack to compile your client-app or Vue-CLI which can scaffold an app for you, complete with build tools etc. You then just need to use the Feathers client modules in the client chapter.

Application Bootstrap

All of the code examples that follow should be appended to the `app.js` file, which the HTML file already loads. The first step is to add a placeholder user in the event that a user image or other user information doesn't exist:

```
// A placeholder image if the user does not have one
const PLACEHOLDER = 'https://placeimg.com/60/60/people';

// An anonymous user if the message does not have that information
const dummyUser = {
  avatar: PLACEHOLDER,
  email: 'Anonymous'
}
```

Now we can add this code to set up the Socket.io connection and the Feathers app:

```
// Establish a Socket.io connection
const socket = io()

// Initialize our Feathers client application through Socket.io
// with hooks and authentication.
const app = feathers()
  .configure(feathers.socketio(socket))
  .configure(feathers.hooks())
  // Use localStorage to store our login token
  .configure(feathers.authentication({
    storage: window.localStorage
  }))

  // Get the Feathers services we want to use
const userService = app.service('users');
const messageService = app.service('messages');
```

Initialising our Vue instance

We need to now initialize our Vue instance and handle authentication. If the user is authenticated, we set `this.user.authenticate = true` and the `v-if` directive on the `<chat-app>` component will evaluate to true, thus revealing the app.

```
var vm = new Vue({
  el: 'body',
  data: {
    user: {
      authenticated: false
    }
  },

  created() {
    app.authenticate().then(() => {
      this.user.authenticated = true
    })
    // On errors we just redirect back to the login page
    .catch(error => {
      if (error.code === 401) window.location.href = '/login.html'
    });
  }
})
```

Adding components

In `chat.html`, we added some custom components and their corresponding templates, all with unique `id` attributes. If you are unsure of the usage of `<slots>`, please check the Vue Documentation on the subject. For those that are familiar with Angular, Vue calls this *content distribution*, otherwise known as *transclusion* in Angular.

As you can see in the below markup, within `chat.html` we have a main parent component and some nested components. Also note that within the `<message-list>` template, we loop over a `message` component and bind `message` to a `message` prop which gets passed into the `message` component.

```
<chat-app v-if="user.authenticated">
  <user-list></user-list>
  <message-list>
    <compose-message></compose-message>
  </message-list>
</chat-app>
...
...
...
<div class="message flex flex-row" v-for="message in messages" track-by="$index" v-clo
ak>
    <message :message=message></message>
</div>
```

Each of your components will use the Feathers application we initialized above.

The first component is a wrapper for our app. By checking if the user is authenticated here, we ensure that the user doesn't see our page for a split-second before the JavaScript kicks in. Other than that, this component doesn't do a whole lot but binds to its template in `chat.html`.

```
Vue.component('chat-app', {
  template: '#chat-app-template'
})
```

We then have a `<user-list>` component which fetches the users from the `users` Feathers service once the component is ready. We also listen for events coming from the server to know when a new user has been created and we add them to the `users` array (`this.users.push(user)`).

The only method within this component is a `logout` method which will redirect the user back to the index page after successfully logging-out.

```
Vue.component('user-list', {
  template: '#user-list-template',

  data() {
    return {
      dummyUser: dummyUser,
      users: []
    }
  },

  ready() {
    // Find all users
    userService.find().then(page => {
      this.users = page.data
    })

    // We will also see when new users get created in real-time
    userService.on('created', user => {
      this.users.push(user)
    })
  },

  methods: {
    logout() {
      app.logout().then(() => {
        vm.user.authenticated = false
        window.location.href = '/index.html'
      })
    }
  }
})
```

The `<message-list>` component is responsible for getting the messages from the Feathers server and much like the `<user-list>` component, whenever a new message has been created on the backend, it's sent to the client via websockets and pushed on to the `messages` array.

We then have a method that will scroll this view to the bottom after we've added a new message to the view.

```
Vue.component('message-list', {
  template: '#message-list-template',

  data () {
    return {
      placeholder: PLACEHOLDER,
      messages: []
    }
  },

  ready () {
    // Find the latest 10 messages. They will come with the newest first
    // which is why we have to reverse before adding them
    messageService.find({
      query: {
        $sort: {createdAt: -1},
        $limit: 25
      }
    }).then(page => {
      page.data.reverse()
      this.messages = page.data
      this.scrollToBottom()
    })

    // Listen to created events and add the new message in real-time
    messageService.on('created', message => {
      this.messages.push(message)
      this.newMessage = ''
      this.scrollToBottom()
    })
  },

  methods: {
    scrollToBottom: () => {
      vm.$nextTick(() => {
        const node = vm.$el.getElementsByClassName('chat')[0]
        node.scrollTop = node.scrollHeight
      })
    }
  }
})
```

Within the `<message-list>` component you'll notice that we have a `<message>` component. Its job is to simply render each message correctly. As the `<message-list>` component loops over each message, it passes the current message into the `<message>` component as a prop and we use a Vue filter to format the date (refer to the `message-list-template` template.

```
Vue.component('message', {
  props: ['message', 'index'],
  template: '#message-template',
  filters: {
    moment: date => {
      return moment(date).format('MMM Do, hh:mm:ss')
    }
  }
})
```

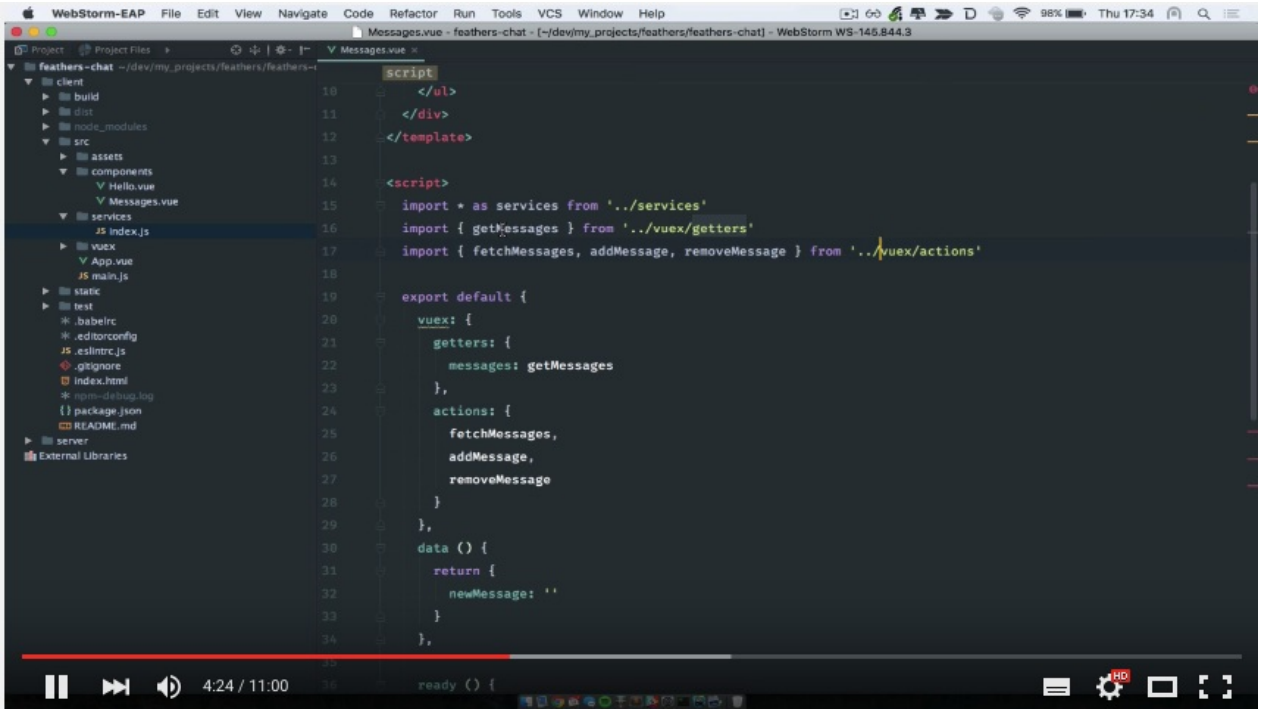
The `<compose-message>` component shows a form that when submitted creates a new message on the Feathers `messages` service and clears out the input field.

```
Vue.component('compose-message', {
  template: '#compose-message-template',

  data () {
    return {
      newMessage: ''
    }
  },

  methods: {
    addMessage () {
      // Create a new message and then clear the input field
      messageService.create({text: this.newMessage}).then(this.newMessage = '')
    }
  }
})
```

Further study

The Vue documentation is a fantastic source of reference. As your app grows, you may become interested in introducing Vuex, a state management library for Vue that's based heavily on Redux.

Below is an introduction to using Vuex with Vue and Feathers on the backend.



> Mad ♥ to Niall O'Brien for putting together the video.

Feathers + iOS

We haven't had time to publish this guide yet. If you feel like you could contribute one feel free to open a pull request.

There is an official iOS Feathers client in progress. If you feel like helping out head over to the feathers-ios repo.

Feathers + Android

We haven't had time to publish this guide yet. If you feel like you could contribute one feel free to open a pull request.

An official Feathers client does not exist yet. We have set up a placeholder for it so if you feel like contributing head over to the feathers-android repo.

Middleware

In the previous chapter we learned about how Hooks are used as data-oriented middleware for services.

A slightly different kind of middleware is one of the key concepts behind Express and since Feathers just extends Express, it can use all existing Express middleware and features.

In this chapter we will look at how services and Express middleware play together and how routing and versioning works for Feathers APIs.

Express middleware

In Express middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next` . How this middleware plays with Feathers services is outlined below.

Rendering views

While services primarily provide APIs for a client side application to use, they also play well with rendering views on the server with Express. For more details, please refer to the Using a View Engine guide.

Custom service middleware

Custom Express middleware that only should run before or after a specific service can be passed to `app.use` in the order it should run:

```
const todoService = {
  get(id) {
    return Promise.resolve({
      id,
      description: `You have to do ${id}!`
    });
  }
};

app.use('/todos', ensureAuthenticated, logRequest, todoService, updateData);
```

Middleware that runs after the service will have `res.data` available which is the data returned by the service. For example `updateData` could look like this:

```
function updateData(req, res, next) {
  res.data.updateData = true;
  next();
}
```

Keep in mind that shared authentication (between REST and websockets) should use a service based approach as described in the authentication guide.

Information about how to use a custom formatter (e.g. to send something other than JSON) can be found in the REST provider chapter.

Setting service parameters

All middleware registered after the REST provider has been configured will have access to the `req.feathers` object to set properties on the service method `params`:

```
app.configure(rest())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({extended: true}))
  .use(function(req, res, next) {
    req.feathers.fromMiddleware = 'Hello world';
    next();
  });

app.use('/todos', {
  get(id, params) {
    console.log(params.provider); // -> 'rest'
    console.log(params.fromMiddleware); // -> 'Hello world'

    return Promise.resolve({
      id, params,
      description: `You have to do ${id}!`
    });
  }
});
```

We recommend not setting `req.feathers = something` directly since it may already contain information that other Feathers plugins rely on. Adding individual properties or using `Object.assign(req.feathers, something)` is the more reliable option.

> **ProTip:** Although it may be convenient to set `req.feathers.req = req;` to have access to the request object in the service, we recommend keeping your services as provider independent as possible. There usually is a way to pre-process your data in a middleware so that the service does not need to know about the HTTP request or response.

Routing and Versioning

In this chapter we will look at custom URL routes and different ways for versioning a Feathers API.

Nested routes

A quite common question is how to provide nested routes for Feathers services. In general, Feathers does not know about associations between your services. Services are usually connected by their ids so any nested route can also be expressed by query parameters. For example if you have a user service and would like to get all todos (assuming the associated user id is stored in each todo), for that user the url would be `/todos?user_id=<userid>`. This approach also makes it easier to use by non REST providers like websockets and any other protocols Feathers might support in the future.

You can however add Express style parameters to your routes when you register a service (for additional information also see the REST chapter). This will then be set in the `params` object in each service call. For example a `/users/:user_id/todos` route can be provided like this:

```
app.use('/users/:user_id/todos', {
  find: function(params, callback) {
    // params.user_id == current user id
    // e.g. 1234 for /users/1234/todos
  },
  create: function(data, params, callback) {
    data.user_id = params.user_id;
    // store the data
  }
});
```

> **ProTip:** This route should be registered after the `/users` service.

To make the user id part of `params.query` we can use a before hook:

```
app.service('users/:user_id/todos').before({
  find: function(hook) {
    // Only do the mapping for the REST provider
    if(hook.params.provider === 'rest') {
      hook.params.query.user_id = hook.params.user_id;
    }
  }
});
```

Now all `GET /users/<id>/todos` requests will make a `find` query limited to the given user id.

> **ProTip:** Think of Feathers services as their own router that can only be used directly on an application. Services can *not* be used with instances of Express router (`feathers.Router`).

It is important to keep in mind that those routes are only possible for the REST API of a service. The actual service name is still `users/:user_id/todos` . This means that Socket.io and Primus connections need to provide the parameter in their query. To be able to use those route parameters both, in Socket.io and REST you have to add a hook that maps those parameters to the query like this:

```
app.service('users/:user_id/todos').before(function(hook) {
  if(hook.params.userId) {
    hook.params.query.user_id = hook.params.user_id;
  }
});
```

Then it can be used via websockets like this:

```
// Using the socket directly
socket.send('users/:user_id/todos::find', { user_id: 1234 }, function(error, todos) {}
);

// Or with a feathers client
const feathers = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');

const socket = io();
const app = feathers().configure(socketio(socket));

app.service('users/:user_id/todos').find({
  query: { user_id: 1234 }
}).then(todos => console.log('Todos for user', todos));
```

Versioning

It is a common practice to provide different version endpoints like `/v1/todos` for an API that evolves over time.

As routes

The most straightforward way to do this in Feathers is to simply register services with the version in their path:

```
app.use('/api/v1/todos', myService);
```

This setup is useful if you want to be able to access services from other versions of your API inside hooks. However, all versions will have to use share the same middleware and plugins.

As sub-applications

Another way is to use entirely separate sub-apps on versioned paths in a parent application. For example with the following application in the `v1/` folder:

```
// v1/todos.js
module.exports = {
  get(id) {
    return Promise.resolve({
      id,
      description: `You have to do ${id}!`
    });
  }
}

// v1/app1.js
const feathers = require('feathers');
const todoService = require('./todos');

const app = feathers().use('/todos', todoService);

module.exports = app;
```

And a different application in the `/v2` folder:

```
// v2/todos.js
module.exports = {
  get(id) {
    return Promise.resolve({
      id,
      description: `v2 todo for ${id}!`
    });
  }
}

// v2/app1.js
const feathers = require('feathers');
const todoService = require('./todos');

const app = feathers().use('/todos', todoService);

module.exports = app;
```

Both applications can be versioned in a top-level `app.js` with:

```
const feathers = require('feathers');
const v1app = require('./v1/app');
const v2app = require('./v2/app');

const app = feathers()
  .use('/v1', v1app)
  .use('/v2', v2app);
```

Now `/v1/todos/dishes` and `/v2/todos/dishes` will show a different response. For websocket services, the path to listen and send events on will now be `v1/todos` and `v2/todos`. The advantage to doing versioning as shown above (as opposed to doing it directly on the service) is that you will be able to use custom plugins and a custom configuration in each version of your API.

> **ProTip:** Currently, using `app.service('v1/todos')` does not work. You will have to use the imported applications like `v1app.service('todos')` or `v2app.service('todos')`

Virtual Hosting, HTTPS and sub-apps

Virtual hosts, HTTPS and sub-applications work the same exact same as in Express. If an app is not started via `app.listen` however, you will have to call `app.setup(server)` manually for Feathers to do its setup thing. See the Core API documentation for more details on those methods.

Sub-Apps

As already described in the routing chapter, sub-apps make it easy to provide different versions for an API. Currently, when using the Socket.io and Primus real-time providers providers, `app.setup` will be called automatically, however, with only the REST provider or when using plain Express in the parent application you will have to call the sub-apps `setup` yourself:

```
const express = require('express');
const feathers = require('feathers');
const api = feathers().use('/service', myService);

const mainApp = express().use('/api/v1', api);

const server = mainApp.listen(3030);

// Now call setup on the Feathers app with the server
api.setup(server);
```

HTTPS

With your Feathers application initialized it is easy to set up an HTTPS REST and SocketIO server:

```
const fs = require('fs');
const https  = require('https');

app.configure(socketio()).use('/todos', todoService);

const server = https.createServer({
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem')
}, app).listen(443);

// Call app.setup to initialize all services and SocketIO
app.setup(server);
```

Virtual Hosts

You can use the vhost middleware to run your Feathers app on a virtual host:

```
const vhost = require('vhost');

app.use('/todos', todoService);

const host = feathers().use(vhost('foo.com', app));
const server = host.listen(8080);

// Here we need to call app.setup because .listen on our virtal hosted
// app is never called
app.setup(server);
```

Authentication

At some point, you are probably going to put information in your databases that you want to keep private. You'll need to implement an *authentication* scheme to identify your users, and *authorization* to control access to resources. The feathers-authentication plugin makes it easy to add token-based auth to your app.

Cookie based and token based authentication are the two most common methods of putting server side authentication into practice. Cookie based authentication relies on server side cookies to remember the user. Token based authentication requires an encrypted auth token with each request. While cookie based authentication is the most common, token based authentication offers several advantages for modern web apps. Two primary advantages are security and scalability.

The Auth0 blog has a great article on the advantages that token authentication offers.

Usage

Server Side

If you are using the default options, setting up JSON Web Token auth for your Feathers app is as simple as the example below. This example would typically be used alongside a User Service to keep track of the users within your app.

```
let app = feathers()
  .configure(rest())
  .configure(socketio())
  .configure(hooks())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  .configure(authentication());
```

> **ProTip:** You must set up the `body-parser`, `feathers-hooks` and possibly `cors` modules before setting up `feathers-authentication`. The Feathers CLI generator does this for you already.

Options

The options passed to the authentication plugin are wrapped in an object with the following top level keys:

- `local` (default: see options) [optional] - The local auth provider config. By default this is included in a Feathers app. If set to `false` it will not be initialized.
- `token` (default: see options) [optional] - The JWT auth provider config. By default this is included in a Feathers app even if you don't pass in any options. You can't disable it like `local` auth.
- `<oauth-provider>` (default: see options) [optional] - A lowercased oauth provider name (ie. `facebook` or `github`)
- `successRedirect` (default: '/auth/success') [optional] - The endpoint to redirect to after successful authentication or signup. Only used for requests not over Ajax or sockets. Can be set to `false` to disable redirects.
- `failureRedirect` (default: '/auth/failure') [optional] - The endpoint to redirect to for a failed authentication or signup. Only used for requests not over Ajax or sockets. Can be set to `false` to disable redirects.
- `shouldSetupSuccessRoute` (default: `true`) [optional] - Can be set to `false` to disable setting up the default success redirect route handler. **Required** if you want to render your own custom page on auth success.
- `shouldSetupFailureRoute` (default: `true`) [optional] - Can be set to `false` to disable setting up the default failure redirect route handler. **Required** if you want to render your own custom page on auth failure.
- `idField` (default: '_id') [optional] - the id field for you user's id. This is use by many of the authorization hooks.
- `localEndpoint` (default: '/auth/local') [optional] - The local authentication endpoint used to create new tokens using local auth
- `userEndpoint` (default: '/users') [optional] - The user service endpoint
- `tokenEndpoint` (default: '/auth/token') [optional] - The JWT auth service endpoint
- `header` (default: 'authorization') [optional] - The header field to check for the token. **This is case sensitive**.
- `cookie` (default: see options) [optional] - The cookie options used when sending the JWT in a cookie for OAuth or plain form posts. You can disable sending the cookie by setting this to `false` .

Cookie Options

All the options get passed to Express' `res.cookie` function. See the Express docs for more detail.

- `name` (default: 'feathers-jwt') [optional] - The cookie name. **This is case sensitive**.
- `httpOnly` (default: 'false') [optional] - Prevents JavaScript from accessing the cookie on the client. Should be set to `true` if you are not using OAuth or Form Posts for authentication.
- `secure` (default: 'true' in production) [optional] - Marks the cookie to be used with

HTTPS only.

- `expires` (default: 30 seconds from current time) [optional] - The time when the cookie should expire. Must be a valid `Date` object.

Example Configuration

Below is an example config providing some common override options:

```
{
  userEndpoint: '/api/users',
  tokenEndpoint: '/api/tokens',
  idField: 'id',
  local: {
    usernameField: 'username'
  },
  token: {
    secret: 'shhh secrets'
  },
  facebook: {
    strategy: FacebookStrategy,
    'clientID': 'your facebook client id',
    'clientSecret': 'your facebook client secret',
    'permissions': {
      authType: 'rerequest',
      'scope': ['public_profile', 'email']
    }
  }
}
```

Client Side

If you are using the default options setting up Feathers authentication client side is a breeze.

```
// Set up socket.io
const host = 'http://localhost:3030';
let socket = io(host);

// Set up Feathers client side
let app = feathers()
  .configure(feathers.socketio(socket))
  .configure(hooks())
  .configure(authentication());

// Authenticate. Normally you'd grab these from a login form rather than hard-coding t
hem
app.authenticate({
  type: 'local',
  'email': 'admin@feathersjs.com',
  'password': 'admin'
}).then(function(result){
  console.log('Authenticated!', result);
}).catch(function(error){
  console.error('Error authenticating!', error);
});
```

To store the token in localStorage and use that initially, use this:

```
// Set up socket.io
const host = 'http://localhost:3030';
let socket = io(host);

// Set up Feathers client side
let app = feathers()
  .configure(feathers.socketio(socket))
  .configure(hooks())
  .configure(authentication({
    storage: window.localStorage
  }));

function showApplication() {
  // Authentication was successfull, you can now render the application here
}

function showLogin() {
  // Show the login screen, which calls the following at one point:
  app.authenticate({
    type: 'local',
    'email': 'admin@feathersjs.com',
    'password': 'admin'
  }).then(showApplication).catch(function(error){
    console.error('Error authenticating!', error);
  });
}

// First try to authenticate with the token from localStorage.
// If successful call `showApplication`, if not call `showLogin`
app.authenticate().then(showApplication, showLogin);
```

> **ProTip:** You can also use Primus or a handful of Ajax providers instead of Socket.io. Check out the Feathers client authentication section for more detail.

> **ProTip:** You can only have one provider client side per "app". For example if you want to use both Socket.io and a REST Ajax provider then you need to configure two apps.

Options

- `tokenEndpoint` (default: '/auth/token') [optional] - The JWT auth service endpoint.
- `localEndpoint` (default: '/auth/local') [optional] - The local auth service endpoint
- `header` (default: 'authorization') [optional] - The header field to set the token. **This is case sensitive**.
- `cookie` (default: 'feathers-jwt') [optional] - The cookie field to check for the token. **This is case sensitive**.
- `tokenKey` (default: 'feathers-jwt') [optional] - The key to use to store the JWT in localStorage. **This is case sensitive**.

Example Configuration

Below is an example config providing some common override options:

```
app.configure(authentication({
  tokenEndpoint: '/token',
  localEndpoint: '/login',
  header: 'X-Authorization',
  cookie: 'app-token',
  tokenKey: 'app-token'
}));
```

How does it work?

Regardless of whether you use OAuth, a token, or email + password to authenticate (even if you don't use the Feathers client), after successful login the `feathers-authentication` plugin gives back a signed JSON web token containing the user `id` as the payload and the user object associated with that id.

```
// successful auth response
{
  token: 'your encrypted json web token',
  data: {
    email: 'hulk@hogan.net'
  }
}
```

Authentication Over REST

Creating Tokens

To create a new token with an HTTP request make a `POST` request to the local authentication endpoint with a valid set of user credentials. By default this endpoint is `/auth/local`. If you have not already set up a User Service you should do that first. See the README for a complete example.

The following cURL request can be used to authenticate a user from the command line using the default options. If the authentication request was successful you will receive a response back with your token.

```
 # Assuming a user exists with the following credentials
$ curl -X POST \
-H 'Content-Type: application/json' \
-d '{ "email": "hulk@hogan.net", "password": "bandana" }' \
http://127.0.0.1:3000
```

> **ProTip** These defaults can all be overridden as described in the server-side config options and local auth config options.

Using Tokens

For REST the token needs to be passed with each request. Therefore if you did `.configure(rest())` in your Feathers app, the auth plugin also includes a special middleware that ensures that a token, if sent, is available on the Feathers `params` object that is passed to services and hooks by setting it on `req.feathers.token`.

> **ProTip:** The `req.feathers` object is special because its attributes are made available inside Feathers hooks on the `hook.params` object.

This middleware uses graceful fall-back to check for a token in order from most secure/appropriate to least secure:

1. Authorization header (recommended)
2. Cookie
3. The request body
4. The query string (not recommended but supported)

So you can send your token using any of those methods. Using the `authorization` header it should look like this:

```
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IklseW
EgRmFkZWV2IiwiYWRtaW4iOnRydWV9.YiG9JdVVm6Pvpqj8jDT5bMxsm0gwoQTOaZOLI-QfSNc
```

> **ProTip:** The `Bearer` part can be omitted and the case doesn't matter.

> **ProTip:** You can use a custom header name for your token by passing a `header` option as described above.

Authentication Over Sockets

After a socket is connected an `authenticate` event needs to be emitted from the client to initiate the socket authentication. The data passed with it can either be an email and password, a JWT or OAuth access tokens. After successful authentication an

`authenticated` event is emitted from the server and just like with REST you get back a JWT and the current user. From then on you are now using an authenticated socket until that socket is disconnected, the token expires, or you log out.

What Happens During Authentication?

Regardless of the mechanism, the credentials used to authenticate, and the transport, the high level order of execution is as follows:

1. The credentials passed in are verified or you go through a series of OAuth redirects and are verified by the OAuth provider.
2. Once credentials are verified the user associated with those credentials is looked up and if a user does not exist they are created by calling your `user` service.
3. The user id is encrypted into the JWT by an asymmetric hashing function using your `secret` inside the `token` service.
4. The user is added to the response *after* a token is created using the `populateUser()` hook and the new token and user are returned to the client.

Authorizing Future Requests

Regardless of the protocol, once a valid auth token as been returned to the client, for any subsequent request the token (if present) is normalized and the verifyToken() hook should be called by you prior to any restricted service methods.

This hook decrypts the token found in `hook.params.token`. After the JWT is decrypted, the populateUser() hook should be called. This is used to look up the user by id in the database and attach them to `hook.params.user` so that any other hooks in the chain can reference the current user, for example the requireAuth() hook.

For more information on refer to the Authorization chapter.

What's next?

Adding authentication allows you to know **who** users are. Now you'll want to specify **what** they can do. This is done using authorization hooks. Learn more about it in the Authorization section or dive deeper into each of the individual authentication schemes:

- Setting Up Local Auth (username and password)
- Setting Up Token Auth (JWT)
- Setting Up OAuth1 (Twitter)
- Setting Up OAuth2 (Facebook, LinkedIn, etc.)
- Setting Up 2 Factor

- Auth with Feathers client

Token Authentication

The JSON web token (JWT) auth strategy is enabled by default with Feathers authentication. Currently it cannot be disabled because right now session or cookie based authentication is not supported and token based auth is better.

Usage

Server Side

This is what a typical sever setup looks like:

```
app.configure(authentication({
  token: {
    secret: 'my-secret'
  }
}));
```

Normally the only option you might want to pass when registering the `feathers-authentication` module sever side is your token `secret`. That's all.

> **ProTip:** If you don't pass a token `secret` a secure one will be randomly generated for you each time your app starts.

Token Service Specific Options

All of the top level authentication options are passed to the token authentication service. If you need to customize your token specific configuration further you can use these options:

- `secret` (**required**) (default: a strong auto generated one) - Your secret used to sign JWT's. If this gets compromised you need to rotate it immediately!
- `payload` (default: '[]') [optional] - An array of fields from your user object that should be included in the JWT payload.
- `passwordField` (default: 'password') [optional] - The database field containing the password on the user service.
- `issuer` (default: 'feathers') [optional] - The JWT issuer field
- `algorithm` (default: 'HS256') [optional] - The accepted JWT hash algorithm
- `expiresIn` (default: '1d') [optional] - The time a token is valid for

You can view additional available JWT signing options in the node-jsonwebtoken repo.

> **ProTip:** JWT payloads can be decoded on the client. Do not store sensitive information in the JWT payload. If you must then it should be encrypted before the JWT is signed in the token service using your secret.

Client Side

Using Feathers Client

The Feathers authentication module has a client side component that makes it very easy for you to add authentication to your app. It can be used in the browser, NodeJS and React Native. Refer to the feathers client authentication section for more detail.

Other Clients

Of course, if you don't want to use the feathers authentication client you can also just use vanilla sockets or ajax. It's a bit more work but honestly, not much more. We have some examples here.

Setting up Local Authentication

The local auth strategy makes it easy to add *username* and *password* authentication for your app. Once a user has successfully authenticated they get a JSON Web Token (JWT) in return to user for future authentication with your feathers app.

Usage

Server Side

This is what a typical server setup looks like:

```
app.configure(authentication());
```

Normally you don't need to pass any options for local auth when registering the `feathers-authentication` module server side.

Local Service Specific Options

All of the top level authentication options are passed to the local authentication service. If you need to customize your local specific configuration further you can use these options:

- `usernameField` (default: 'email') [optional] - The database field on the user service you want to use as the username.
- `passwordField` (default: 'password') [optional] - The database field containing the password on the user service.
- `session` (default: 'false') [optional] - Whether the local Passport auth strategy should use sessions.

Client Side

Using Feathers Client

The Feathers authentication module has a client side component that makes it very easy for you to add authentication to your app. It can be used in the browser, NodeJS and React Native. Refer to the feathers client authentication section for more detail.

Other Clients

Of course, if you don't want to use the feathers authentication client you can also just use vanilla sockets or ajax. It's a bit more work but honestly, not much more. We have some examples here.

OAuth1 & OAuth1a Authentication

Coming Soon!

Because Twitter doesn't support OAuth2 for authentication we need to use OAuth1. OAuth1 requires a session, which kind of defeats the purpose of using JWT based authentication so we intentionally chose to leave OAuth1 out of the Feathers 2 release.

We are planning on adding it over the coming months. If you'd like to help out with this effort please head over to the feathers-authentication repo.

OAuth2 Authentication

The OAuth2 strategy is only enabled if you pass in a top level config object other than `local` and `token`. Once a user has successfully authenticated, if they aren't in your database they get created, and a JSON Web Token (JWT) along with the current user are returned to use for future authentication with your feathers app.

Because OAuth relies on a series of redirects we need to get the user their JWT somehow without putting it in the query string, which is potentially insecure.

To solve this problem, Feathers redirects to a configurable `successRedirect` route and puts the user's JWT token in a cookie with the default name `feathers-jwt`. Your client side app can then parse the cookie for the token and use it to further authenticate. This is exactly what the client side component of the Feathers authentication module does for you automatically.

> **ProTip:** Many other frameworks either use sessions for auth or if using a token + OAuth just shove the token in the query string. This is potentially insecure as an intermediary could be logging these URLs with the token in them, even over HTTPS.

Usage

Server Side

This is what a typical server setup looks like:

```
var FacebookStrategy = require('passport-facebook').Strategy;

app.configure(authentication({
  facebook: {
    strategy: FacebookStrategy,
    'clientID': 'your-facebook-client-id',
    'clientSecret': 'your-facebook-client-secret',
    'permissions': {
      authType: 'rerequest',
      'scope': ['public_profile', 'email']
    }
  }
}));
```

OAuth2 Service Specific Options

All of the top level authentication options are passed to an OAuth2 authentication service and the passport strategies. If you need to customize your OAuth2 specific configuration you can use these options:

- `clientID` (**required**) - Your OAuth2 clientID
- `clientSecret` (**required**) - Your OAuth2 clientSecret
- `permissions` (**required**) - An object with the permissions you are requesting. See your passport provider docs for details. `state` is set to `true` and `session` is set to `false` by default.
- `strategy` (**required**) - The Passport OAuth strategy for your oauth provider (ie. passport-facebook)
- `tokenStrategy` [optional] - The Passport OAuth token strategy if you want to support mobile authentication without a browser.
- `passReqToCallback` (default: true) [optional] - A Passport option to pass the request object to the oauth callback handler.
- `endPoint` (default: '/auth/') [optional] - This is the endpoint that your OAuth2 provider service is located at. For example, `/auth/facebook`.
- `callbackSuffix` (default: 'callback') [optional] - This gets added to the provider endpoint to form the callback url. For example `/auth/facebook/callback`.

> **ProTip:** Feathers just uses Passport authentication strategies so you can pass any strategy specific options in your provider config and it will be automatically passed on to the strategy you are using.

Client Side

Typically the only thing you need to do client-side is have a link or a redirect to your authentication provider's endpoint (ie. `/auth/facebook`).

The Feathers server will handle all the OAuth2 redirects, verification and will do a final redirect to the `successRedirect` endpoint, which by default is `/auth/success`. From there you would load your client side app and read the JWT token from the cookie and use it to authenticate. Using the Feathers client this all handled for you. It parses the JWT out of the cookie and places it in local storage. You simply need to call `app.authenticate()`.

Using Feathers Client

The Feathers authentication module has a client side component that makes it very easy for you to add authentication to your app. It can be used in the browser, NodeJS and React Native. Refer to the feathers client authentication section for more detail.

Other Clients

Of course, if you don't want to use the feathers authentication client you can also just use vanilla sockets or ajax. It's a bit more work but honestly, not much more. We have some examples here.

Two Factor Authentication

Coming Soon!

If you'd like to help out with this effort please head over to the feathers-authentication repo.

Password Management

It's common in every app that you need to provide the ability for users to reset their password and/or confirm their account. Currently you can implement this functionality yourself but we have some updates coming soon that will make this easier.

You can track the progress here.

Authenticating With Feathers Client

The Feathers authentication module has a client side component that makes it very easy for you to authenticate with a Feathers API in the browser, from another NodeJS server, or a React Native app.

API

Feathers authentication adds a few methods to a client side Feathers app. They are as follows:

authenticate

`app.authenticate()` attempts to authenticate with the server using the data you passed it. If you don't provide any options it will attempt to authenticate using a token stored in memory or in your `storage` engine. It returns a promise.

Options

- `type` (optional) - Either `local` or `token`.

Then you pass whatever other fields you need to send to authenticate. See below for examples.

logout

`app.logout()` clears the token and the user from your local store. In the future it will call the server to invalidate the token.

user

`app.get('user')` is a convenience method to get the user. Currently it only pulls from your local store. In the future it may fall back to fetching from the server if the user isn't available in the client.

token

`app.get('token')` is a convenience method to get your access token. Currently it only pulls from your local store.

Usage

> **ProTip:** All the client examples below demonstrate how it works in the browser without a module loader. You can also use Feathers client with a module loader like Webpack, browserify, React Native packager and also in NodeJS. See the Feathers client section for more detail.

> **ProTip:** If you pass a `storage` engine when configuring `feathers-authentication` it will persist your token there, otherwise it will just store it in memory. This is highly recommended to use a storage engine as it allows you to refresh without having to authenticate.

REST

On the client

```
<script src="//code.jquery.com/jquery-1.12.0.min.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.
js"></script>
<script type="text/javascript">
  var host = 'http://localhost:3030';

  // Set up Feathers client side
  var app = feathers()
    .configure(feathers.rest(host).jquery(jQuery))
    .configure(feathers.hooks())
    .configure(feathers.authentication({ storage: window.localStorage }));

  // Authenticate. Normally you'd grab these from a login form rather than hard coding
 them
  app.authenticate({
    type: 'local',
    'email': 'admin@feathersjs.com',
    'password': 'admin'
  }).then(function(result){
    console.log('Authenticated!', app.get('token'));
  }).catch(function(error){
    console.error('Error authenticating!', error);
  });
</script>
```

> **ProTip:** You can use other Ajax providers like Superagent or Fetch for REST on the client. Check out the Feathers client section for more info.

On the server

```
let app = feathers()
  .configure(rest())
  .configure(hooks())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  .configure(authentication());
```

Socket.io

On the client

```
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.
js"></script>
<script type="text/javascript">
  // Set up socket.io
  var host = 'http://localhost:3030';
  var socket = io(host);

  // Set up Feathers client side
  var app = feathers()
    .configure(feathers.socketio(socket))
    .configure(feathers.hooks())
    .configure(feathers.authentication({ storage: window.localStorage }));

  // Authenticating using a token instead
  app.authenticate({
    type: 'token',
    'token': 'your token'
  }).then(function(result){
    console.log('Authenticated!', app.get('token'));
  }).catch(function(error){
    console.error('Error authenticating!', error);
  });
</script>
```

On the server

```
let app = feathers()
  .configure(socketio())
  .configure(hooks())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  .configure(authentication());
```

Primus

On the client

```
<script type="text/javascript" src="/primus/primus.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/cor
e.min.js"></script>
<script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/feathers.
js"></script>
<script type="text/javascript">
  // Set up primus
  var host = 'http://localhost:3030';
  var primus = new Primus(host);

  // Set up Feathers client side
  var app = feathers()
    .configure(feathers.primus(primus))
    .configure(feathers.hooks())
    .configure(feathers.authentication({ storage: window.localStorage }));

  // Authenticate. Normally you'd grab these from a login form rather than hard-coding
 them
  app.authenticate({
    type: 'local',
    'email': 'admin@feathersjs.com',
    'password': 'admin'
  }).then(function(result){
    console.log('Authenticated!', app.get('token'));
  }).catch(function(error){
    console.error('Error authenticating!', error);
  });
</script>
```

On the server

```
let app = feathers()
  .configure(primus({ transformer: 'websockets' }))
  .configure(hooks())
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  .configure(authentication());
```

Now that you've setup authentication you'll probably want to start locking down your application. Head on over to the Authorization chapter to get started.

Authorization / Access Control

Once we know which user is logged in, we need to know which parts of the app they can access. This is called Authorization, and it's where hooks really come in handy.

Security

By default Feathers is pretty loose and is not locked down. This is to allow for rapid prototyping. Before you go to production there are a few things you should do.

Remove passwords from responses

Make sure that your user's passwords are not being sent to the client. There could very likely be a password on your `user` object. Read the section on bundled auth hooks to find out how to make sure passwords don't go out to the public.

> **ProTip:** Feathers authentication automatically removes the password field from the user object that is sent in the response to a successful login and if you used the generator we've already set up the hook to remove passwords on the user service for you.

Lock down restricted services

Make sure any restricted endpoints are actually locked down appropriately by adding the appropriate hooks to your services. Check out the bundled authentication hooks. They probably do most of what you need.

Filter socket events

Filter socket events so only authenticated and authorized users get can receive restricted ones.

To make this easy we've created a bunch of hooks to help you out. Their interface's are documented so if you are unsure how to use hooks check out the Hooks chapter. Below is an example of how you can create your own.

Creating a custom authorization hook

In the example below, only a user in the `feathers` group can delete messages.

```
// Create a hook that requires that a user is logged in.
var isFeathersUser = function(options = {}) {
  return function(hook) {
    // We can assume hook.params.user exists because the auth.restrictToAuthenticated()

    // hook is called before this and will throw an error if it doesn't
    if (hook.params.user.group !== 'feathers') {
      throw new Error('You must be a feathers user to do that.');
    }
  };
};

// Must be logged in to do anything a feathers user to delete a message.
app.service('messages').before({
  all: [
    auth.verifyToken(),
    auth.populateUser(),
    auth.restrictToAuthenticated()
  ],
  remove: [isFeathersUser()]
});
```

Authorization Hooks

These hooks come bundled with the Feathers authentication plugin.

Implementing authorization is not automatic, but is easy to set up with the included hooks. You can also create your own hooks to handle your app's custom business logic. For more information about hooks, refer to the chapter on hooks.

The `feathers-authentication` plug-in includes the following hooks to help with the authorization process. The most common scenario is that you simply want to restrict a service to only authenticated users. That can be done like so:

```
const hooks = require('feathers-authentication').hooks;

// Must be logged in do anything with messages.
app.service('messages').before({
  all: [
    hooks.verifyToken(),
    hooks.populateUser(),
    hooks.restrictToAuthenticated()
  ]
});
```

> **ProTip:** All bundled authorization hooks will automatically pull values from your `auth` config. You can override them explicitly by passing them to the hook.

hashPassword

The `hashPassword` hook will automatically hash the data coming in on the provided `passwordField`. It is intended to be used as a **before** hook on the user service for the `create`, `update`, or `patch` methods.

```
const hooks = require('feathers-authentication').hooks;

app.service('user').before({
  create: [
    hooks.hashPassword()
  ]
});
```

Options

- `passwordField` (default: 'password') [optional] - The field you use to denote the password on your user object.

verifyToken

The `verifyToken` hook will attempt to verify a token. If the token is missing or is invalid it returns an error. If the token is valid it adds the decrypted payload to `hook.params.payload` which contains the user id. It is intended to be used as a **before** hook on **any** of the service methods.

```
const hooks = require('feathers-authentication').hooks;

app.service('user').before({
  get: [
    hooks.verifyToken()
  ]
});
```

Options

- `secret` (default: the one from your config) [optional] - Your secret used to encrypt and decrypt JWT's on the server. If this gets compromised you need to rotate it immediately!
- `issuer` (default: 'feathers') [optional] - The JWT issuer field
- `algorithm` (default: 'HS512') [optional] - The accepted JWT hash algorithm
- `expiresIn` (default: '1d') [optional] - The time a token is valid for

You can view the all available options in the node-jsonwebtoken repo.

populateUser

The `populateUser` hook is for populating a user based on an id. It can be used on **any** service method as either a **before** or **after** hook. It is called internally after a token is created.

```
const hooks = require('feathers-authentication').hooks;

app.service('user').before({
  get: [
    hooks.populateUser()
  ]
});
```

Options

- `userEndpoint` (default: '/users') [optional] - The endpoint for the user service.
- `idField` (default: '_id') [optional] - The database field containing the user id.

restrictToAuthenticated

The `restrictToAuthenticated` hook throws an error if there isn't a logged-in user by checking for the `hook.params.user` object. It can be used on **any** service method and is intended to be used as a **before** hook. It doesn't take any arguments.

```
const hooks = require('feathers-authentication').hooks;

app.service('user').before({
  get: [
    hooks.restrictToAuthenticated()
  ]
});
```

queryWithCurrentUser

The `queryWithCurrentUser` **before** hook will automatically add the user's `id` as a parameter in the query. This is useful when you want to only return data, for example "messages", that were sent by the current user.

```
const hooks = require('feathers-authentication').hooks;

app.service('messages').before({
  find: [
    hooks.queryWithCurrentUser({ idField: 'id', as: 'sentBy' })
  ]
});
```

Options

- `idField` (default: '_id') [optional] - The id field on your user object.
- `as` (default: 'userId') [optional] - The id field for a user on the resource you are requesting.

When using this hook with the default options the `User._id` will be copied into `hook.params.query.userId`.

associateCurrentUser

The `associateCurrentUser` **before** hook is similar to the `queryWithCurrentUser`, but works on the incoming **data** instead of the **query** params. It's useful for automatically adding the userId to any resource being created. It can be used on `create`, `update`, or `patch` methods.

```
const hooks = require('feathers-authentication').hooks;

app.service('messages').before({
  create: [
    hooks.associateCurrentUser({ idField: 'id', as: 'sentBy' })
  ]
});
```

Options

- `idField` (default: '_id') [optional] - The id field on your user object.
- `as` (default: 'userId') [optional] - The id field for a user that you want to set on your resource.

restrictToOwner

`restrictToOwner` is meant to be used as a **before** hook. It only allows the user to retrieve resources that are owned by them. It will return a *Forbidden* error without the proper permissions. It can be used on `get`, `create`, `update`, `patch` or `remove` methods.

```
const hooks = require('feathers-authentication').hooks;

app.service('messages').before({
  remove: [
    hooks.restrictToOwner({ idField: 'id', ownerField: 'sentBy' })
  ]
});
```

Options

- `idField` (default: '_id') [optional] - The id field on your user object.
- `ownerField` (default: 'userId') [optional] - The id field for a user on your resource.

restrictToRoles

`restrictToRoles` is meant to be used as a **before** hook. It only allows the user to retrieve resources that are owned by them or protected by certain roles. It will return a *Forbidden* error without the proper permissions. It can be used on `all` methods when the **owner** option is set to 'false'. When the **owner** option is set to `true` the hook can only be used on `get`, `update`, `patch`, and `remove` service methods.

```
const hooks = require('feathers-authentication').hooks;

app.service('messages').before({
  remove: [
    hooks.restrictToRoles({
        roles: ['admin', 'super-admin'],
        fieldName: 'permissions',
        idField: 'id',
        ownerField: 'sentBy',
        owner: true
    })
  ]
});
```

Options

- `roles` (**required**) - An array of roles that a user must have at least one of in order to access the resource.
- `fieldName` (default: 'roles') [optional] - The field on your user object that denotes their roles.
- `idField` (default: '_id') [optional] - The id field on your user object.
- `ownerField` (default: 'userId') [optional] - The id field for a user on your resource.
- `owner` (default: 'false') [optional] - Denotes whether it should also allow owners regardless of their role (ie. the user has the role **or** is an owner).

Error Handling

By default Feathers just uses the default error handler that comes with Express. It's pretty basic so the feathers-errors module comes bundled with a more robust error handler that you can use in your app. This error handler is the one that is included in a generated Feathers app by default.

> **ProTip:** Because Feathers extends Express you can use any Express compatible error middleware with Feathers. In fact, the error handler bundled with `feathers-errors` is just a slightly customized one.

Many Feathers plugins (like the database adapters and authentication) already throw Feathers errors, which include their status codes. The default error handler sends a JSON representation of the error (without the stacktrace in production) or sends a default `404.html` or `500.html` error page when visited in the browser.

If you want to use your own custom error pages you can do with a custom HTML formatter like this:

```
const error = require('feathers-errors/handler');
const app = feathers();

// Just like Express your error middleware needs to be
// set up last in your middleware chain.
app.use(error({
    html: function(error, req, res, next) {
      // render your error view with the error object
      res.render('error', error);
    }
}))
```

> **ProTip:** If you want to have the response in json format be sure to set the `Accept` header in your request to `application/json` otherwise the default error handler will return HTML.

Options

The following options can be passed when creating a new localstorage service:

- `html` (Function|Object) [optional] - A custom formatter function or an object that contains the path to your custom html error pages.

> **ProTip:** `html` can also be set to `false` to disable html error pages altogether so that only JSON is returned.

Catching Global Server Side Errors

Promises swallow errors if you for get to add a `catch()` statement. Therefore, you should make sure that you **always** call `.catch()` on your promises. To catch uncaught errors at a global level you can add the code below to your top-most file.

```
process.on('unhandledRejection', (reason, p) => {
  console.log("Unhandled Rejection at: Promise ", p, " reason: ", reason);
});
```

Feathers Error Types

`feathers-errors` currently provides the following error types, all of which are instances of `FeathersError` :

> **ProTip:** All of the feathers plugins will automatically emit the appropriate Feathers errors for you.

- `BadRequest` : 400
- `NotAuthenticated` : 401
- `PaymentError` : 402
- `Forbidden` : 403
- `NotFound` : 404
- `MethodNotAllowed` : 405
- `NotAcceptable` : 406
- `Timeout` : 408
- `Conflict` : 409
- `Unprocessable` : 422
- `GeneralError` : 500
- `NotImplemented` : 501
- `Unavailable` : 503

FeathersError API

Feathers errors are pretty flexible. They contain the following fields:

- `type` - `FeathersError`

- `name` - The error name (ie. "BadRequest", "ValidationError", etc.)
- `message` - The error message string
- `code` - The HTTP status code
- `className` - A CSS class name that can be handy for styling errors based on the error type. (ie. "bad-request" , etc.)
- `data` - An object containing anything you passed to a Feathers error except for the `errors` object.
- `errors` - An object containing whatever was passed to a Feathers error inside `errors`. This is typically validation errors or if you want to group multiple errors together.

Here are a few ways that you can use them:

```
const errors = require('feathers-errors');

// If you were to create an error yourself.
const notFound = new errors.NotFound('User does not exist'));

// You can wrap existing errors
const existing = new errors.GeneralError(new Error('I exist'));

// You can also pass additional data
const data = new errors.BadRequest('Invalid email', {email: 'sergey@google.com'});

// You can also pass additional data without a message
const dataWithoutMessage = new errors.BadRequest({email: 'sergey@google.com'});

// If you need to pass multiple errors
const validationErrors = new errors.BadRequest('Invalid Parameters', {errors: {email:
'Email already taken'} });

// You can also omit the error message and we'll put in a default one for you
const validationErrors = new errors.BadRequest({errors: {email: 'Invalid Email'} });
```

Logging

Just like Express, Feathers does not include a logger. It's left up to you how you want to log things and which logger you want to use. However, a Feathers app created using the generator comes with a basic one that uses winston underneath.

A couple other options are bunyan and morgan.

For production it is recommended that you use an error reporting service. Sentry is an amazing one and has a simple Express middleware that you can easily drop in to your Feathers app.

> **ProTip:** Because Feathers extends Express you can use any Express compatible logging middleware with Feathers.

Configuration

`feathers-configuration` allows you to load default and environment specific JSON configuration files and environment variables and set them on your application. In a generated application the `config/default.json` and `config/production.json` files are set up with feathers-configuration automatically.

Here is what it does:

- Given a root and configuration path load a `default.json` in that path
- When the `NODE_ENV` is not `development`, also try to load `<NODE_ENV>.json` in that path and merge both configurations (with `<NODE_ENV>.json` taking precedence)
- Go through each configuration value and sets it on the application (via `app.set(name, value)`).
  - If the value is a valid environment variable (e.v. `NODE_ENV`), use its value instead
  - If the value start with `./` or `../` turn it it an absolute path relative to the configuration file path
  - If the value starts with a `\`, do none of the above two

Usage

The `feathers-configuration` module is an app configuration function that takes a root directory (usually something like `__dirname` in your application) and the configuration folder (set to `config` by default):

```
const feathers = require('feathers');
const configuration = require('feathers-configuration')

// Use the current folder as the root and look configuration up in `settings`
const app = feathers().configure(configuration(__dirname, 'settings'))
```

Example

In `config/default.json` we want to use the local development environment and default MongoDB connection string:

```
{
  "frontend": "../public",
  "host": "localhost",
  "port": 3030,
  "mongodb": "mongodb://localhost:27017/myapp",
  "templates": "../templates"
}
```

In `config/production.js` we are going to use environment variables (e.g. set by Heroku) and use `public/dist` to load the frontend production build:

```
{
  "frontend": "./public/dist",
  "host": "myapp.com",
  "port": "PORT",
  "mongodb": "MONGOHQ_URL"
}
```

Now it can be used in our `app.js` like this:

```
const feathers = require('feathers');
const configuration = require('feathers-configuration')

const app = feathers()
  .configure(configuration(__dirname));

console.log(app.get('frontend'));
console.log(app.get('host'));
console.log(app.get('port'));
console.log(app.get('mongodb'));
console.log(app.get('templates'));
```

If you now run

```
node app
// -> path/to/app/public
// -> localhost
// -> 3030
// -> mongodb://localhost:27017/myapp
// -> path/to/templates
```

Or with a different environment and variables:

```
PORT=8080 MONGOHQ_URL=mongodb://localhost:27017/production NODE_ENV=production node ap
p
// -> path/to/app/public/dist
// -> myapp.com
// -> 8080
// -> mongodb://localhost:27017/production
// -> path/to/templates
```

Debugging Your Feathers App

You can debug your Feathers app the same as you would any other Node app. There are a few different options you can resort to. NodeJS has a built in debugger that works really well by simply running:

```
node debug src/
```

Moar Logs!

In addition to setting breakpoints we also use the fabulous debug module throughout Feathers core and many of the plug-ins. This allows you to get visibility into what is happening inside all of Feathers by simply setting a `DEBUG` environmental variable to the scope of modules that you want visibility into.

- Debug logs for all the things

  ```
  DEBUG=* npm start
  ```

- Debug logs for all Feathers modules

  ```
  DEBUG=feathers* npm start
  ```

- Debug logs for a specific module

  ```
  DEBUG=feathers-authentication* npm start
  ```

- Debug logs for a specific part of a module

  ```
  DEBUG=feathers-authentication:middleware npm start
  ```

Using Hooks

Since hooks can be registered dynamically anywhere in your app, using them to debug your state at any point in the hook the chain (either before or after a service call) is really handy. For example,

```
const hooks = require('feathers-authentication').hooks;

const myDebugHook = function(hook) {
  // check to see what is in my hook object after
  // the token was verified.
  console.log(hook);
};

// Must be logged in do anything with messages.
app.service('messages').before({
  all: [
    hooks.verifyToken(),
    myDebugHook,
    hooks.populateUser(),
    hooks.restrictToAuthenticated()
  ]
});
```

You can then move that hook around the hook chain and inspect what your `hook` object looks like.

Feathers Security

We take security very seriously at Feathers. We welcome any peer review of our 100% open source code to ensure nobody's Feathers app is ever compromised or hacked. As a web application developer you are responsible for any security breaches. We do our very best to make sure Feathers is as secure as possible.

Where should I report security issues?

In order to give the community time to respond and upgrade we strongly urge you report all security issues to us. Send us a PM in Slack or email us at hello@feathersjs.com with details and we will respond ASAP. Security issues always take precedence over bug fixes and feature work so we'll work with you to come up with a resolution and plan and document the issue on Github in the appropriate repo.

Issuing releases is typically very quick. Once an issue is resolved it is usually released immediately with the appropriate semantic version.

Security Considerations

Here are some things that you should be aware of when writing your app to make sure it is secure.

- Escape any HTML and JavaScript to avoid XSS attacks.
- Escape any SQL (typically done by the SQL library) to avoid SQL injection.
- Events are sent by default to any client listening for that event. Lock down any private events that should not be broadcast by adding filters. Feathers authentication does this for all auth services by default.
- JSON Web Tokens (JWT's) are only signed, they are **not** encrypted. Therefore, the payload can be examined on the client. This is by design. **DO NOT** put anything that should be private in the JWT `payload` unless you encrypt it first.
- Don't use a weak `secret` for you token service. The generator creates a strong one for you automatically. No need to change it.
- Use hooks to check security roles to make sure users can only access data they should be permitted to. We've provided some built in authorization hooks to make this process easier (many of which are added by default to a generated app).

Some of technologies we employ

- Password storage inside `feathers-authentication` uses bcrypt. We don't store the salts separately since they are included in the bcrypt hashes.
- JWT is used instead of cookies to avoid CSRF attacks. We use the `HS512` algorithm by default (HMAC using SHA-512 hash algorithm).
- We run nsp as part of our CI. This notifies us if we are susceptible to any vulernabilites that have been reported to the Node Security Project.

XSS Attacks

As with any web application **you** need to guard against XSS attacks. Since Feathers persists the JWT in localstorage in the browser, if your app falls victim to a XSS attack your JWT could be used by an attacker to make malicious requests on your behalf. This is far from ideal. Therefore you need to take extra care in preventing XSS attacks. Our stance on this particular attack vector is that if you are susceptible to XSS attacks then a compromised JWT is the least of your worries because keystrokes could be logged and attackers can just steal passwords, credit card numbers, directly etc.

For more information see:

- this issue
- and this Auth0 forum thread.

API

Feathers core API is very small and an initialized application provides the same functionality as an Express 4 application. The differences and additional methods and their usage are outlined in this chapter. For the service API refer to the Services chapter.

configure

`app.configure(callback)` runs a `callback` function with the application as the context (`this`). It can be used to initialize plugins or services.

```
function setupService() {
    this.use('/todos', todoService);
}

app.configure(setupService);
```

listen

`app.listen([port])` starts the application on the given port. It will first call the original Express app.listen([port]), then run `app.setup(server)` (see below) with the server object and then return the server object.

setup

`app.setup(server)` is used to initialize all services by calling each services `.setup(app, path)` method (if available). It will also use the `server` instance passed (e.g. through `http.createServer`) to set up SocketIO (if enabled) and any other provider that might require the server instance.

Normally `app.setup` will be called automatically when starting the application via `app.listen([port])` but there are cases when you need to initialize the server separately as described in the VHost, SSL and sub-app chapter.

use

`app.use([path], service)` works just like Express app.use([path], middleware) but additionally allows to register a service object (an object which at least provides one of the service methods as outlined in the Services section) instead of the middleware function. Note that REST services are registered in the same order as any other middleware so the below example will allow the `/todos` service only to Passport authenticated users.

```
// Serve public folder for everybody
app.use(feathers.static(__dirname + '/public');
// Make sure that everything else only works with authentication
app.use(function(req,res,next){
  if(req.isAuthenticated()){
    next();
  } else {
    // 401 Not Authorized
    next(new Error(401));
  }
});
// Add a service.
app.use('/todos', {
  get(id) {
    return Promise.resolve({
      id,
      description: `You have to do ${name}!`
    });
  }
});
```

service

`app.service(path [, service])` does two things. It either returns the Feathers wrapped service object for the given path or registers a new service for that path.

`app.service(path)` returns the wrapped service object for the given path. Feathers internally creates a new object from each registered service. This means that the object returned by `service(path)` will provide the same methods and functionality as your original service object but also functionality added by Feathers and its plugins (most notably it is possible to listen to service events). `path` can be the service name with or without leading and trailing slashes.

```
app.use('/my/todos', {
  create(data) {
    return Promise.resolve(data);
  }
});

var todoService = app.service('my/todos');
// todoService is an event emitter
todoService.on('created', todo =>
    console.log('Created todo', todo)
);
```

You can use `app.service(path, service)` instead `app.use(path, service)` if you want to be more explicit that you are registering a service. It is called internally by `app.use([path], service)` if a service object is passed. `app.service` does **not** provide the Express `app.use` functionality and does not check the service object for valid methods.

Guides

This is where we will have guides for common situations. We will be adding to this section over time but if you have done something cool with Feathers that we don't already have a guide for feel free to open a pull request.

Naturally, our first guide is how you can migrate to your existing app to Feathers 2.

Migrating to Feathers 2

Feathers 2 has become even smaller and more modular. There are no changes in the service API although we recommend using ES6 Promises instead of callbacks. This guide describes how to migrate from previous versions to Feathers v2.

Provider modules

The biggest breaking API change for Feathers 2 core is that the providers which used to be available in `feathers.rest`, `feathers.socketio` and `feathers.primus` are now in their own modules:

- `feathers.rest` in feathers-rest
- `feathers.socketio` in feathers-socketio
- `feathers.primus` in feathers-primus

To migrate, install the provider module you want to use:

```
$ npm install feathers-rest
$ npm install feathers-socketio
$ npm install feathers-primus
```

And then change any configuration that used to look like this:

```
// app.js
var feathers = require('feathers');
var app = feathers();

// Add REST API support
app.configure(feathers.rest());
// Configure Socket.io real-time APIs
app.configure(feathers.socketio());
// Configure Primus real-time APIs
app.configure(feathers.primus());
```

To

```
// app.js
var feathers = require('feathers');
var rest = require('feathers-rest');
var socketio = require('feathers-socketio');
var primus = require('feathers-primus');

var app = feathers();

// Add REST API support
app.configure(rest());
// Configure Socket.io real-time APIs
app.configure(socketio());
// Configure Primus real-time APIs
app.configure(primus());
```

All configuration options are still the same.

> **ProTip:** One additional small difference is that `feathers-socketio` now sets up the connection as a service mixin in the services `setup`. This means `app.configure(socketio())` has to be called **before** registering any services.

Database adapters

If you are using an older version of a database adapter, it will continue to work just the same with Feathers 2 since the service interface didn't change.

The usage of the latest database adapters has been unified to support a common way for extension, querying and pagination. They now require establishing a connection outside of the adapter and you now pass the database connection instance or ORM model to the service. For detailed information follow up in the adapter, ORM or database chapters below:

- **Memory** - feathers-memory
- **MongoDB** - feathers-mongoose
- **NeDB** - feathers-nedb
- **PostgreSQL, MySQL, MariaDB, and SQLite**
  - feathers-knex
  - feathers-sequelize
- **Oracle** - feathers-knex
- **Microsoft SQL Server** - feathers-sequelize
- **Waterline** - feathers-waterline

Feathers client

Feathers client is now universal! Meaning it can be used in in the browser, NodeJS, and in React Native. However, it can still be used almost the same way but it is now a module that consolidates all the individual client side modules of Feathers 2.

The main API difference is that REST clients changed from an initialization like:

```
var app = feathers('http://baseUrl');

app.configure(feathers.jquery());
app.configure(feathers.request(request));
app.configure(feathers.superagent(superagent));
app.configure(feathers.fetch(fetch));
```

To:

```
var app = feathers();
var rest = feathers.rest('http://api.my-feathers-app.com');

// jQuery now always needs to be passed
app.configure(rest.jquery(window.jQuery));

app.configure(rest.request(request));
app.configure(rest.superagent(superagent));
app.configure(rest.fetch(fetch));
```

This will configure a default service provider which will use one of those HTTP client libraries to connect to a remote service.

```
// will use the service at http://api.my-feathers-app.com/todos
var todoService = app.service('todos');

todoService.find().then(function(todos) {
  console.log('Found todos', todos);
});
```

Because a client `app` is now the same as a normal Feathers application it is possible to use Feathers functionality like hooks (`app.configure(feathers.hooks())`), custom services and client side authentication on the client.

Using A View Engine

Since Feathers is just an extension of Express it's really simple to render templated views on the server with data from your Feathers services. There are a few different ways that you can structure your app so this guide will show you 3 typical ways you might have your Feathers app organized.

You can see a basic working example here.

A Single "Monolithic" App

You probably already know that when you register a Feathers service, Feathers creates RESTful endpoints for that service automatically. Well, really those are just Express routes, so you can define your own as well.

> **ProTip:** Your own defined REST endpoints won't work with hooks and won't emit socket events. If you find you need that functionality it's probably better for you to turn your endpoints into a minimal Feathers service.

Let's say you want to render a list of messages from most recent to oldest using the Jade template engine.

```
// You've set up your main Feathers app already

// Register your view engine
app.set('view engine', 'jade');

// Register your message service
app.use('/api/messages', memory());

// Inside your main Feathers app
app.get('/messages', function(req, res, next){
  // You namespace your feathers service routes so that
  // don't get route conflicts and have nice URLs.
  app.service('api/messages')
    .find({ query: {$sort: { updatedAt: -1 } } })
    .then(result => res.render('message-list', result.data))
    .catch(next);
});
```

Simple right? We've now rendered a list of messages. All your hooks will get triggered just like they would normally so you can use hooks to pre-filter your data and keep your template rendering routes super tight.

> **ProTip:** If you call a Feathers service "internally" (ie. not over sockets or REST) you won't have a `hook.params.provider` attribute. This allows you to have hooks only execute when services are called externally vs. from your own code. See bundled hooks for an example.

Feathers As A Sub-App

Sometimes a better way to break up your Feathers app is to put your services into an API and mount your API as a sub-app. This is just like you would do with Express. If you do this, it's only a slight change to get data from your services.

```
// You've set up your main Feathers app already

// Register your view engine
app.set('view engine', 'jade');

// Require your configured API sub-app
const api = require('./api');

// Register your API sub app
app.use('/api', api);

app.get('/messages', function(req, res, next){
  api.service('messages')
    .find({ query: {$sort: { updatedAt: -1 } } })
    .then(result => res.render('message-list', result.data))
    .catch(next);
});
```

Not a whole lot different. Your API sub app is pretty much the same as your single app in the previous example, and your main Feathers app is just a really small wrapper that does little more than render templates.

Feathers As A Separate App

If your app starts to get a bit busier you might decide to move your API to a completely separate standalone Feathers app, maybe even on a different server. In order for both apps to talk to each other they'll need some way to make remote requests. Well, Feathers just so happens to have a client side piece that can be used on the server. This is how it works.

```
// You've set up your feathers app already

// Register your view engine
app.set('view engine', 'jade');

// Include the Feathers client modules
const client = require('feathers/client');
const socketio = require('feathers-socketio/client');
const io = require('socket.io-client');

// Set up a socket connection to our remote API
const socket = io('http://api.feathersjs.com');
const api = client().configure(socketio(socket));

app.get('/messages', function(req, res, next){
  api.service('messages')
    .find({ query: {$sort: { updatedAt: -1 } } })
    .then(result => res.render('message-list', result.data))
    .catch(next);
});
```

> **ProTip:** In the above example we set up sockets. Alternatively you could use a Feathers client REST provider.

And with that, we've shown 3 different ways that you use a template engine with Feathers to render service data. If you see any issues in this guide feel free to submit a pull request.

File uploads in Feathersjs

Over the last months we at ciancoders.com have been working in a new SPA project using Feathers and React, the combination of those two turns out to be **just amazing**.

Recently we where struggling to find a way to upload files whitout having to write a separate Express middleware or having to (re)write a complex Feathers service.

Our Goals

We want to implement an upload service in a way that it acomplishes these few important things:

1. It has to handle large files (+10MB).
2. Needs to work with the app's authetication and authorization.
3. The files has to be validated.
4. At the moment there is no third party storage service involved, but this will change in the near future, so it has to be prepared.
5. Has to show the upload progress.

The plan is to upload the files to a feathers service, so we can take advantage of the hooks for authentication, authorization and validation, not to mention the service events.

Fortunately, there is an already developed file storage service: feathers-blob. With it we can easily archieve our goals, but (spoiler alert) it isn't the ideal solution, as it has some problems we will discuss below.

Basic upload with feathers-blob and feathers-client

For the sake of simplicity, we will be working over a very basic feathers server, with just the upload service.

Lets look at the server code:

```
/* --- server.js --- */

const feathers = require('feathers');
const rest = require('feathers-rest');
const socketio = require('feathers-socketio');
const hooks = require('feathers-hooks');
const bodyParser = require('body-parser');
const handler = require('feathers-errors/handler');


// feathers-blob service
const blobService = require('feathers-blob');
// Here we initialize a FileSystem storage,
// but you can use feathers-blob with any other
// storage service like AWS or Google Drive.
const fs = require('fs-blob-store');
const blobStorage = fs(__dirname + '/uploads');


// Feathers app
const app = feathers();

// Parse HTTP JSON bodies
app.use(bodyParser.json());
// Parse URL-encoded params
app.use(bodyParser.urlencoded({ extended: true }));
// Register hooks module
app.configure(hooks());
// Add REST API support
app.configure(rest());
// Configure Socket.io real-time APIs
app.configure(socketio());


// Upload Service
app.use('/uploads', blobService({Model: blobStorage}));


// Register a nicer error handler than the default Express one
app.use(handler());

// Start the server
app.listen(3030, function(){
    console.log('Feathers app started at localhost:3030')
});
```

`feathers-blob` works over abstract-blob-store, wich is an abstract interface to various storage backends, such as filesystem, AWS, or Google Drive. It only accepts and retrieves files encoded as dataURI strings.

Just like that we have our backend ready, go ahead and POST something to localhost:3030/uploads`, for example with postman:

```
{
    "uri": "
WfvxW/zwXPrtW/vxXvfrXv3xYvrvYvntYvnvY/ruZPrwZPfsZPjsZfjtZvfsZvHmY/zxavftaPrvavjuafzxbf
nua/jta/ftbP3yb/zzcPvwb/zzcfvxcfzxc/3zdf3zdv70efvwd/rwd/vwefftd/3yfPvxfP70f/zzfvnwffvz
f/rxf/rxgPjvgPjvgfnwhPvzhvjvhv71jfz0kPrykvz0mv72nvblTPnnUPjoUPrpUvnnUfnpUvXlUfnpU/npVP
nqVPfnU/3uVvvsWPfpVvnqWfrrXPLiW/nrX/vtYv7xavrta/Hlcvnuf/Pphvbsif3zk/zzlPzylfjuk/z0o/Lq
nvbhSPbhSfjiS/jlS/jjTPfhTfjlTubUU+/iiPPokfrvl/Dll/ftovLWPfHXPvHZP/PbQ/bcRuDJP/PaRvjgSf
fdSe3ddu7fge7fi+zkuO7NMvPTOt2/Nu7SO+3OO/PWQdnGbOneqeneqvDqyu3JMuvJMu7KNfHNON7GZdnEbeja
nObXnOW8JOa9KOvCLOnBK9+4Ku3FL9ayKuzEMcenK9e+XODOiePSkODOkOW3ItisI9yxL+a9NtGiHr+VH5h5Js
SfNM2bGN6rMJt4JMOYL5h4JZl5Jph3Jpl4J5h5J5h3KJl4KZp5Ks+sUN7Gi96lLL+PKMmbMZt2Jpp3Jpt3KZl4
K7qFFdyiKdufKsedRdm7feOpQN2QKMKENrpvJbFfIrNjJL1mLMBpLr9oLrFhK69bJFkpE1kpFYNeTqFEIlsoFb
mlnlsmFFwpGFkoF/////7+/v///wAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACH5BAEAANAALAAAAAATABMAAAj/AKEJHCgokKJKlhTh
GciQYSIva7r8SHPFzqGGAwPd4bKlh5YsPKy0qFLnT0NAaHTcsIHDho0aKkaAwGCGEkM1NmSkIjWLBosVJT6cOj
UrzsBKPl54KmYsACoTMmk1WwaA1CRoeM7siJEqmTIAsjp40ICK2bEApfZcsoQlxwxRzgI8W8XhgoVYA+Kq6sMK
0QEYKVCUkoVqQwQJFTwFEAAAFZ9PlFy4OEEiRIYJD55EodDA1ClTbPp0okRFxBQDBRgskAKhiRMlc+Sw4SNpFC
IoBBwkUMBkCBIiY8qAgcPG0KBHrBTFQbCEV5EjQYQACfNFjp5CgxpxagVtUhIjwzaJYSHzhQ4cP3ryQHLEqJbA
Snu+6EIW6o2b2X0ISXK0CFSugazs0YYmwQhziyuE2PLLIv3h0hArkRhiCCzAENOLL7tgAoqDGLXSSSaPMLIIJp
mAUst/GA3UCiuv1PIKLtw1FBAAOw=="
}
```

The service will respond with something like this:

```
{
  "id": "6454364d8facd7a88e627e4c4b11b032d2f83af8f7f9329ffc2b7a5c879dc838.gif",
  "uri": "the-same-uri-we-uploaded",
  "size": 1156
}
```

Or we can implement a very basic frontend with `feathers-client` and `jQuery`:

```
<!doctype html>
<html>
    <head>
        <title>Feathersjs File Upload</title>
        <script   src="https://code.jquery.com/jquery-2.2.3.min.js"   integrity="sha25
6-a23g1Nt4dtEYOj7bR+vTu7+T8VP13humZFBJNIYoEJo="   crossorigin="anonymous"></script>
        <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2
.1.4/core.min.js"></script>
        <script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/f
eathers.js"></script>
        <script type="text/javascript">
            // feathers client initialization
            const rest = feathers.rest('http://localhost:3030');
            const app = feathers()
```

```
            .configure(feathers.hooks())
            .configure(rest.jquery($));

            // setup jQuery to watch the ajax progress
            $.ajaxSetup({
                xhr: function () {
                    var xhr = new window.XMLHttpRequest();
                    // upload progress
                    xhr.addEventListener("progress", function (evt) {
                        if (evt.lengthComputable) {
                            var percentComplete = evt.loaded / evt.total;
                            console.log('upload progress: ', Math.round(percentComplet
e * 100) + "%");
                        }
                    }, false);
                    return xhr;
                }
            });

            const uploadService = app.service('uploads');
            const reader  = new FileReader();

            // encode selected files
            $(document).ready(function(){
                $('input#file').change(function(){
                    var file = this.files[0];
                    // encode dataURI
                    reader.readAsDataURL(file);
                })
            });

            // when encoded, upload
            reader.addEventListener("load", function () {
                console.log('encoded file: ', reader.result);
                var upload = uploadService
                    .create({uri: reader.result})
                    .then(function(response){
                        // success
                        alert('UPLOADED!! ');
                        console.log('Server responded with: ', response);
                    });
            }, false);
        </script>
    </head>
    <body>
        <h1>Let's upload some files!</h1>
        <input type="file" id="file"/>
    </body>
</html>
```

This code watches for file selection, then encodes it and does an ajax post to upload it, watching the upload progress via the xhr object. Everything works as expected.

Every file we select gets uploaded and saved to the `./uploads` directory.

Work done!, let's call it a day, shall we?

... But hey, there is somethiing that doesn't feels quite right ...right?

DataURI upload problems

It doesn't feels right because it is not. Let's imagine what would happen if we try to upload a large file, say 25MB or more: The entire file (plus some extra MB due to the encoding) has to be kept in memory for the entire upload proccess, this could look like nothing for a normal computer but for mobile devices it's a big deal.

We have a big RAM consumption problem. Not to mention we have to encode the file before sending it...

The solution would be to modify the service, adding support for splitting the dataURI into small chunks, then uploading one at a time, collecting and reasembling everything on the server. But hey, it's not that the same thing browsers and web servers has been doing since maybe the very early days of the web? maybe since Netscape Navigator?

Well, actually it is, and doing a `multipart/form-data` post is still the easiest way to upload a file.

Feathers-blob with multipart support.

Back with the backend, in order to accept multipart uploads, we need a way to handle the `multipart/form-data` received by the web server. Given that Feathers behaves like Express, let's just use `multer` and a custom middleware to handle that.

```
/* --- server.js --- */
const multer = require('multer');
const multipartMiddleware = multer();

// Upload Service with multipart support
app.use('/uploads',

    // multer parses the file named 'uri'.
    // Without extra params the data is
    // temporarely kept in memory
    multipartMiddleware.single('uri'),

    // another middleware, this time to
    // transfer the received file to feathers
    function(req,res,next){
        req.feathers.file = req.file;
        next();
    },
    blobService({Model: blobStorage})
);
```

Notice we kept the file field name as *uri* just to maintain uniformity, as the service will always work with that name anyways. But you can change it if you preffer.

Feathers-blob only understands files encoded as dataURI, so we need to convert them first. Let's make a Hook for that:

```
/* --- server.js --- */
const dauria = require('dauria');

// before-create Hook to get the file (if there is any)
// and turn it into a datauri,
// transparently getting feathers-blob to work
// with multipart file uploads
app.service('/uploads').before({
    create: [
        function(hook) {
            if (!hook.data.uri && hook.params.file){
                const file = hook.params.file;
                const uri = dauria.getBase64DataURI(file.buffer, file.mimetype);
                hook.data = {uri: uri};
            }
        }
    ]
});
```

Et voilà!. Now we have a Feahtersjs file storage service working, with support for traditional multipart uploads, and a variety of storage options to choose.

Simply awesome.

Further improvements

The service always return the dataURI back to us, wich may not be neccesary as we'd just uploaded the file, also we need to validate the file and check for authorization.

All those things can be easily done with more Hooks, and that's the benefit of keeping all inside feathersjs services. I left that to you.

For the frontend, there is a problem with the client: in order to show the upload progress it's stuck with only REST functionality and not real-time with socket.io.

The solution is to switch `feathers-client` from REST to `socket.io`, and just use wherever you like for uploading the files, thats an easy task now that we are able to do a traditional `form-multipart` upload.

Here is an example using dropzone:

```
<!doctype html>
<html>
    <head>
        <title>Feathersjs File Upload</title>

        <link rel="stylesheet" href="assets/dropzone.css">
        <script src="assets/dropzone.js"></script>

        <script type="text/javascript" src="socket.io/socket.io.js"></script>
        <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2
.1.4/core.min.js"></script>
        <script type="text/javascript" src="//npmcdn.com/feathers-client@^1.0.0/dist/f
eathers.js"></script>
        <script type="text/javascript">
            // feathers client initialization
            var socket = io('http://localhost:3030');
            const app = feathers()
            .configure(feathers.hooks())
            .configure(feathers.socketio(socket));
            const uploadService = app.service('uploads');

            // Now with Real-Time Support!
            uploadService.on('created', function(file){
                alert('Received file created event!', file);
            });


            // Let's use DropZone!
            Dropzone.options.myAwesomeDropzone = {
                paramName: "uri",
                uploadMultiple: false,
                init: function(){
                    this.on('uploadprogress', function(file, progress){
                        console.log('progresss', progress);
                    });
                }
            };
        </script>
    </head>
    <body>
        <h1>Let's upload some files!</h1>
        <form action="/uploads"
          class="dropzone"
          id="my-awesome-dropzone"></form>
    </body>
</html>
```

All the code is available via github here: https://github.com/CianCoders/feathers-example-fileupload

Hope you have learned something today, as I learned a lot writing this.

Cheers!

Creating a Feathers Plugin

The easiest way to create a plugin is with the Yeoman generator.

First install the generator

```
$ npm install -g generator-feathers-plugin
```

Then in a new directory run:

```
$ yo feathers-plugin
```

This will scaffold out everything that is needed to start writing your plugin.

Output files from generator:

```
    create package.json
    create .babelrc
    create .editorconfig
    create .jshintrc
    create .travis.yml
    create example/app.js
    create src/index.js
    create test/index.test.js
    create README.md
    create LICENSE
    create .gitignore
    create .npmignore
```

Simple right? We technically could call it a day as we have created a Plugin. However, we probably want to do just a bit more. Generally speaking a Plugin is a Service. The fun part is that a Plugin can contain multiple Services which we will create below. This example is going to build out 2 services. The first will allow us to find members of the Feathers Core Team & the second will allow us to find the best state in the United States.

Verifying our Service

Before we start writing more code we need to see that things are working.

```
$ cd example && node app.js

Error: Cannot find module '../lib/index'
```

Dang! Running the example app resulted in an error and you said to yourself, "The generator must be broken and we should head over to the friendly Slack community to start our debugging journey". Well, as nice as they may be we can get through this. Let's take a look at the package.json that came with our generator. Most notably the scripts section.

```
"scripts": {
    "prepublish": "npm run compile",
    "publish": "git push origin && git push origin --tags",
    "release:patch": "npm version patch && npm publish",
    "release:minor": "npm version minor && npm publish",
    "release:major": "npm version major && npm publish",
    "compile": "rimraf lib/ && babel -d lib/ src/",
    "watch": "babel --watch -d lib/ src/",
    "jshint": "jshint src/. test/. --config",
    "mocha": "mocha --recursive test/ --compilers js:babel-core/register",
    "test": "npm run compile && npm run jshint && npm run mocha",
    "start": "npm run compile && node example/app"
  }
```

Back in business. That error message was telling us that we need to build our project. In this case it means babel needs to do it's thing. For development you can run watch

```
$ npm run watch

> creatingPlugin@0.0.0 watch /Users/ajones/git/training/creatingPlugin
> babel --watch -d lib/ src/

src/index.js -> lib/index.js
```

After that you can run the example app and everything should be good to go.

```
$ node app.js
Feathers app started on 127.0.0.1:3030
```

Expanding our Plugin

Now that we did our verification we can safely change things. For this example we want 2 services to be exposed from our Plugin. Let's create a services directory within the src folder.

```
// From the src directory
$ mkdir services
$ ls
index.js services
```

Now let's create our two services. We will just copy the index.js file.

```
$ cp index.js services/core-team.js
$ cp index.js services/best-state.js
$ cd services && ls
best-state.js core-team.js

$ cat best-state.js

if (!global._babelPolyfill) { require('babel-polyfill'); }

import errors from 'feathers-errors';
import makeDebug from 'debug';

const debug = makeDebug('creatingPlugin');

class Service {
  constructor(options = {}) {
    this.options = options;
  }

  find(params) {
    return new Promise((resolve, reject) => {
      // Put some async code here.
      if (!params.query) {
        return reject(new errors.BadRequest());
      }

      resolve([]);
    });
  }
}

export default function init(options) {
  debug('Initializing creatingPlugin plugin');
  return new Service(options);
}

init.Service = Service;
```

At this point we have index.js, best-state.js and core-team.js with identical content. The next step will be to change index.js so that it is our main file.

Our new index.js

```
if (!global._babelPolyfill) { require('babel-polyfill'); }

import coreTeam from './services/core-team';
import bestState from './services/best-state';

export default { coreTeam, bestState };
```

Now we need to actually write the services. We will only be implementing the find action as you can reference the service docs to learn more. Starting with core-team.js we want to find out the names of the members listed in the feathers.js org on github.

```
//core-team.js
if (!global._babelPolyfill) { require('babel-polyfill'); }

import errors from 'feathers-errors';
import makeDebug from 'debug';

const debug = makeDebug('creatingPlugin');

class Service {
  constructor(options = {}) {
    this.options = options;
  }

  //We are only changing the find...
  find(params) {
    return Promise.resolve(['Mikey', 'Cory Smith', 'David Luecke', 'Emmanuel Bourmalo'
, 'Eric Kryski',
      'Glavin Wiechert', 'Jack Guy', 'Anton Kulakov', 'Marshall Thompson'])
  }
}

export default function init(options) {
  debug('Initializing creatingPlugin plugin');
  return new Service(options);
}

init.Service = Service;
```

That will now return an array of names when service.find is called. Moving on to the best-state service we can follow the same pattern

```
if (!global._babelPolyfill) { require('babel-polyfill'); }

import errors from 'feathers-errors';
import makeDebug from 'debug';

const debug = makeDebug('creatingPlugin');

class Service {
  constructor(options = {}) {
    this.options = options;
  }

  find(params) {
    return Promise.resolve(['Alaska']);
  }
}

export default function init(options) {
  debug('Initializing creatingPlugin plugin');
  return new Service(options);
}

init.Service = Service;
```

Notice in the above service it return a single item array with the best state listed.

Usage

The easiest way to use our plugin will be within the same app.js file that we were using earlier. Let's write out some basic usage in that file.

```
//app.js
const feathers = require('feathers');
const rest = require('feathers-rest');
const hooks = require('feathers-hooks');
const bodyParser = require('body-parser');
const errorHandler = require('feathers-errors/handler');
const plugin = require('../lib/index');

// Initialize the application
const app = feathers()
  .configure(rest())
  .configure(hooks())
  // Needed for parsing bodies (login)
  .use(bodyParser.json())
  .use(bodyParser.urlencoded({ extended: true }))
  // Initialize your feathers plugin
  .use('/plugin/coreTeam', plugin.coreTeam())
  .use('/plugin/bestState', plugin.bestState())
  .use(errorHandler());


var bestStateService = app.service('/plugin/bestState')
var coreTeamService = app.service('/plugin/coreTeam')

bestStateService.find().then( (result) => {
  console.log(result)
}).catch( error => {
  console.log('Error finding the best state ', error)
})

coreTeamService.find().then( (result) => {
  console.log(result)
}).catch( error => {
  console.log('Error finding the core team ', error)
})

app.listen(3030);

console.log('Feathers app started on 127.0.0.1:3030');
```

```
$ node app.js

Feathers app started on 127.0.0.1:3030
[ 'Alaska' ]
[ 'Mikey',
  'Cory Smith',
  'David Luecke',
  'Emmanuel Bourmalo',
  'Eric Kryski',
  'Glavin Wiechert',
  'Jack Guy',
  'Anton Kulakov',
  'Marshall Thompson' ]
```

Testing

Our generator stubbed out some basic tests. We will remove everything and replace it with the following.

```
import { expect } from 'chai';
import plugin from '../src';

const bestStateService = plugin.bestState()

describe('bestState', () => {
  it('is Alaska', () => {
    bestStateService.find().then(result => {
      console.log(result)
      expect(result).to.eql(['Alaska']);
      done();
    });
  });
});
```

```
$ npm run test
```

Because this is just a quick sample jshint might fail. You can either fix the syntax or change the test command.

```
$ npm run compile && npm run mocha
```

This should give you the basic idea of creating a Plugin for Feathers.

Help!

There are many ways that you can get help but before you explore them please check other parts of this guide, the FAQ, Stackoverflow, Github Issues and our Medium publication "All About FeathersJS".

If none of those work it's a very real possibility that we screwed something up or it's just not clear. We're sorry ☹. We want to hear about it and are very friendly so feel free to come talk to us in Slack, submit your issue on Github or ask on StackOverflow using the feathersjs tag.

FAQ

We've been collecting some commonly asked questions here. We'll either be updating the guide directly, providing answers here, or both.

How do I debug my app

It's really no different than debugging any other NodeJS app but you can refer to the Debugging section of the guide for more Feathers specific tips and tricks.

Can I expose custom service methods?

Yes and no. You can create custom methods but they won't be exposed over sockets automatically and they won't be mapped to a REST verb (GET, POST, PUT, PATCH, DELETE). See this section for more detail.

I am getting plain text errors with a 500 status code

If you get a plain text error and a 500 status code for errors that should return different status codes, make sure you have `feathers-errors/handler` configured as described in the error handling chapter.

How can I do custom methods like `findOrCreate`?

Custom functionality can almost always be mapped to an existing service method using hooks. For example, `findOrCreate` can be implemented as a before-hook on the service's `get` method. See this gist for an example of how to implement this in a before-hook.

How do I do nested routes?

Normally we find that they actually aren't needed and that it's much better to keep your routes as flat as possible. However, if the need arises there are a couple different ways. Refer to this section for details.

How do I do I render templates?

Feathers works just like Express so it's the exact same. We've created a helpful little guide right here.

How do I create channels or rooms

Although Socket.io has a concept of rooms and you can always fall back to it other websocket libraries that Feathers supports do not. The Feathers way of letting a user listen to e.g. messages on a room is through event filtering. There are two ways:

1. Update the user object with the rooms they are subscribed to and filter based on those

```
// On the client
function joinRoom(roomId) {
  const user = app.get('user');

  return app.service('users').patch(user.id, { rooms: user.rooms.concat(roomId) });
}

// On the server
app.service('messages').filter(function(message, connection) {
  return connection.user.rooms.indexOf(message.room_id) !== -1;
});
```

The advantage of this method is that you can show offline/online users that are subscribed to a room.

1. Create a custom `join` event with a room id and then filter based on it

```
app.use(socketio(function(io) {
  io.on('connection', function(socket) {
    socket.on('join', function(roomId) {
      socket.feathers.rooms.push(roomId);
    });
  });
}));

app.service('messages').filter(function(message, connection) {
  return connection.rooms.indexOf(message.room_id) !== -1;
});
```

The room assignment will persist only for the duration of the socket connection.

I got a `possible EventEmitter memory leak detected` warning

This warning is not as bad as it sounds. If you got it from Feathers you most likely registered more than 64 services and/or event listeners on a Socket. If you don't think there are that many services or event listeners you may have a memory leak. Otherwise you can increase the number in the Socket.io configuration via `io.sockets.setMaxListeners(number)` and with Primus via `primus.setMaxListeners(number)`. `number` can be `0` for unlimited listeners or any other number of how many listeners you'd expect in the worst case.

How do I do validation?

If your database/ORM supports a model or schema (ie. Mongoose or Sequelize) then you have 2 options.

The preferred way

You perform validation at the service level using hooks. This is better because it keeps your app database agnostic so you can easily swap databases without having to change your validations much.

If you write a bunch of small hooks that validate specific things it is easier to test and also slightly more performant because you can exit out of the validation chain early instead of having to go all the way to the point of inserting data into the database to find out if that data is invalid.

If you don't have a model or schema then validating with hooks is currently your only option. If you come up with something different feel free to submit a PR!

The ORM way

With ORM adapters you can perform validation at the model level:

- Using Mongoose
- Using Sequelize

The nice thing about the model level validations is Feathers will return the validation errors to the client in a nice consistent format for you.

How do I return related entities?

Similar to validation, it depends on if your database/ORM supports models or not.

The preferred way

For any of the feathers database/ORM adapters you can just use hooks to fetch data from other services as described here.

This is a better approach because it keeps your application database agnostic and service oriented. By referencing the services (using `app.service().find()`, etc.) you can still decouple your app and have these services live on entirely separate machines or use entirely different databases without having to change any of your fetching code.

The ORM way

With mongoose you can use the `$populate` query param to populate nested documents.

```
// Find Hulk Hogan and include all the messages he sent
app.service('user').find({
  query: {
    name: 'hulk@hogan.net',
    $populate: ['sentMessages']
  }
});
```

With Sequelize you can do this:

```
// Find Hulk Hogan and include all the messages he sent
app.service('user').find({
  name: 'hulk@hogan.net',
  sequelize: {
    include: [{
      model: Message,
      where: { sender: Sequelize.col('user.id') }
    }]
  }
});
```

What about Koa/Hapi/X?

There are many other Node server frameworks out there like Koa, a *"next generation web framework for Node.JS"* using ES6 generator functions instead of Express middleware or HapiJS etc. Because Feathers 2 is already univerally usable we are planning the ability for it to hook into other frameworks on the server as well. More information can be found in this issue.

How do I filter emitted service events?

See this section.

How do I access the request object in hooks or services?

In short, you shouldn't need to. If you look at the hooks chapter you'll see all the params that are available on a hook.

If you still need something from the request object (for example, the requesting IP address) you can simply tack it on to the `req.feathers` object as described here.

How do I mount sub apps?

It's pretty much exactly the same as Express. There is an example of how to do this in our Examples repository.

Help Us Write the Book!

Just like Feathers itself, all of the documentation is open source and available to edit on GitHub. If you see something that you can contribute, we would LOVE a pull request with your edits! To make this easy you can click the *"Edit this page"* link at the top of the web docs.

Contributing Guidelines

The docs are all written in GitHub Flavored Markdown. If you've used GitHub, it's pretty likely you've encountered it before. You can become a pro in a few minutes by reading their GFM Documentation page.

Organizing Files

You'll notice that the GitHub Repo is organized in a nice logical folder structure. The first file in each chapter is named as a description of the entire chapter's topic. For example, the intro to databases is located in `databases/readme.md`.

Some of the chapters are split into multiple sections to help break up the content and make it easier to digest. You can easily see how chapters are laid out by looking at the `SUMMARY.md` file. This convention helps keep chapters together in the file system and easy to view either directly on github or gitbook.

Table of Contents

SUMMARY.md = Table of Contents.

You'll find the table of contents in the SUMMARY.md file. It's a nested list of markdown links. You can link to a file simply by putting the filename (including the extension) inside the link target.

Introduction Page

Intro Page = README.md

Give you the elevator pitch of what Feathers is and why we think it is useful.

Send a Pull Request

So that's it. You make your edits, keep your files and the Table of Contents organized, and send us a pull request.

Enjoy the Offline Docs

Moments after your edits are merged, they will be automatically published to the web, as a downloadable PDF, .mobi file (Kindle compatible), and ePub file (iBooks compatible).

Share

We take pride in having great documentation and we are very appreciative of any help we can get. Please, let the world know you've contributed to the Feathers Book or give @FeathersJS a shout out on Twitter when your changes are merged.

Changelog

2.0.1

- Allow services with only a `setup` method (#285, #308)
- Remove JSON loading from the client version for better loader support (#306)

2.0.0

- Separate API providers into feathers-rest, feathers-socketio and feathers-primus
- Make Feathers universal (isomorphic)
- Change websocket service paths to use parent application mountpoint
- Migrate codebase to ES6

1.3.0

- Add ability to create, update, patch and remove many (#144, #179)
- Handle middleware passed after the service to app.use (#176, #178)

1.2.0

- Add hook object to service events parameters (#148)
- Argument normalization runs before event mixin punched methods (#150)

1.1.1

- Fix 404 not being properly thrown by REST provider (#146)

1.1.0

- Service `setup` called before `app.io` instantiated (#131)
- Allow to register services that already are event emitter (#118)
- Clustering microservices (#121)
- Add debug module and messages (#114)
- Server hardening with socket message validation and normalization (#113)
- Custom service events (#111)
- Support for registering services dynamically (#67, #96, #107)

1.0.2

- Use Uberproto extended instance when creating services (#105)
- Make sure that mixins are specific to each new app (#104)

1.0.1

- Rename Uberproto .create to avoid conflicts with service method (#100, #99)

1.0.0

- Remove app.lookup and make the functionality available as app.service (#94)
- Allow not passing parameters in websocket calls (#92)
- Add _setup method (#91)
- Throw an error when registering a service after application start (#78)
- Send socket parameters as params.query (#72)
- Send HTTP 201 and 204 status codes (#71)
- Upgrade to SocketIO 1.0 (#70)
- Upgrade to Express 4.0 (#55, #54)
- Allow service methods to return a promise (#59)
- Allow to register services with custom middleware (#56)
- REST provider should not be added by default (#53)

0.4.0

- Allow socket provider event filtering and params passthrough (#49, #50, #51)
- Added `patch` support (#47)
- Allow to configure REST handler manually (#40, #52)

0.3.2

- Allows Feathers to use other Express apps (#46)
- Updated dependencies and switched to Lodash (#42)

0.3.1

- REST provider refactoring (#35) to make it easier to develop plugins
- HTTP requests now return 405 (Method not allowed) when trying to access unavailable service methods (#35)

0.3.0

- Added Primus provider (#34)
- `app.setup(server)` to support HTTPS (and other functionality that requires a custom server) (#33)
- Removed bad SocketIO configuration (#19)
- Add .npmignore to not publish .idea folder (#30)
- Remove middleware: connect.bodyParser() (#27)

0.2.0

- Pre-initialize `req.feathers` in REST provider to set service parameters
- Allowing to initialize services with or without slashes to be more express-compatible

0.1.0

- First beta release
- Directly extends Express
- Removed built in services and moved to Legs
- Created example repository

0.0.x

- Initial test alpha releases

The MIT license