Feathers

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

The official guide

Table of Contents

> An open source REST and realtime API layer for modern applications.

Feathers is *a batteries included but entirely optional* minimal web application framework.

At it's core, it's a set of tools and an architecture pattern that make it easy to create scalable REST APIs and real-time applications. With Feathers, you can literally build prototypes in minutes and production-ready apps in days.

Feathers achieves this by being the glue code between some amazing battle tested open source technologies - adding a few minimal abstractions and introducing an application architecture that is easier to understand, maintain, and scale than the traditional MVC architecture.

If you're interested, you can read more about how and why Feathers came to be or find out more ab out its features and how Feathers compares to others.

If you've decided that Feathers might be for you, feel free to dive right in and learn about the basics.

Guides

The basics

Learn about Feathers core concepts and how they fit together by going from your first Feathers application to a database backed REST and real-time API.

A Chat Application

Learn how to create a chat REST and real-time API complete with authentication and data processing and how to use Feathers on the client in a simple browser chat application.

Authentication

Learn how to add local (username & password), OAuth1, and OAuth2 authentication to your Feathers Applications and some recipes.

Advanced topics

Guides for more advanced Feathers topics like debugging, file uploads and more.

The FeathersJS Youtube playlist

A growing collection of Feathers related talks, tutorials and discussions.

Feathers basics

This guide will go over the basics and core concepts of any Feathers application.

Setting up

Learn about what is required to know and install to best learn and get started with Feathers.

Getting started

Create your first Feathers application that works in NodeJS and the browser.

Services

The heart of every Feathers application and the core concept for abstracting data access.

Hooks

Middleware for services to handle things like validation, authorization, logging, populating related entities, sending notifications and more.

REST APIs

Learn how to turn a service into a REST API.

Databases

Learn about the pre-built services for different databases.

Real-time APIs

Turn a database into a fully featured REST and real-time API.

Clients

Use a Feathers REST and real-time API server through Feathers in the browser.

The generator (CLI)

Shows how to install the Feathers CLI and the patterns it uses to structure an application.

Setting up

In this section we will go over the required tools and preliminary knowledge for best learning Feathers.

Prerequisites

Feathers and most plug-ins work on NodeJS v6.0.0 and up. For the guides we will use syntax that only works with Node v8.0.0 and later. On MacOS and other Unix systems the Node Version Manager is a good way to quickly install the latest version of NodeJS and keep up it up to date.

After successful installation, the `node` and `npm` commands should be available on the terminal and show something similar when running the following commands:

```
$ node --version
v8.5.0
```

```
$ npm --version
5.5.1
```

Feathers does work in the browser and supports IE 10 and up. The examples used in this guide will however only work in the most recent versions of Chrome, Firefox, Safari and Edge.

What you should know

Readers should have reasonable JavaScript experience using ES6 and some experience with NodeJS and the JavaScript features it supports like the module system. Some familiarity with HTTP and REST APIs as well as websockets is also helpful.

The guide examples use async/await. Familiarity with Promises and `async/await` (and how they interact) is highly recommended. For a good introduction to JavaScript promises see promisejs.org and then follow up with `async/await` in this blog post.

Feathers works standalone but also provides an integration with Express. This guide does not require any in-depth knowledge of Express but some experience with Express will be helpful in the future (see the Express guide to get started).

What we won't cover

Although Feathers works with many database this guide will only use examples of standalone database adapters so there is no need to run a database server.

Authentication will be shown later in the chat application guide.

All examples will reside within a single file. The Feathers generator (CLI) will create a recommended structure for a Feathers application. You can see what it does to structure an application in the Generator guide and how to use it in the chat application guide.

What's next?

All set up and good to go? Let's install Feathers and create our first app.

Our first Feathers application

Now that we are set up we can create our first Feathers application. It will work in both, NodeJS and the browser. First, let's create a new folder for all our examples to run in:

```
mkdir feathers-basics
cd feathers-basics
```

Since any Feathers application is a Node application, we can create a default package.json using `npm`:

```
npm init --yes
```

Installing Feathers

Feathers can be installed like any other Node module by installing the @feathersjs/feathers package through npm. The same package can also be used with a module loader like Browserify or Webpack and React Native.

```
npm install @feathersjs/feathers --save
```

> **Note:** All Feathers core modules are in the `@feathersjs` npm namespace.

Your first app

The base of any Feathers application is the app object which can be created like this:

```
const feathers = require('@feathersjs/feathers');
const app = feathers();
```

This application object has several methods, most importantly it allows us to register services. We will learn more about services in the next chapter, for now let's register and use a simple service that has only a `get` method by creating an `app.js` file (in the current folder) like this:

```
const feathers = require('@feathersjs/feathers');
const app = feathers();

// Register a simple todo service that return the name and a text
app.use('todos', {
  async get(name) {
    // Return an object in the form of { name, text }
    return {
      name,
      text: `You have to do ${name}`
    };
  }
});

// A function that gets and logs a todo from the service
async function getTodo(name) {
  // Get the service we registered above
  const service = app.service('todos');
  // Call the `get` method with a name
  const todo = await service.get(name);
```

```
  // Log the todo we got back
  console.log(todo);
}

getTodo('dishes');
```

We can run it with

```
node app.js
```

And should see

```
{ name: 'dishes', text: 'You have to do dishes' }
```

> **Pro tip:** For more information about the Feathers application object see the Application API documentation.

In the browser

The Feathers application we created above can also run just the same in the browser. The easiest way to load Feathers here is through a `<script>` tag pointing to the CDN version of Feathers. Loading it will make a `feathers` global variable available.

Let's put the browser files into a new folder

```
mkdir public
```

We will also need to host the folder with a webserver. This can be done with any webserver like Apache or with a the http-server module that we can install and host the `public/` folder like this:

```
npm install http-server -g
http-server public/
```

> **Note:** You have to keep this server running for all browser examples in the basics guide tow work.

In the `public/` folder we add two files, an `index.html` that will load Feathers:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Feathers Basics</title>
</head>
<body>
  <h1>Welcome to Feathers</h1>
  <p>Open up the console in your browser.</p>
  <script type="text/javascript" src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
  <script src="client.js"></script>
</body>
</html>
```

And an `client.js` looking like this:

```
const app = feathers();

// Register a simple todo service that return the name and a text
app.use('todos', {
```

```
  async get(name) {
    // Return an object in the form of { name, text }
    return {
      name,
      text: `You have to do ${name}`
    };
  }
});

// A function that gets and logs a todo from the service
async function logTodo(name) {
  // Get the service we registered above
  const service = app.service('todos');
  // Call the `get` method with a name
  const todo = await service.get(name);

  // Log the todo we got back
  console.log(todo);
}

logTodo('dishes');
```

You may notice that it is pretty much the same as our `app.js` for Node except the missing `feathers` import (since it is alaready available as a global variable).

If you now go to localhost:8080 with the console open you will also see the result logged.

> **Note:** You can also load Feathers with a module loader like Webpack or Browserify. For more information see the client API chapter.

What's next?

In this chapter we created our first Feathers application with a simple service that works in Node and the browser. Next, let's learn more about Services and Service events.

Services

Services are the heart of every Feathers application and JavaScript objects or instances of a class that implement certain methods. Services provide a uniform, protocol independent interface for how to interact with any kind of data like:

- Reading and/or writing from a database
- Interacting with the file system
- Call another API
- Call other services like
  - Sending an email
  - Processing a payment
  - Returning the current weather for a location, etc.

Protocol independent means that to a Feathers service it does not matter if it has been called internally, through a REST API or websockets (both of which we will look at later) or any other way.

Service methods

Service methods are CRUD methods that a service object can implement. Feathers service methods are:

- `find` - Find all data (potentially matching a query)
- `get` - Get a single data entry by its unique identifier
- `create` - Create new data
- `update` - Update an existing data entry by completely replacing it
- `patch` - Update one or more data entries by merging with the new data
- `remove` - Remove one or more existing data entries

Below is an example of Feathers service interface as a normal object and a JavaScript class:

Object
Class

```
const myService = {
  async find(params) {
    return [];
  },
  async get(id, params) {},
  async create(data, params) {},
  async update(id, data, params) {},
  async patch(id, data, params) {},
  async remove(id, params) {}
}

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

```
class myService {
  async find(params) {
    return return [];
  }
  async get(id, params) {}
  async create(data, params) {}
  async update(id, data, params) {}
  async patch(id, data, params) {}
  async remove(id, params) {}
}
```

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

The parameters for service methods are:

- `id` - The unique identifier for the data
- `data` - The data sent by the user (for creating and updating)
- `params` (*optional*) - Additional parameters, for example the authenticated user or the query.

> **Note:** A service does not have to implement all those methods but must have at least one.

> **Pro tip:** For more information about service, service methods and parameters see the Service API documentation.

A messages service

Now that we know how service methods look like we can implement our own chat message service that allows us to find, create, remove and update messages in-memory. Here we will use a JavaScript class to work with our messages but as we've seen above it could als be a normal object.

Below is the complete updated `app.js` with comments:

```
const feathers = require('@feathersjs/feathers');

class Messages {
  constructor() {
    this.messages = [];
    this.currentId = 0;
  }

  async find(params) {
    // Return the list of all messages
    return this.messages;
  }

  async get(id, params) {
    // Find the message by id
    const message = this.messages.find(message => message.id === parseInt(id, 10));

    // Throw an error if it wasn't found
    if(!message) {
      throw new Error(`Message with id ${id} not found`);
    }

    // Otherwise return the message
    return message;
  }

  async create(data, params) {
    // Create a new object with the original data and an id
    // taken from the incrementing `currentId` counter
    const message = Object.assign({
      id: ++this.currentId
    }, data);

    this.messages.push(message);

    return message;
  }

  async patch(id, data, params) {
    // Get the existing message. Will throw an error if not found
    const message = await this.get(id);
```

```
    // Merge the existing message with the new data
    // and return the result
    return Object.assign(message, data);
  }

  async remove(id, params) {
    // Get the message by id (will throw an error if not found)
    const message = await this.get(id);
    // Find the index of the message in our message array
    const index = this.messages.indexOf(message);

    // Remove the found message from our array
    this.messages.splice(index, 1);

    // Return the removed message
    return message;
  }
}

const app = feathers();

// Initialize the messages service by creating
// a new instance of our class
app.use('messages', new Messages());
```

Using services

A service object can be registered on a Feathers application by calling `app.use(path, service)`. `path` will be the name of the service (and the URL if it is exposed as an API which we will learn later).

We can retrieve that service via `app.service(path)` and then call any of its service methods. Add the following to the end of `app.js`:

```
async function processMessages() {
  await app.service('messages').create({
    text: 'First message'
  });

  await app.service('messages').create({
    text: 'Second message'
  });

  const messageList = await app.service('messages').find();

  console.log('Available messages', messageList);
}

processMessages();
```

And run it with

```
node app.js
```

We should see this:

```
Available messages [ { id: 0, text: 'First message' },
  { id: 1, text: 'Second message' } ]
```

> **Important:** Always use the service returned by `app.service(path)` not the service object (what we called `messageService` above) directly. See the app.service API documentation for more information.

Service events

When you register a service it will automatically become a NodeJS EventEmitter that sends events with the new data when a service method that modifies data (`create` , `update` , `patch` and `remove`) returns. Events can be listened to with `app.service('messages').on('eventName', data => {})` . Here is a list of the service methods and their correspondingn events:

Service method	Service event
`service.create()`	`service.on('created')`
`service.update()`	`service.on('updated')`
`service.patch()`	`service.on('patched')`
`service.remove()`	`service.on('removed')`

We will see later that this is the key to how Feathers enables real-time functionality. For now, let's update the `processMessage` function in `app.js` as follows:

```
async function processMessages() {
  app.service('messages').on('created', message => {
    console.log('Created a new message', message);
  });

  app.service('messages').on('removed', message => {
    console.log('Deleted message', message);
  });

  await app.service('messages').create({
    text: 'First message'
  });

  const lastMessage = await app.service('messages').create({
    text: 'Second message'
  });

  // Remove the message we just created
  await app.service('messages').remove(lastMessage.id);

  const messageList = await app.service('messages').find();

  console.log('Available messages', messageList);
}

processMessages();
```

If we now run the file via

```
node app.js
```

We will see how the event handlers are logging the information of created and deleted message like this:

```
Created a new message { id: 0, text: 'First message' }
Created a new message { id: 1, text: 'Second message' }
Deleted message { id: 1, text: 'Second message' }
Available messages [ { id: 0, text: 'First message' } ]
```

What's next?

In this chapter we learned about services as Feathers core concept for abstracting data operations. We also saw how a service sends events which we will use later to create real-time applications. First, we will look at Feathers Hooks which is the other key part of how Feathers works.

Hooks

As we have seen the previous chapter, Feathers services are a great way to implement data storage and modification. Technically, we could implement all our application logic within services but very often an application requires similar functionality across multiple services. For example, we might want to check for all services if a user is allowed to even call the service method or add the current date to all data that we are saving. With just using services we would have to implement this every time again.

This is where Feathers hooks come in. Hooks are pluggable middleware functions that can be registered **before**, **after** or on **error**s of a service method. You can register a single hook function or create a chain of them to create complex work-flows.

Just like services themselves, hooks are *transport independent*. They are usually also service agnostic, meaning they can be used with *any* service. This pattern keeps your application logic flexible, composable, and much easier to trace through and debug.

> **Note:** A full overview of the hook API can be found in the hooks API documentation.

Hooks are commonly used to handle things like validation, authorization, logging, populating related entities, sending notifications and more.

> **Pro tip:** For more information about the design patterns behind hooks see this blog post.

Quick example

Here is a quick example for a hook that adds a `createdAt` property to the data before calling the actual `create` service method:

```
app.service('messages').hooks({
  before: {
    create: async context => {
      context.data.createdAt = new Date();

      return context;
    }
  }
})
```

Hook functions

A hook function is a function that takes the hook context as the parameter and returns that context or nothing. Hook functions run in the order they are registered and will only continue to the next once the current hook function completes. If a hook function throws an error, all remaining hooks (and possibly the service call) will be skipped and the error will be returned.

A common pattern to make hooks more re-usable (e.g. making the `createdAt` property name from the example above configurable) is to create a wrapper function that takes those options and returns a hook function:

```
const setTimestamp = name => {
  return async context => {
    context.data[name] = new Date();

    return context;
  }
```

```
}

app.service('messages').hooks({
  before: {
    create: setTimestamp('createdAt'),
    update: setTimestamp('updatedAt')
  }
});
```

Now we have a re-usable hook that can set the timestamp on any property.

Hook context

The hook `context` is an object which contains information about the service method call. It has read-only and writable properties. Read-only properties are:

- `context.app` - The Feathers application object
- `context.service` - The service this hook is currently running on
- `context.path` - The path of the service
- `context.method` - The service method
- `context.type` - The hook type (`before` , `after` or `error`)

Writeable properties are:

- `context.params` - The service method call `params` . For external calls, `params` usually contians:
  - `context.params.query` - The query (e.g. query string for REST) for the service call
  - `context.params.provider` - The name of the transport (which we will look at in the next chapter) the call has been made through. Usually `rest` , `socketio` , `primus` . Will be `undefined` for internal calls.
- `context.id` - The `id` for a `get` , `remove` , `update` and `patch` service method call
- `context.data` - The `data` sent by the user in a `create` , `update` and `patch` service method call
- `context.error` - The error that was thrown (in `error` hooks)
- `context.result` - The result of the service method call (in `after` hooks)

> **Note:** For more information about the hook context see the hooks API documentation.

Registering hooks

The most common way to register hooks is in an object like this:

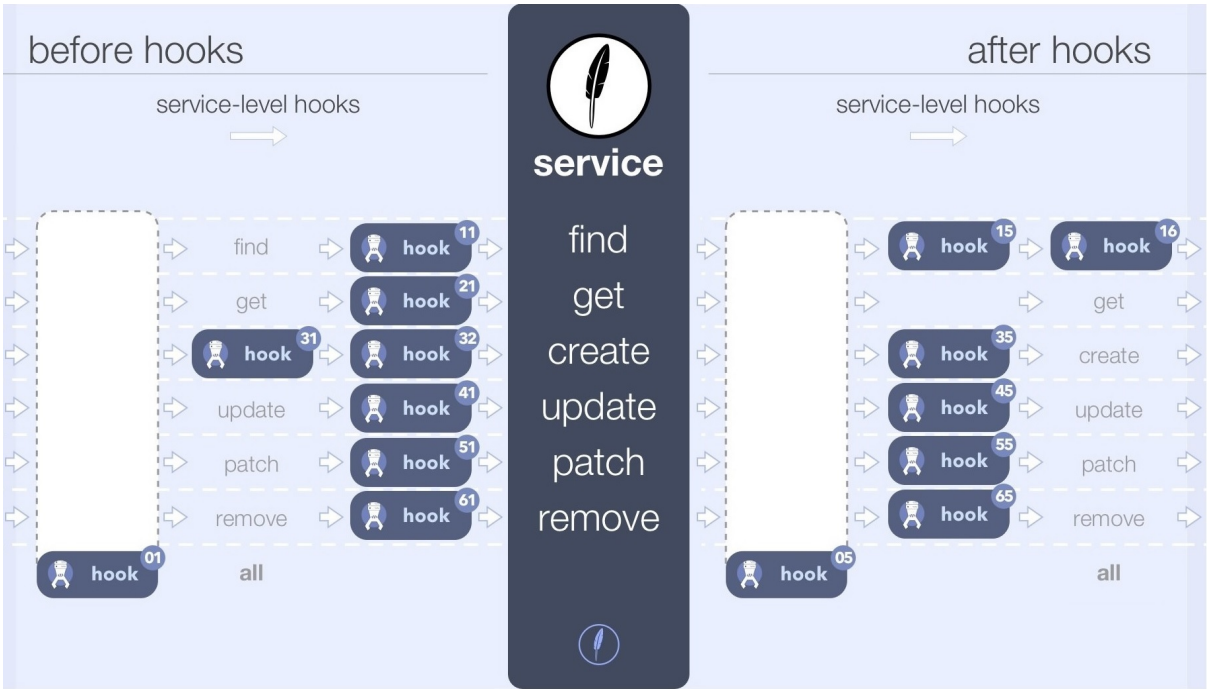
```
const messagesHooks = {
  before: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: [],
  },
  after: {
    all: [],
    find: [],
    create: [],
    update: [],
    patch: [],
    remove: [],
  }
};
```

```
app.service('messages').hooks(messagesHooks);
```

This makes it easy to see at one glance in which order hooks are executed and for which method.

> **Note:** `all` is a special keyword which means those hooks will run before the method specific hooks in this chain.

A flow how different hooks will be executed like this:



Can be registered like this:

```
const messagesHooks = {
  before: {
    all: [ hook01() ],
    find: [ hook11() ],
    get: [ hook21() ],
    create: [ hook31(), hook32() ],
    update: [ hook41() ],
    patch: [ hook51() ],
    remove: [ hook61() ],
  },
  after: {
    all: [ hook05() ],
    find: [ hook15(), hook16() ],
    create: [ hook35() ],
    update: [ hook45() ],
    patch: [ hook55() ],
    remove: [ hook65() ],
  }
};

app.service('messages').hooks(messagesHooks);
```

Validating data

If a hook throws an error, all following hooks will be skipped and the error will be returned to the user. This makes `before` hooks a great place to validate incoming data by throwing an error for invalid data. We can throw a normal JavaScript error or Feathers error which has some additional functionality (like returning the proper error code for REST calls). We will only need the hook for `create`, `update` and `patch` since those are the only service methods that allow user submitted data:

```
const { BadRequest } = require('@feathersjs/errors');

const validate = async context => {
  const { data } = context;

  // Check if there is `text` property
  if(!data.text) {
    throw new BadRequest('Message text must exist');
  }

  // Check if it is a string and not just whitespace
  if(typeof data.text !== 'string' || data.text.trim() === '') {
    throw new BadRequest('Message text is invalid');
  }

  // Change the data to be only the text
  // This prevents people from adding other properties to our database
  context.data = {
    text: data.text.toString()
  }

  return context;
};

app.service('messages').hooks({
  before: {
    create: validate,
    update: validate,
    patch: validate
  }
});
```

> **Note:** Throwing an appropriate Feathers errors allows to add more information and return the correc

Application hooks

Sometimes we want to automatically add a hook to every service in our Feathers application. This is what application hooks can be used for. They work the same as service specific hooks but run in a more specific order:

- `before` application hooks will always run *before* all service `before` hooks
- `after` application hooks will always run *after* all service `after` hooks
- `error` application hooks will always run *after* all service `error` hooks

Error logging

A good use for application hooks is to log any service method call error. The following example logs every service method error with the path and method name as well as the error stack:

```
app.hooks({
  error: async context => {
    console.error(`Error in '${context.path}' service method '${context.method}`, context.error.stack);
  }
});
```

More examples

The chat application guide will show several more examples like how to associate data and adding user information for hooks created by the generator.

What's next?

In this chapter we learned how Feathers hooks can be used middleware for service method calls to validate and manipulate incoming and outgoing data without having to change our service. In the next chapter we will turn our messages service into a fully functional REST API.

REST APIs

In the previous chapters we learned about Feathers services and hooks and created a messages service that works in NodeJS and the browser. We saw how Feathers automatically sends events but so far we didn't really create a web API that other people can use.

This what Feathers transports do. A transport is a plugin that turns a Feathers application into a server that exposes our services through different protocols for other clients to use. Since a transport involves running a server it won't work in the browser but we wil learn later that there are complementary plugins for connecting to a Feathers server in a browser Feathers application.

Currently Feathers officially has three transports:

- HTTP REST via Express for exposing services through a JSON REST API
- Socket.io for connecting to services through websockets and also receiving real-time service events
- Primus an alternative to Socket.io supporting several websocket protocols which also supports real-time events

In this chapter we will look at the HTTP REST transport and Feathers Express framework integration.

REST and services

One of the goals of Feathers is make building REST APIs easier since it is by far the most common protocol for web APIs. For example, we want to make a request like `GET /messages/1` and get a JSON response like `{ "id": 1, "text": "The first message" }`. You may already have noticed that the Feathers service methods and the HTTP methods like `GET`, `POST`, `PATCH` and `DELETE` are fairly complementary to each other:

Service method	HTTP method	Path
.find()	GET	/messages
.get()	GET	/messages/1
.create()	POST	/messages
.update()	PUT	/messages/1
.patch()	PATCH	/messages/1
.remove()	DELETE	/messages/1

What the Feathers REST transport essentially does is to automatically map our existing service methods to those endpoints.

Express integration

Express is probably the most popular Node framework for creating web applications and APIs. The Feathers Express integration allows us to turn a Feathers application into an application that is both, a Feathers application and a fully compatible Express application. This means you can use Feathers functionality like services but also any existing Express middleware. As mentioned before, the Express framework integration only works on the server.

To add the integration we install `@feathersjs/express`:

```
npm install @feathersjs/express --save
```

Then we can initialize a Feathers and Express application that exposes services as a REST API on port `3030` like this:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');

// This creates an app that is both, an Express and Feathers app
const app = express(feathers());

// Turn on JSON body parsing for REST services
app.use(express.json())
// Turn on URL-encoded body parsing for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport using Express
app.configure(express.rest());

// Set up an error handler that gives us nicer errors
app.use(express.errorHandler());

// Start the server on port 3030
app.listen(3030);
```

`express.json`, `express.urlencoded` and `express.errorHandler` is a normal Express middleware. We can still also use `app.use` to register a Feathers service though.

> **Pro tip:** You can find more information about the Express framework integration in the Express API chapter.

A messages REST API

The code above is really all we need to turn our messages service into a REST API. Here is the complete code for our `app.js` exposing the service from the services chapter through a REST API:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');

class Messages {
  constructor() {
    this.messages = [];
    this.currentId = 0;
  }

  async find(params) {
    // Return the list of all messages
    return this.messages;
  }

  async get(id, params) {
    // Find the message by id
    const message = this.messages.find(message => message.id === parseInt(id, 10));

    // Throw an error if it wasn't found
    if(!message) {
      throw new Error(`Message with id ${id} not found`);
    }

    // Otherwise return the message
    return message;
  }

  async create(data, params) {
    // Create a new object with the original data and an id
    // taken from the incrementing `currentId` counter
    const message = Object.assign({
```

```
      id: ++this.currentId
    }, data);

    this.messages.push(message);

    return message;
  }

  async patch(id, data, params) {
    // Get the existing message. Will throw an error if not found
    const message = await this.get(id);

    // Merge the existing message with the new data
    // and return the result
    return Object.assign(message, data);
  }

  async remove(id, params) {
    // Get the message by id (will throw an error if not found)
    const message = await this.get(id);
    // Find the index of the message in our message array
    const index = this.messages.indexOf(message);

    // Remove the found message from our array
    this.messages.splice(index, 1);

    // Return the removed message
    return message;
  }
}

const app = express(feathers());

// Turn on JSON body parsing for REST services
app.use(express.json())
// Turn on URL-encoded body parsing for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport using Express
app.configure(express.rest());

// Initialize the messages service by creating
// a new instance of our class
app.use('messages', new Messages());

// Set up an error handler that gives us nicer errors
app.use(express.errorHandler());

// Start the server on port 3030
const server = app.listen(3030);

// Use the service to create a new message on the server
app.service('messages').create({
  text: 'Hello from the server'
});

server.on('listening', () => console.log('Feathers REST API started at localhost:3030'));
```

You can start the server by running

```
node app.js
```

> **Note:** The server will stay running until you stop it by pressing Control + C in the terminal. Remember to stop and start the server every time `app.js` changes.

> **Important:** In Express an error handler, here `app.use(express.errorHandler());`, always has to be the last line before starting the server.

Using the API

Once the server is running the first thing we can do is hit localhost:3030/messages in the browser. Since we already created a message on the server, the JSON repsonse will look like this:

```
[{"id":0,"text":"Hello from the server"}]
```

We can also retrieve that specific message by going to localhost:3030/messages/1.

> **Pro Tip:** A browser plugin like JSON viewer for Chrome makes it much nicer to view JSON responses.

New messages can now be created by sending a POST request with JSON data to the same URL. Using CURL on the command line like this:

```
curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --data-binary '{ "text": "Hello from
 the command line!" }'
```

> **Note:** You can also use tools like Postman to make HTTP requests.

If you now refresh localhost:3030/messages you will see the newly created message.

We can also remove a message by sending a `DELETE` to its URL:

```
curl -X "DELETE" http://localhost:3030/messages/1
```

What's next?

In this chapter we built a fully functional messages REST API. You can probably already imagine how our messages service could store its data in a database instead of the `messages` array. In the next chapter, let's look at some services that implement different databases allowing us to create those APIs with even less code!

Databases

In the services chapter we created a custom in-memory messages service that can create, update and delete messages. You can probably imagine how we could implement the same thing talking to any database instad of storing it in memory so there isn't really a database that Feathers doesn't support.

Writing all that code yourself is pretty repetitive and cumbersome though which is why Feathers has a collection of pre-built services for different databases. They offer most the basic functionality and can always be fully customized to your requirements using hooks. Feathers database adapters support a common usage API, pagination and querying syntax for many popular databases and NodeJS ORMs:

Database	Adapter
In memory	feathers-memory, feathers-nedb
Localstorage, AsyncStorage	feathers-localstorage
Filesystem	feathers-nedb
MongoDB	feathers-mongodb, feathers-mongoose
MySQL, PostgreSQL, MariaDB, SQLite, MSSQL	feathers-knex, feathers-sequelize
Elasticsearch	feathers-elasticsearch
RethinkDB	feathers-rethinkdb

> **Pro tip:** Each one of the the linked adapters has a complete REST API example in their readme.

In this chapter we will look at the basic usage of the in-memory database adapter and create a persistent REST API using NEDB.

> **Important:** You should be familiar with the database technology and ORM (Sequelize, KnexJS or Mongoose) before using a Feathers database adapter.

An in-memory database

feathers-memory is a Feathers database adapter that - similar to our messages service - stores its data in memory. We will use it for the examples because it also works in the browser. Let's install it with:

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

We can use the adapter by requiring it and initializing it with the options we want. Here we enable pagination showing 10 entries by default and a maximum of 25 (so that clients can't just request all data at once and crash our server):

```
const feathers = require('@feathersjs/feathers');
const memory = require('feathers-memory');

const app = feathers();

app.use('messages', memory({
  paginate: {
    default: 10,
    max: 25
  }
}));
```

That's it. We have a complete CRUD service for our messages with querying functionality.

In the browser

We can also include `feathers-memory` in the browser, most easily by loading its browser build which will add it as `feathers.memory`. In `public/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Feathers Basics</title>
</head>
<body>
  <h1>Welcome to Feathers</h1>
  <p>Open up the console in your browser.</p>
  <script type="text/javascript" src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
  <script type="text/javascript" src="//unpkg.com/feathers-memory@^2.0.0/dist/feathers-memory.js"></script>
  <script src="client.js"></script>
</body>
</html>
```

And `public/client.js`:

```
const app = feathers();

app.use('messages', feathers.memory({
  paginate: {
    default: 10,
    max: 25
  }
}));
```

Querying

As mentioned, all database adapters support a common way of querying the data in a `find` method call using `params.query`. You can find a complete list in the querying syntax API.

With pagination enabled, the `find` method will return an object with the following properties:

- `data` - The current list of data
- `limit` - The page size
- `skip` - The number of entries that were skipped
- `total` - The total number of entries for this query

The following example automatically creates a couple of messages and makes some queries. You can add it at the end of both, `app.js` and `public/client.js` to see it in Node and the browser:

```
async function createAndFind() {
  // Stores a reference to the messages service so we don't have to call it all the time
  const messages = app.service('messages');

  for(let counter = 0; counter < 100; counter++) {
    await messages.create({
      counter,
      message: `Message number ${counter}`
    });
  }
```

```
  // We show 10 entries by default. By skipping 10 we go to page 2
  const page2 = await messages.find({
    query: { $skip: 10 }
  });

  console.log('Page number 2', page2);

  // Show 20 items per page
  const largePage = await messages.find({
    query: { $limit: 20 }
  });

  console.log('20 items', largePage);

  // Find the first 10 items with counter greater 50 and less than 70
  const counterList = await messages.find({
    query: {
      counter: { $gt: 50, $lt: 70 }
    }
  });

  console.log('Counter greater 50 and less than 70', counterList);

  // Find all entries with text "Message number 20"
  const message20 = await messages.find({
    query: {
      message: 'Message number 20'
    }
  });

  console.log('Entries with text "Message number 20"', message20);
}

createAndFind();
```

As a REST API

In the REST API chapter we created a REST API from our custom messages service. Using a database adapter instead will make our `app.js` a lot shorter:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const memory = require('feathers-memory');

const app = express(feathers());

// Turn on JSON body parsing for REST services
app.use(express.json())
// Turn on URL-encoded body parsing for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport using Express
app.configure(express.rest());
// Set up an error handler that gives us nicer errors
app.use(express.errorHandler());

// Initialize the messages service
app.use('messages', memory({
  paginate: {
    default: 10,
    max: 25
  }
}));

// Start the server on port 3030
```

```
const server = app.listen(3030);

// Use the service to create a new message on the server
app.service('messages').create({
  text: 'Hello from the server'
});

server.on('listening', () => console.log('Feathers REST API started at localhost:3030'));
```

After starting the server with `node app.js` we can pass a query e.g. by going to localhost:3030/messages?$limit=2.

> **Note:** The querying syntax API documentation has more examples how URLs should look like.

What's next?

Feathers database adapters are a quick way to get an API and up and running. The great thing is that hooks still give us all the flexibility we need to customize how they work. We already saw how we can easily create a database backed REST API, in the next chapter we will make our API real-time.

Real-time APIs

In the services chapter we saw that Feathers services automatically send `created`, `updated`, `patched` and `removed` events when a `create`, `update`, `patch` or `remove` service method returns. Real-time means that those events are also published to connected clients so that they can react accordingly, e.g. update their UI.

To allow real-time communication with clients we need a transport that supports bi-directional communication. In Feathers those are the Socket.io and Primus transport both of which use websockets to receive real-time events *and also* call service methods.

> **Important:** The REST transport does not support real-time updates. Since socket transports also allow to call service methods and generally perform better, we recommend using a real-time transport whenever possible.

In this chapter we will use Socket.io and create a database backed real-time API that also still supports a REST endpoint.

Using the transport

After installing

```
npm install @feathersjs/socketio --save
```

The Socket.io transport can be configured and used with a standard configuration like this:

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio');

// Create a Feathers application
const app = feathers();

// Configure the Socket.io transport
app.configure(socketio());

// Start the server on port 3030
app.listen(3030);
```

It also works in combination with a REST API setup:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const socketio = require('@feathersjs/socketio');

// This creates an app that is both, an Express and Feathers app
const app = express(feathers());

// Turn on JSON body parsing for REST services
app.use(express.json())
// Turn on URL-encoded body parsing for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport using Express
app.configure(express.rest());
// Set up an error handler that gives us nicer errors
app.use(express.errorHandler());

// Configure the Socket.io transport
app.configure(socketio());
```

```
// Start the server on port 3030
app.listen(3030);
```

Channels

Channels determine which real-time events should be sent to which client. For example, we want to only send messages to authenticated users or users in the same room. For this example we will just enable real-time functionality for all connections however:

```
// On any real-time connection, add it to the `everybody` channel
app.on('connection', connection => app.channel('everybody').join(connection));

// Publish all events to the `everybody` channel
app.publish(() => app.channel('everybody'));
```

> **Note:** More information about channels can be found in the channel API documentation.

A messages API

Putting it all together, our REST and real-time API with a messages service `app.js` looks like this:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const socketio = require('@feathersjs/socketio');
const memory = require('feathers-memory');

// This creates an app that is both, an Express and Feathers app
const app = express(feathers());

// Turn on JSON body parsing for REST services
app.use(express.json())
// Turn on URL-encoded body parsing for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport using Express
app.configure(express.rest());

// Configure the Socket.io transport
app.configure(socketio());

// On any real-time connection, add it to the `everybody` channel
app.on('connection', connection => app.channel('everybody').join(connection));

// Publish all events to the `everybody` channel
app.publish(() => app.channel('everybody'));

// Initialize the messages service
app.use('messages', memory({
  paginate: {
    default: 10,
    max: 25
  }
}));

// Set up an error handler that gives us nicer errors
app.use(express.errorHandler());

// Start the server on port 3030
const server = app.listen(3030);

server.on('listening', () => console.log('Feathers API started at localhost:3030'));
```

As always, we can start our server again by running

```
node app.js
```

Using the API

The real-time API can be used by establishing a websocket connection. For that we need the Socket.io client which we can included by updating `public/index.html` to:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Feathers Basics</title>
</head>
<body>
  <h1>Welcome to Feathers</h1>
  <p>Open up the console in your browser.</p>
  <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/socket.io/2.0.4/socket.io.js"></script>
  <script type="text/javascript" src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
  <script type="text/javascript" src="//unpkg.com/feathers-memory@^2.0.0/dist/feathers-memory.js"></script>
  <script src="client.js"></script>
</body>
</html>
```

Then we can initialize and use the socket directly making some calls and listening to real-time events by updating `public/client.js` to this:

```
// Create a websocket connecting to our Feathers server
const socket = io('http://localhost:3030');

// Listen to new messages being created
socket.on('messages created', message =>
  console.log('Someone created a message', message)
);

socket.emit('create', 'messages', {
  text: 'Hello from socket'
}, (error, result) => {
  socket.emit('find', 'messages', (error, messageList) => {
    console.log('Current messages', messageList);
  });
});
```

Client use

So far, we have seen that Feathers with its services, events and hooks can also be used in the browser which is a very unique feature. By implementing custom services that talk to an API in the browser Feathers allows us to structure any client side application with any framework.

This is exactly what Feathers client side services do. In order to connect to a Feathers server, a client creates Services that use a REST or websocket connection to relay method calls and allow listening to events from the server. This means that we can use a client side Feathers application to transparently talk to a Feathers server the same way as if we would use it locally (like we did in all the previous examples)!

> **Note:** The following examples show the use of the Feathers client through a `<script>` tag. For more information on using a module loader like Webpack or Browserify and loading individual modules see the client API documentaiton.

Real-time client

In the real-time chapter we saw an example of how to directly use a websocket connection to make service calls an listen to events. We can also use a browser Feathers application and client services that use those conneciton. Let's update `public/client.js` to:

```
// Create a websocket connecting to our Feathers server
const socket = io('http://localhost:3030');
// Create a Feathers application
const app = feathers();
// Configure Socket.io client services to use that socket
app.configure(feathers.socketio(socket));

app.service('messages').on('created', message => {
  console.log('Someone created a message', message);
});

async function createAndList() {
  await app.service('messages').create({
    text: 'Hello from Feathers browser client'
  });

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

  console.log('Messages', messages);
}

createAndList();
```

REST client

We can also create services that communicate via REST using many different Ajax libraries like jQuery or Axios. For this example we will be using fetch since it is already built into the browser.

> **Important:** REST services do emit real-time events but only locally to themselves. REST does not support real-time updates from the server.

Since we a making a cross-domain request, we first have to enable Cross-Origin Resource sharing (CORS) on the server by updating `app.js` to:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const socketio = require('@feathersjs/socketio');
const memory = require('feathers-memory');

// This creates an app that is both, an Express and Feathers app
const app = express(feathers());

// This enables CORS
app.use(function(req, res, next) {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
  next();
});

// Turn on JSON body parsing for REST services
app.use(express.json())
// Turn on URL-encoded body parsing for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport using Express
app.configure(express.rest());

// Configure the Socket.io transport
app.configure(socketio());

// On any real-time connection, add it to the `everybody` channel
app.on('connection', connection => app.channel('everybody').join(connection));

// Publish all events to the `everybody` channel
app.publish(() => app.channel('everybody'));

// Initialize the messages service
app.use('messages', memory({
  paginate: {
    default: 10,
    max: 25
  }
}));

// Set up an error handler that gives us nicer errors
app.use(express.errorHandler());

// Start the server on port 3030
const server = app.listen(3030);

server.on('listening', () => console.log('Feathers API started at localhost:3030'));
```

> **Note:** This is just a basic middleware setting the headers. In production (and applications created by the Feathers generator) we will use the cors module.

Then we can update `public/client.js` to:

```
// Create a Feathers application
const app = feathers();
// Initialize a REST connection
const rest = feathers.rest('http://localhost:3030');
// Configure the REST client by using `window.fetch`
app.configure(rest.fetch(window.fetch));

app.service('messages').on('created', message => {
  console.log('Created a new message locally', message);
});

async function createAndList() {
  await app.service('messages').create({
    text: 'Hello from Feathers browser client'
  });
```

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

  console.log('Messages', messages);
}

createAndList();
```

What's next?

In this chapter we learned how to transparently connect to another Feathers server and use its services just the same as we are used to. In the last chapter let's briefly look at the Feathers generator (CLI) and the patterns it uses to structure an application before jumping into building a full chat application.

The Feathers generator (CLI)

Until now we wrote code by hand in a single file to get a better understanding how Feathers itself works. The Feathers CLI allows us to initialize a new Feathers application with a recommended structure. It also helps with

- Configuring authentication
- Generating database backed services
- Setting up database connections
- Generating hooks (with tests)
- Adding Express middleware

In this chapter we will look at installing the CLI and common patterns the generator uses to structure our server application. Further use of the CLI will be discussed in the chat application guide.

Installing the CLI

The CLI should be installed globally via npm:

```
npm install @feathersjs/cli -g
```

Once successful we should now have the `feathers` command available on the command line which we can check with:

```
feathers --version
```

Which should show a version of `3.3.0` or later.

Configure functions

The most common pattern used in the generated application is *configure functions* which are functions that take the Feathers app object and then use it to e.g. register services. Those functions are then passed to app.configure.

Let's look at our basic database example:

```
const feathers = require('@feathersjs/feathers');
const memory = require('feathers-memory');

const app = feathers();

app.use('messages', memory({
  paginate: {
    default: 10,
    max: 25
  }
}));
```

Which could be split up using a configure function like this:

```
const feathers = require('@feathersjs/feathers');
const memory = require('feathers-memory');

const configureMessages = function(app) {
```

```
  app.use('messages', memory({
    paginate: {
      default: 10,
      max: 25
    }
  }));
};

const app = feathers();

app.configure(configureMessages);
```

Now we can move that function into a separate file like `messages.service.js` and set it as the default module export for that file:

```
module.exports = function(app) {
  app.use('messages', memory({
    paginate: {
      default: 10,
      max: 25
    }
  }));
};
```

And then import it into `app.js` and use it:

```
const feathers = require('@feathersjs/feathers');
const memory = require('feathers-memory');
const configureMessages = require('./messages.service.js');

const app = feathers();

app.configure(configureMessages);
```

This is the most common pattern how the generators split things up into separate files and any documentation example that uses the `app` object can be used in a configure function. You can create your own files that export a configure function and `require` and `app.configure` them in `app.js`

> **Note:** Keep in mind that the order in which configure functions are called might matter, e.g. if it is using a service, that service has to be registered first.

Hook functions

We already saw in the hooks guide how we can create a wrapper function that allows to customize the options of a hook with the `setTimestamp` example:

```
const setTimestamp = name => {
  return async context => {
    context.data[name] = new Date();

    return context;
  }
}

app.service('messages').hooks({
  before: {
    create: setTimestamp('createdAt'),
    update: setTimestamp('updatedAt')
  }
});
```

This is also the pattern the hook generator uses but in its own file like `hooks/set-timestamp.js` which could look like this:

```
module.exports = ({ name }) => {
  return async context => {
    context.data[name] = new Date();

    return context;
  }
}
```

Now we can use that hook like this:

```
const setTimestamp = require('./hooks/set-timestamp.js');

app.service('messages').hooks({
  before: {
    create: setTimestamp({ name: 'createdAt' }),
    update: setTimestamp({ name: 'updatedAt' })
  }
});
```

> **Note:** We are using an options object here which allows us to more easily add new options than function parameters.

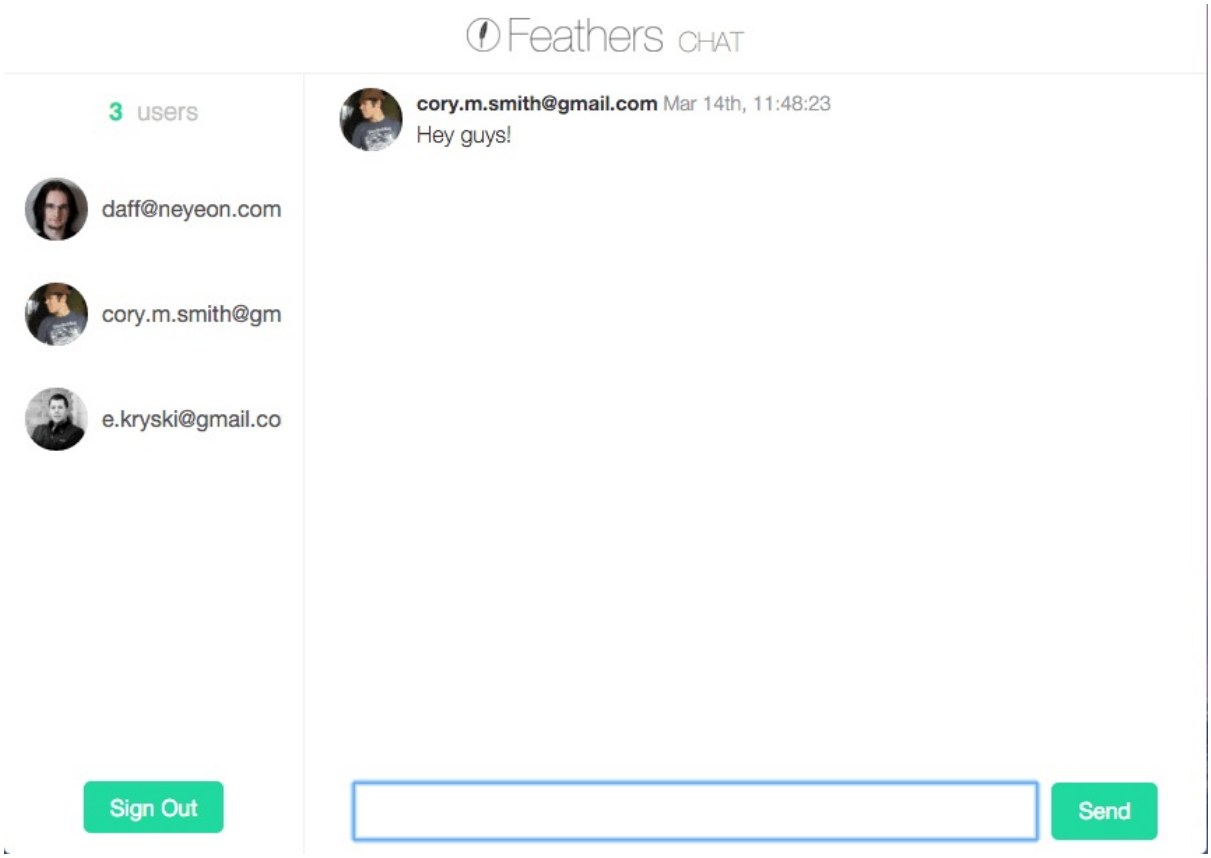
What's next?

In this chapter we installed the Feathers CLI (and generator) and looked at patterns that are used in structuring the generated application. Now we can use the generator to build a full chat application complete with authentication and a JavaScript frontend!

Creating a Chat Application

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 and authorization all with a RESTful and real-time API. You can go through this guide right away but we do recommend to get a basic understanding of Feathers in the basics guide and the generated application in the generator guide first.



You can find a complete working example here.

Creating the application

Create a new application using the generator.

Generating a service

Add an API endpoint to store messages.

Adding Authentication

Add user registration and login.

Processing data

Sanitize and process user data.

Building a frontend

Learn how to use Feathers in the browser by creating a small real-time chat frontend.

Creating the application

In this part we are going to create a new Feathers application using the generator. We can install the generator via:

```
npm install @feathersjs/cli -g
```

Generating the application

With everything set up let's create a directory for our new app:

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

Now we can generate the application:

```
$ feathers generate app
```

When presented with the project name just hit enter, or enter a name (no spaces).

Next, enter in a short description of your application.

The next prompt asking for the source folder can be answered by just hitting enter. This will put all source files into the `src/` folder.

The next prompt will ask for the package manager you want to use. The default is the standard npm.

> **Note:** If you choose Yarn instead, make sure it has been installed via `npm install yarn -g` first.

You're now presented with the option to choose which transport you want to support. Since we're setting up a real-time and REST API we'll go with the default REST and Socket.io options. So just hit enter.

Once you confirm the final prompt you will see something like this:

```
[? Project name feathers-chat
[? Description A Feathers chat application
[? What folder should the source files live in? src
 ? Which package manager are you using (has to be installed globally)? npm
 ? What type of API are you making? REST, Realtime via Socket.io
   create package.json
   create config/default.json
   create LICENSE
   create public/favicon.ico
   create public/index.html
   create .editorconfig
   create .eslintrc.json
   create src/app.hooks.js
   create src/channels.js
   create src/hooks/logger.js
   create src/index.js
   create src/middleware/index.js
   create src/services/index.js
   create .gitignore
   create README.md
   create src/app.js
   create test/app.test.js
   create config/production.json
```

The generated files

Let's have a brief look at the files that have been generated:

- `config/` - Contains the configuration files for the app. `production.json` files override `default.json` when in production mode by setting `NODE_ENV=production`. For more information see the configuration API documentation.
- `node_modules/` - The generator installs the project dependencies either using npm, or yarn. The dependencies are also added in the `package.json`.
- `public/` - Contains static files to be served. A sample favicon and `index.html` (which will show up when going directly to the server URL) are already included.
- `src/` - Contains the Feathers server code.
  - `hooks/` contains our custom hooks. A simple `logger` hook for logging debug information about our service calls is already included
  - `middleware/` contains any Express middleware
  - `services/` will contain our services
  - `index.js` is used to load and start the application
  - `app.js` configures our Feathers application
  - `app.hooks.js` contains hooks which that run for all services.
  - `channels.js` sets up Feathers event channels
  - `test/` - Contains Mocha test files for the app, hooks and services
  - `app.test.js` tests that the index page appears, as well as 404 errors for HTML pages and JSON
- `.editorconfig` is an EditorConfig setting which and helps developers define and maintain consistent coding styles among different editors and IDEs.
- `.eslintrc.json` contains defaults for liniting your code with ESLint.
- `.gitignore` - specifies intentionally untracked files which git, GitHub and other similar projects ignore.
- `.npmignore` specifies files which are not to be published for distribution.
- `LICENSE` - contains the License so that people know how they are permitted to use it, and any restrictions you're placing on it. It defaults to the Feathers license.
- `package.json` contains information about our project which npm, yarn and other package managers need to install and use your package.

Running the server and tests

The server can now be started by running

```
npm start
```

After that, you can see a welcome page at localhost:3030. When making modifications, remember to stop (CTRL + C) and start the server again.

The app also comes with a set of basic tests which can be run with

```
npm test
```

What's next?

We scaffolded a new Feathers application. The next step is to create a service for messages.

Creating a service

Now that we have our Feathers application generated we can create a new API endpoint to store messages.

Generating a service

In Feathers any API endpoint is represented as a service which we already learned about in the basics guide. To generate a new service we can run

```
feathers generate service
```

First we have to choose what kind of service we would like to create. You can choose between many databases and ORMs but for this guide we will go with the default NeDB. NeDB is a database that stores its data locally in a file and requires no additional configuration or a database server running.

Next we are asked for the name of the service which we can answer with `messages` and then can answer the next question for the path with the default (`/messages`) by pressing enter.

The database connection string (in the case of NeDB the name of the path where it should store its database files) can also be answered with the default.

Confirming the last prompt will create a couple of files and wire our service up:

```
? What kind of service is it? NeDB
[? What is the name of the service? messages
[? Which path should the service be registered on? /messages
[? What is the database connection string? nedb://../data
     force config/default.json
    create src/services/messages/messages.service.js
     force src/services/index.js
    create src/models/messages.model.js
    create src/services/messages/messages.hooks.js
    create test/services/messages.test.js
```

Et voilà! We have a fully functional REST and real-time API for our messages.

The generated files

As we can see, several files were created:

- `src/services/messages/messages.service.js` - The service setup file which registers the service in a configure function
- `src/services/messages/messages.hooks.js` - A file that returns an object with all hooks that should be registered on the service.
- `src/models/messages.model.js` - The model for our messages. Since we are using NeDB this will just instantiate the filesystem database.
- `test/services/messages.test.js` - A Mocha test for the service which by default just makes sure that it exists.

Testing the API

If we now start our API with

```
npm start
```

We can go to localhost:3030/messages and will see an (empty) response from our new messages service.

We can also `POST` new messages and `PUT`, `PATCH` and `DELETE` existing messages (via `/messages/<_id>`), for example from the command line using CURL:

```
curl 'http://localhost:3030/messages/' -H 'Content-Type: application/json' --data-binary '{ "name": "Curler", "
text": "Hello from the command line!" }'
```

Or with a REST client, e.g. Postman using this button:

If we now go to localhost:3030/messages again we will see the newly created message(s).

What's next?

With just one command, we created a fully functional REST and real-time API endpoint. Next, let's add authentication and make sure messages only go to users that are allowed to see them.

Adding authentication

In the previous chapters we created our Feathers chat application and initialized a service for storing messages. For a proper chat application we need to be able to register and authenticate users.

Generating authentication

To add authentication to our application we can run

```
feathers generate authentication
```

This will first ask us which authentication providers we would like to use. In this guide we will only cover local authentication which is already selected so we can just confirm by pressing enter.

Next we have to define the service we would like to use to store user information. Here we can just confirm the default `users` and the database with the default NeDB:

```
? What authentication providers do you want to use? Other PassportJS strategies not in this list can still be con
figured manually. Username + Password (Local)
? What is the name of the user (entity) service? users
? What kind of service is it? NeDB
   force config/default.json
  create src/authentication.js
   force src/app.js
  create src/services/users/users.service.js
   force src/services/index.js
  create src/models/users.model.js
  create src/services/users/users.hooks.js
  create test/services/users.test.js
```

> **Note:** For more information on Feathers authentication see the authentication server API documentaion.

Creating a user and logging in

We just created a `users` service and enabled local authentication. When restarting the application we can now create a new user with `email` and `password` similar to what we did with messages and then use the login information to get a JWT (for more information see the How JWT works guide).

Creating the user

We will create a new user with the following data:

```
{
  "email": "feathers@example.com",
  "password": "secret"
}
```

The generated user service will automatically securely hash the password in the database for us and also exclude it from the response (passwords should never be transmitted). There are several ways to create a new user, for example via CURL like this:

```
curl 'http://localhost:3030/users/' -H 'Content-Type: application/json' --data-binary '{ "email": "feathers@exa
mple.com", "password": "secret" }'
```

With a REST client, e.g. Postman using this button:

> **Note:** Creating a user with the same email address will only work once and fail when it already exists in the database. This is a restriction implemented for NeDB and might have to be implemented manually when using a different database.

Getting a token

To create a JWT we can now post the login information with the strategy we want to use (`local`) to the `authentication` service:

```
{
  "strategy": "local",
  "email": "feathers@example.com",
  "password": "secret"
}
```

Via CURL:

```
curl 'http://localhost:3030/authentication/' -H 'Content-Type: application/json' --data-binary '{ "strategy": "
local", "email": "feathers@example.com", "password": "secret" }'
```

With a REST client, e.g. Postman:

The returned token can then be used to authenticate the user it was created for by adding it to the `Authorization` header of new HTTP requests.

Securing the messages service

Now we have to restrict our messages service to authenticated users. If we run `feathers generate authentication` *before* generating other services it will ask if the service should be restricted to authenticated users. Because we created the messages service first, however we have to update `src/services/messages/messages.hooks.js` manually to look like this:

```
const { authenticate } = require('@feathersjs/authentication').hooks;

module.exports = {
  before: {
    all: [ authenticate('jwt') ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },

  after: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
```

```
  },

  error: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  }
};
```

This will now only allow users with a valid JWT to access the service and automatically set `params.user`.

Securing real-time events

The `authenticate` hook that we used above will restrict *access* to service methods to only authenticated users. No we also have to make sure that real-time service events are only sent to connections that are allowed to see them. Feathers uses channels to accomplish that which the generator already sets up for us in `src/channels.js` (have a look at the comments in the generated file and the channel API documentation to get a better idea about channels).

We could use channels to e.g. only send events to users in a specific room or with specific permissions. To make things easier for our basic chat however, let's just send all events to all authenticated users by updating `src/channels.js` to this:

```
module.exports = function(app) {
  if(typeof app.channel !== 'function') {
    // If no real-time functionality has been configured just return
    return;
  }

  app.on('connection', connection => {
    // On a new real-time connection, add it to the anonymous channel
    app.channel('anonymous').join(connection);
  });

  app.on('login', (authResult, { connection }) => {
    // connection can be undefined if there is no
    // real-time connection, e.g. when logging in via REST
    if(connection) {
      // Obtain the logged in user from the connection
      // const user = connection.user;

      // The connection is no longer anonymous, remove it
      app.channel('anonymous').leave(connection);

      // Add it to the authenticated user channel
      app.channel('authenticated').join(connection);
    }
  });

  app.publish((data, hook) => { // eslint-disable-line no-unused-vars
    // Publish all service events to all authenticated users
    return app.channel('authenticated');
  });
};
```

This is almost the same as the original file except for the line `return app.channel('authenticated');` being commented in `app.publish()`. Now only authenticated users will receive real-time updates.

What's next?

In this chapter we initialized authentication and created a user and JWT. We secured the messages service and made sure that only authenticated users get real-time updates. We can now use that user information to process new message data.

Processing data

Now that we can create and authenticate users, we are going to process data, sanitize the input we get from the client and add additional information.

Sanitizing new message

When creating a new message, we automatically sanitized our input, add the user that sent it and include the date the message has been created before saving it in the database. This is where hooks come into play. In our case specifically a *before* hook. To create a new hook we can run:

```
feathers generate hook
```

The hook we want to create will be called `process-message`. Since we want to pre-process our data, the next prompt asking for what kind of hook, we will choose `before` from the list (and confirming enter).

Next we will see a list of all our services we can add this hook to. For this hook we will only choose the `messages` service (navigate to the entry with the arrow keys and select it with the space key).

A hook can run before any number of service methods. For this specific hook we will only select `create`. After confirming the last prompt we will see something like this:

```
[? What is the name of the hook? process-message
? What kind of hook should it be? before
? What service(s) should this hook be for (select none to add it yourself)?
 messages
? What methods should the hook be for (select none to add it yourself)? create
   create src/hooks/process-message.js
    force src/services/messages/messages.hooks.js
   create test/hooks/process-message.test.js
```

This will create our hook and wire it up to the service we selected. Now it is time to add some code. Update `src/hooks/process-message.js` to look like this:

```
// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/api/hooks.html

module.exports = function (options = {}) { // eslint-disable-line no-unused-vars
  return async context => {
    const { data } = context;

    // Throw an error if we didn't get a text
    if(!data.text) {
      throw new Error('A message must have a text');
    }

    // The authenticated user
    const user = context.params.user;
    // The actual message text
    const text = context.data.text
      // Messages can't be longer than 400 characters
      .substring(0, 400);

    // Override the original data (so that people can't submit additional stuff)
    context.data = {
      text,
      // Set the user id
      userId: user._id,
```

```
      // Add the current date
      createdAt: new Date().getTime()
    };

    // Best practise, hooks should always return the context
    return context;
  };
};
```

This will do several things:

1. Check if there is a `text` in the data and throw an error if not
2. Truncate the messages `text` property to 400 characters
3. Update the data submitted to the database to contain
   - The new truncated text
   - The currently authenticated user (so we always know who sent it)
   - The current (creation) date

Adding a user avatar

Let's create another more hook that adds a link to the Gravatar image of the users email address so we can show an avatar. After running

```
feathers generate hook
```

The selections are almost the same as our previous hook:

- The hook will be called `gravatar`
- It will be a `before` hook
- On the `users` service
- For the `create` method

```
[? What is the name of the hook? gravatar
? What kind of hook should it be? before
? What service(s) should this hook be for (select none to add it yourself)?
 messages
? What methods should the hook be for (select none to add it yourself)? create
   create src/hooks/gravatar.js
    force src/services/messages/messages.hooks.js
   create test/hooks/gravatar.test.js
```

Then we update `src/hooks/gravatar.js` with the following code:

```
// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/api/hooks.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';

module.exports = function (options = {}) { // eslint-disable-line no-unused-vars
  return async context => {
    // The user email
    const { email } = context.data;
    // Gravatar uses MD5 hashes from an email address to get the image
```

```
    const hash = crypto.createHash('md5').update(email).digest('hex');

    context.data.avatar = `${gravatarUrl}/${hash}?${query}`;

    // Best practise, hooks should always return the context
    return context;
  };
};
```

Here we use Node's crypto library to create an MD5 hash of the users email address. This is what Gravatar uses as the URL for the avatar of an email address. If we now create a new user it will add the link to the image in the `gravatar` property.

Populating the message sender

In the `process-message` hook we are currently just adding the users `_id` as the `userId` property in the message. We want to show more than the `_id` in the UI, so we'll need to populate more data in the message response. In order to show the right user information we want to include that information in our messages.

For that we create another hook:

- The hook will be called `populate-user`
- It will be an `after` hook
- On the `messages` service
- For `all` methods

```
[? What is the name of the hook? populate-user
? What kind of hook should it be? after
? What service(s) should this hook be for (select none to add it yourself)?
 messages
? What methods should the hook be for (select none to add it yourself)? all
   create src/hooks/populate-user.js
    force src/services/messages/messages.hooks.js
   create test/hooks/populate-user.test.js
```

Once created we can update `src/hooks/populate-user.js` to:

```
// Use this hook to manipulate incoming or outgoing data.
// For more information on hooks see: http://docs.feathersjs.com/api/hooks.html

module.exports = function (options = {}) { // eslint-disable-line no-unused-vars
  return async context => {
    // Get `app`, `method`, `params` and `result` from the hook context
    const { app, method, result, params } = context;

    // Make sure that we always have a list of messages either by wrapping
    // a single message into an array or by getting the `data` from the `find` method result
    const messages = method === 'find' ? result.data : [ result ];

    // Asynchronously get user object from each messages `userId`
    // and add it to the message
    await Promise.all(messages.map(async message => {
      // We'll also pass the original `params` to the service call
      // so that it has the same information available (e.g. who is requesting it)
      const user = await app.service('users').get(message.userId, params);

      message.user = user;
    }));

    // Best practise, hooks should always return the context
    return context;
```

```
  };
};
```

> **Note:** `Promise.all` makes sure that all the calls run in parallel instead of waiting for each one to finish.

What's next?

In this section we added three hooks to pre- and postprocess our message and user data. We now have a complete API to send and retrieve messages including authentication.

See the frameworks section for more resources on specific frameworks like React, React Native, Angular or VueJS. You'll find guides for creating a complete chat frontend with signup, logging, user listing and messages. There are also links to full example chat applications built with some popular frontend frameworks.

You can also browse the API which has a lot of information on the usage of Feathers and its database adaptors.

Building a frontend

As we have seen in the basics guide, Feathers works great in the browser and comes with client services that allow to easily connect to a Feathers server.

In this chapter we will create a real-time chat frontend with signup and login using modern plain JavaScript. As with the basics guide, it will only work in the latest versions of Chrome, Firefox, Safari and Edge since we won't be using a transpiler like Babel. The final version can be found here.

> **Note:** We will not be using a frontend framework so we can focus on what Feathers is all about. Feathers is framework agnostic and can be used with any frontend framework like React, VueJS or Angular. For more information see the frameworks section.

Setting up the index page

We are already serving a the static files in the `public` folder and have a placeholder page in `public/index.html`. Let's update it to the following:

```
<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>Vanilla JavaScript Feathers Chat</title>
    <link rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.2.0/public/base.css">
    <link rel="stylesheet" href="//cdn.rawgit.com/feathersjs/feathers-chat/v0.2.0/public/chat.css">
  </head>
  <body>
    <div id="app" class="flex flex-column"></div>
    <script src="//cdnjs.cloudflare.com/ajax/libs/moment.js/2.12.0/moment.js"></script>
    <script src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
    </script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

This will load our chat CSS style, add a container div `#app` and load several libraries:

- The browser version of Feathers (since we are not using a module loader like Webpack or Browserify)
- Socket.io provided by the chat API
- MomentJS to format dates
- An `app.js` for our code to live in

Let's create public/app.js where all the following code will live (with each code sample added to the end of that file).

Connecting to the API

We'll start with the most important thing first, the connection to our Feathers API. We already learned how Feathers can be used on the client in the basics guide. Here, we do pretty much the same thing: Establish a Socket connection and initialize a new Feathers application but we also set up the authentication client which we will use later:

```
// Establish a Socket.io connection
```

```
const socket = io();
// Initialize our Feathers client application through Socket.io
// with hooks and authentication.
const client = feathers();

client.configure(feathers.socketio(socket));
// Use localStorage to store our login token
client.configure(feathers.authentication({
  storage: window.localStorage
}));
```

This allows us to talk to the chat API through websockets, which means we will also get real-time updates.

Base and user/message list HTML

Next, we have to define some static and dynamic HTML that we can insert into the page when we want to show the login page (which also doubles as the signup page) and the actual chat interface:

```
// Login screen
const loginHTML = `<main class="login container">
  <div class="row">
    <div class="col-12 col-6-tablet push-3-tablet text-center heading">
      <h1 class="font-100">Log in or signup</h1>
    </div>
  </div>
  <div class="row">
    <div class="col-12 col-6-tablet push-3-tablet col-4-desktop push-4-desktop">
      <form class="form">
        <fieldset>
          <input class="block" type="email" name="email" placeholder="email">
        </fieldset>

        <fieldset>
          <input class="block" type="password" name="password" placeholder="password">
        </fieldset>

        <button type="button" id="login" class="button button-primary block signup">
          Log in
        </button>

        <button type="button" id="signup" class="button button-primary block signup">
          Sign up and log in
        </button>
      </form>
    </div>
  </div>
</main>`;

// Chat base HTML (without user list and messages)
const chatHTML = `<main 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="#" id="logout" class="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>
</main>`;

// Add a new user to the list
const addUser = user => {
  const userList = document.querySelector('.user-list');

  if(userList) {
    // Add the user to the list
    userList.insertAdjacentHTML('beforeend', `<li>
      <a class="block relative" href="#">
        <img src="${user.avatar}" alt="" class="avatar">
        <span class="absolute username">${user.email}</span>
      </a>
    </li>`);

    // Update the number of users
    const userCount = document.querySelectorAll('.user-list li').length;

    document.querySelector('.online-count').innerHTML = userCount;
  }
};

// Renders a new message and finds the user that belongs to the message
const addMessage = message => {
  // Find the user belonging to this message or use the anonymous user if not found
  const { user = {} } = message;
  const chat = document.querySelector('.chat');
  // Escape HTML
  const text = message.text
    .replace(/&/g, '&amp;')
    .replace(/</g, '&lt;').replace(/>/g, '&gt;');

  if(chat) {
    chat.insertAdjacentHTML( 'beforeend', `<div class="message flex flex-row">
      <img src="${user.avatar}" alt="${user.email}" class="avatar">
      <div class="message-wrapper">
        <p class="message-header">
          <span class="username font-600">${user.email}</span>
          <span class="sent-date font-300">${moment(message.createdAt).format('MMM Do, hh:mm:ss')}</span>
        </p>
        <p class="message-content font-300">${text}</p>
      </div>
    </div>`);

    chat.scrollTop = chat.scrollHeight - chat.clientHeight;
  }
};
```

This will add the following variables and functions:

- `loginHTML` contains some static HTML for the login/signup page
- `chatHTML` contains the main chat page content (once a user is logged in)
- `addUser(user)` is a function to add a new user to the user list on the left
- `addMessage(message)` is a function to add a new message to the list. It will also make sure that we always scroll to the bottom of the message list as messages get added

Displaying the login/signup or chat page

Next we will add two functions that show the login and chat page where we will also add a list of the 25 newest chat messages and the registered users.

```
// Show the login page
const showLogin = (error = {}) => {
  if(document.querySelectorAll('.login').length) {
    document.querySelector('.heading').insertAdjacentHTML('beforeend', `<p>There was an error: ${error.message}
</p>`);
  } else {
    document.getElementById('app').innerHTML = loginHTML;
  }
};

// Shows the chat page
const showChat = async () => {
  document.getElementById('app').innerHTML = chatHTML;

  // Find the latest 10 messages. They will come with the newest first
  // which is why we have to reverse before adding them
  const messages = await client.service('messages').find({
    query: {
      $sort: { createdAt: -1 },
      $limit: 25
    }
  });

  // We want to show the newest message last
  messages.data.reverse().forEach(addMessage);

  // Find all users
  const users = await client.service('users').find();

  users.data.forEach(addUser);
};
```

- `showLogin(error)` will either show the content of loginHTML or, if the login page is already showing, add an error message. This will happen when you try to log in with invalid credentials or sign up with a user that already exists.
- `showChat()` does several things. First, we add the static chatHTML to the page. Then we get the latest 25 messages from the messages Feathers service (this is the same as the `/messages` endpoint of our chat API) using the Feathers query syntax. Since the list will come back with the newest message first we need to reverse the data. Then we add each message by calling our `addMessage` function so that it looks like a chat app should—with messages getting older as you scroll up. After that we get a list of all registered users to show them in the sidebar by calling addUser.

Login and signup

Alright. Now we can show the login page (including an error message when something goes wrong) and if we are logged in call the `showChat` we defined above. We've built out the UI, now we have to add the functionality to actually allow people to sign up, log in and also log out.

```
// Retrieve email/password object from the login/signup page
const getCredentials = () => {
  const user = {
    email: document.querySelector('[name="email"]').value,
    password: document.querySelector('[name="password"]').value
  };

  return user;
};

// Log in either using the given email/password or the token from storage
const login = async credentials => {
  try {
    if(!credentials) {
      // Try to authenticate using the JWT from localStorage
      await client.authenticate();
    } else {
      // If we get login information, add the strategy we want to use for login
      const payload = Object.assign({ strategy: 'local' }, credentials);

      await client.authenticate(payload);
    }

    // If successful, show the chat page
    showChat();
  } catch(error) {
    // If we got an error, show the login page
    showLogin(error);
  }
};

document.addEventListener('click', async ev => {
  switch(ev.target.id) {
  case 'signup': {
    // For signup, create a new user and then log them in
    const credentials = getCredentials();

    // First create the user
    await client.service('users').create(credentials);
    // If successful log them in
    await login(credentials);

    break;
  }
  case 'login': {
    const user = getCredentials();

    await login(user);

    break;
  }
  case 'logout': {
    await client.logout();

    document.getElementById('app').innerHTML = loginHTML;

    break;
  }
  }
});
```

- `getCredentials()` gets us the values of the username (email) and password fields from the login/signup page to be used directly with Feathers authentication.
- `login(credentials)` will either authenticate the credentials returned by getCredentials against our Feathers API using the local authentication strategy (e.g. username and password) or, if no credentials are given, try and use the JWT stored in localStorage. This will try and get the JWT from localStorage first where it is put automatically

once you log in successfully so that we don't have to log in every time we visit the chat. Only if that doesn't work it will show the login page. Finally, if the login was successful it will show the chat page.

- We also added click event listeners for three buttons. `#login` will get the credentials and just log in with those. Clicking `#signup` will signup and log in at the same time. It will first create a new user on our API and then log in with that same user information. Finally, `#logout` will forget the JWT and then show the login page again.

Real-time and sending messages

In the last step we will add functionality to send new message and make the user and message list update in real-time.

```
document.addEventListener('submit', async ev => {
  if(ev.target.id === 'send-message') {
    // This is the message text input field
    const input = document.querySelector('[name="text"]');

    ev.preventDefault();

    // Create a new message and then clear the input field
    await client.service('messages').create({
      text: input.value
    });

    input.value = '';
  }
});

// Listen to created events and add the new message in real-time
client.service('messages').on('created', addMessage);

// We will also see when new users get created in real-time
client.service('users').on('created', addUser);

login();
```

- The `#submit` button event listener gets the message text from the input field, creates a new message on the messages service and then clears out the field.
- Next, we added two `created` event listeners. One for `messages` which calls the `addMessage` function to add the new message to the list and one for `users` which adds the user to the list via `addUser` . This is how Feathers does real-time and everything we need to do in order to get everything to update automatically.
- To kick our application off, we call `login()` which as mentioned above will either show the chat application right away (if we signed in before and the token isn't expired) or the login page.

What's next?

That's it. We now have a plain JavaScript real-time chat frontend with login and signup. This example demonstrates many of the core principles of how you interact with a Feathers API and concludes this chat guide. Follow up in the Feathers API documentation for all the details about using Feathers or start building your own first Feathers application.

Integrating with Frontend Frameworks

Feathers works the same on the server and on the client and is front-end framework agnostic! You can use it with Vue, React, React Native, Angular, or whatever other front-end tech stack you choose.

Client Side Docs

If you want to learn how to use Feathers as a client in Node.js, React Native, or in the browser with a module loader like Webpack refer to the client API docs.

Examples

The Feathers Chat application guide gives a basic intro to using the Feathers Client in a vanilla JavaScript environment. That's a good place to start so you can see how simple it is.

Beyond the basics, see this list of Feathers examples in awesome-feathersjs.

Framework Integrations

See this list of Feathers front-end framework integrations if you are looking for something that makes Feathers even easier to use with things like React, Vue or others.

Authentication Guides & Recipes

How JWT Works

Learn more about JWT and how it might differ from authentication methods you've used, previously. (This guide is a work in progress.)

Auth Recipe: Customize the JWT Payload

You can customize the JWT payload. Learn important security implications before you decide to do it.

Auth Recipe: Customize the Login Response

Learn how you can customize the response after a user has attempted to login.

Auth Recipe: Create Endpoints with Mixed Auth

Learn how to setup an endpoint so that it handles unauthenticated and authenticated users with different responses for each.

Auth Recipe: Basic OAuth

Learn how OAuth (Facebook, Google, GitHub) login works, and how you can use it in your application.

Auth Recipe: Custom Auth Strategy

Learn how to setup a completely custom passport based auth stratgies

How JSON Web Tokens Work

This guide is a work in progress. There's some useful information here while we make it more user friendly in the context of Feathers. In the meantime, here are a couple of resources on JWT to get more familiar with how it works, in general:

- The Auth0 JWT Documentation - If you want a good high-level overview.
- The IETF JWT Specification - If you want to get really technical.

Customizing the JWT Claims

`feathers-authentication@1.x` allows you to customize the data stored inside the JWT. We refer to the data in the JWT as the `payload`. There are a few reserved attributes, which in the Official JWT Spec are called `claims`. You can customize some of these claims in the JWT config options on the server. To get more familiar with the purpose of each `claim`, please refer to Section 4 of the Official JWT Specification.

FeathersJS Auth Recipe: Customizing the Login Response

The Auk release of FeathersJS includes a powerful new authentication suite built on top of PassportJS. The new plugins are very flexible, allowing you to customize nearly everything. This flexibility required making some changes. In this guide, we'll look at the changes to the login response and how you can customize it.

Changes to the Login Response

The previous version of `feathers-authentication` always returned the same response. It looked something like this:

```
{
  token: '<the jwt token>',
  user: {
    id: 1,
    email: 'my@email.com'
  }
}
```

The JWT also contained a payload which held an `id` property representing the user `id`. We found that this was too restrictive for some of our more technical apps. For instance, what if you wanted to authenticate a device instead of a user? Or what if you want to authenticate both a device **and** a user? The old plugin couldn't handle those situations. The new one does. To make it work, we started by simplifying the response. The default response now looks like this:

```
{
  accessToken: '<the jwt token>'
}
```

The JWT also contains a payload which has a `userId` property.

Based on the above, you can see that we still authenticate a `user` by default. In this case, the `user` is what we call the `entity`. It's the generic name of what is being authenticated. It's customizable, but that's not covered in this guide. Instead, let's focus on what it takes to add the user in the login response.

Customizing the Login Response

The `/authentication` endpoint is now a Feathers service. It uses the `create` method for login and the `remove` method for logout. Just like with all Feathers services, you can customize the response with the `hook` API. For what we want to do, the important part is the `context.result`, which becomes the response body. We can use an `after` hook to customize the `context.result` to return anything that we want:

```
app.service('/authentication').hooks({
  after: {
    create: [
      context => {
        context.result.foo = 'bar';
      }
    ]
  }
});
```

After a successful login, the `context.result` already contains the `accessToken`. The above example modified the response to look like this:

```
{
  accessToken: '<the jwt token>',
  foo: 'bar'
}
```

Accessing the User Entity

Let's see how to include the `user` in the response, as was done in previous versions. The `/authentication` service modifies the `context.params` object to contain the entity object (in this case, the `user`). With that information, you might have already figured out how to get the user into the response. It just has to be copied from `context.params.user` to the `context.result.user`:

```
app.service('/authentication').hooks({
  after: {
    create: [
      context => {
        context.result.user = context.params.user;

        // Don't expose sensitive information.
        delete context.result.user.password;
      }
    ]
  }
});
```

At this point, the response now includes the `accessToken` and the `user`. Now the client won't have to make an additional request for the `user` data. *As is shown in the above example, be sure to not expose any sensitive information.*

Wrapping Up

You've now learned some of the differences in the new `feathers-authentication` plugin. Instead of using two endpoints, it's using a single service. It also has a simplified response, compared to before. Now, you can customize the response to include whatever information you need.

FeathersJS Auth Recipe: Customizing the JWT Payload

The Auk release of FeathersJS includes a powerful new authentication suite built on top of PassportJS. The new plugins are very flexible, allowing you to customize nearly everything. One feature added in the latest release is the ability to customize the JWT payload using hooks. Let's take a look at what this means, how to make it work, and learn about the potential pitfalls you may encounter by using it.

The JWT Payload

If you read the resources on how JWT works, you'll know that a JWT is an encoded string that can contain a payload. For a quick example, check out the Debugger on jwt.io. The purple section on jwt.io is the payload. You'll also notice that you can put arbitrary data in the payload. The payload data gets encoded as the section section of the JWT string.

The default JWT payload contains the following claims:

```
const decode = require('jwt-decode')
// Retrieve the token from wherever you've stored it.
const jwt = window.localStorage.getItem('feathers-jwt')
const payload = decode(jwt)

payload === {
  aud: 'https://yourdomain.com', // audience
  exp: 23852348347, // expires at time
  iat: 23852132232, // issued at time
  iss: 'feathers', // issuer
  sub: 'anonymous', // subject
  userId: 1 // the user's id
}
```

Notice that the payload ***is encoded*** and ***IS NOT ENCRYPTED***. It's an important difference. It means that you want to be careful what you store in the JWT payload.

Customizing the Payload with Hooks

The authentication services uses the `params.payload` object in the hook context for the JWT payload. This means you can customize the JWT by adding a before hook after the `authenticate` hook.

```
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(config.strategies),

      // This hook adds the `test` attribute to the JWT payload by
      // modifying params.payload.
      context => {
        // make sure params.payload exists
        context.params.payload = context.params.payload || {}
        // merge in a `test` property
        Object.assign(context.params.payload, {test: 'test'})
      }
    ],
    remove: [
      authentication.hooks.authenticate('jwt')
```

```
    ]
  }
})
```

Now the payload will contain the `test` attribute:

```
const decode = require('jwt-decode')
// Retrieve the token from wherever you've stored it.
const jwt = window.localStorage.getItem('feathers-jwt')
const payload = decode(jwt)

payload === {
  aud: 'https://yourdomain.com',
  exp: 23852348347,
  iat: 23852132232,
  iss: 'feathers',
  sub: 'anonymous',
  userId: 1
  test: 'test' // Here's the new claim we just added
}
```

> Note: The payload is not automatically decoded and made available in the hooks, thus, requiring you to implement this functionality in your app. Using `jwt-decode` is a simple solution that could be dropped in a hook as needed.

Important Security Information

As you add data to the JWT payload the token size gets larger. Try it out on jwt.io to see for yourself. There is an important security issue to keep in mind when customizing the payload. This issue involves the default `HS256` algorithm used to sign the token.

With `HS256`, there is a relationship between the length of the secret (which must be a minimum of 256-bits) and the length of the encoded token (which varies with the payload). A larger secret-to-payload ratio (so the secret is larger than the JWT) will result in a more secure JWT. This also means that keeping the secret size the same and increasing the payload size will actually make your JWT comparatively less secure.

The Feathers generator creates a 2048-bit secret, by default, so there is a small amount of allowable space for putting additional attributes in the JWT payload. It's very important to keep the secret-to-payload length ratio as high as possible to avoid brute force attacks. In a brute force attack, the attacker attempts to retrieve the secret by guessing the secret over and over until getting it right. If your secret is compromised, they will be able to create signed JWT with whatever payload they wish. In short, be cautious about what you put in your JWT payload.

Finally, remember that the secret created by the generator is meant for development purposes, only. You never want to check your **production** secret into your version control system (Git, etc.). It is best to put your production secret in an environment variable and reference it in the app configuration.

FeathersJS Auth Recipe: Create Endpoints with Mixed Auth

The Auk release of FeathersJS includes a powerful new authentication suite built on top of PassportJS. It can be customized to handle almost any app's authentication requirements. In this guide, we'll look at how to handle a fairly common auth scenario: Sometimes an endpoint needs to serve different information depending on whether the user is authenticated. An unauthenticated user might only see public records. An authenticated user might be able to see additional records.

Setup the Authentication Endpoint

To get started, we need a working authentication setup. Below is a default configuration and `authentication.js`. They contain the same code generated by the feathers-cli. You can create the below setup using the following terminal commands:

1. `npm install -g feathers-cli@latest`
   or
   `yarn global feathers-cli@latest`
2. `mkdir feathers-demo-local; cd feathers-demo-local`
   or a folder name you prefer.
3. `feathers generate app`
   use the default prompts.
4. `feathers generate authentication`
   - Select `Username + Password (Local)` when prompted for a provider.
   - Select the defaults for the remaining prompts.

config/default.json:

```
{
  "host": "localhost",
  "port": 3030,
  "public": "../public/",
  "paginate": {
    "default": 10,
    "max": 50
  },
  "authentication": {
    "secret": "99294186737032fedad37dc2e847912e1b9393f44a28101c986f6ba8b8bc0eaab48b5b4c5178f55164973c76f8f98f25
23b860674f01c16a23239a2e7d7790ae9fa00b6de5cc0565e335c6f05f2e17fbee2e8ea0e82402959f1d58b2b2dc5272d09e0c1edf1d364
e9911ecad8172bdc2d41381c9ab319de4979c243925c49165a9914471be0aa647896e981da5aec6801a6dccd1511da11b696d4f6cce3a45
34dab9368661458a466661b1e12170ad21a4045ce1358138caf099fbc19e05532336b5626aa376bc158cf84c6a7e0e3dbbb3af666267c08
de12217c9b55aea501e5c36011779ee9dd2e061d0523ddf71cb1d68f83ea5bb1299ca06003b77f0fc69",
    "strategies": [
      "jwt",
      "local"
    ],
    "path": "/authentication",
    "service": "users",
    "jwt": {
      "header": {
        "typ": "access"
      },
      "audience": "https://yourdomain.com",
      "subject": "anonymous",
      "issuer": "feathers",
      "algorithm": "HS256",
```

```
      "expiresIn": "1d"
    },
    "local": {
      "entity": "user",
      "service": "users",
      "usernameField": "email",
      "passwordField": "password"
    }
  },
  "nedb": "../data"
}
```

src/authentication.js:

```
'use strict';

const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const local = require('feathers-authentication-local');

module.exports = function () {
  const app = this;
  const config = app.get('authentication');

  app.configure(authentication(config));
  app.configure(jwt());
  app.configure(local(config.local));

  app.service('authentication').hooks({
    before: {
      create: [
        authentication.hooks.authenticate(config.strategies)
      ],
      remove: [
        authentication.hooks.authenticate('jwt')
      ]
    }
  });
};
```

Set up a “Mixed Auth” Endpoint

Now we need to setup an endpoint to handle both unauthenticated and authenticated users. For this example, we’ll use the `/users` service that was already created by the authentication generator. Let’s suppose that our application requires that each `user` record will contain a `public` boolean property. Each record will look something like this:

```
{
  id: 1,
  email: 'my@email.com'
  password: "password",
  public: true
}
```

If a `user` record contains `public: true`, then **unauthenticated** users should be able to access it. Let’s see how to use the `iff` and `else` conditional hooks from `feathers-hooks-common` to make this happen. Be sure to read the `iff hook API docs` and `else hook API docs` if you haven’t, yet.

We’re going to use the `iff` hook to authenticate users only if a token is in the request. The `feathers-authentication-jwt` plugin, which we used in `src/authentication.js`, includes a token extractor. If a request includes a token, it will automatically be available inside the `context` object at `context.params.token`.

src/services/users/users.hooks.js
(This example only shows the `find` method's `before` hooks.)

```
'use strict';

const { authenticate } = require('feathers-authentication').hooks;
const commonHooks = require('feathers-hooks-common');

module.exports = {
  before: {
    find: [
      // If a token was included, authenticate it with the `jwt` strategy.
      commonHooks.iff(
        context => context.params.token,
        authenticate('jwt')
      // No token was found, so limit the query to include `public: true`
      ).else( context => Object.assign(context.params.query, { public: true }) )
    ]
  }
};
```

Let's break down the above example. We setup the `find` method of the `/users` service with an `iff` conditional before hook:

```
iff(
  context => context.params.token,
  authenticate('jwt')
)
```

For this application, the above snippet is equivalent to the snippet, below.

```
context => {
  if (context.params.token) {
    return authenticate('jwt')
  } else {
    return Promise.resolve(context)
  }
}
```

The `iff` hook is actually more capable than the simple demonstration, above. It can handle an async predicate expression. This would be equivalent to being able to pass a `promise` inside the `if` statement's parentheses. It also allows us to chain an `.else()` statement, which will run if the predicate evaluates to false.

```
.else( context => Object.assign(context.params.query, { public: true }) )
```

The above statement simply adds `public: true` to the query parameters. This limits the query to only find `user` records that have the `public` property set to `true`.

Wrapping Up

With the above code, we've successfully setup a `/users` service that responds differently to unauthenticated and authenticated users. We used the `context.params.token` attribute to either authenticate a user or to limit the search query to only public users. If you become familiar with the Common Hooks API, you'll be able to solve almost any authentication puzzle.

FeathersJS Auth Recipe: Set up Basic OAuth Login

The Auk release of FeathersJS includes a powerful new authentication suite built on top of PassportJS. This now gives the Feathers community access to hundreds of authentication strategies from the Passport community. Since many of the Passport strategies are for OAuth, we've created two auth plugins, `feathers-authentication-oauth1` and `feathers-authentication-oauth2`. These new plugins use a Passport strategy to allow OAuth logins into your app.

Adding OAuth authentication to your app is a great way to quickly allow users to login. It allows the user to use an existing Internet account with another service to login to your app. Among lots of good reasons, it often eliminates the need for the email address verification dance. This is even more likely for very common OAuth providers, like GitHub, Google, and Facebook.

Simplified login is almost always a good idea, but for many developers implementing OAuth can be difficult. Let's take a look at how it works, in general. After that, we'll see how the new `feathers-authentication` server plugin makes it easy to get up and running.

How OAuth Works

There are a couple of different methods you can use to implement OAuth. Here are the basic steps of the flow that the `feathers-authentication-oauth1` and `feathers-authentication-oauth2` plugins use.

1. You register your application with the OAuth Provider. This includes giving the provider a callback URL (more on this later). The provider will give you an app identifier and an app secret. The secret is basically a special password for your app.
2. You direct the user's browser to the OAuth provider's site, providing the app identifier in the query string.
3. The content provider uses the app identifier to retrieve information about your app. That information is then presented to the user with a login form. The user can grant or deny access to your application.
4. Upon making a decision, the provider redirects the user's browser to the callback URL you setup in the first step. It includes a short-lived authorization code in the querystring.
5. Your server sends a request to the OAuth provider's server. It includes the authorization code and the secret. If the authorization code and secret are valid, the provider returns an OAuth access token to your server. Some user data can also be sent.
6. Your server can save the user information into the `/users` table. It can also use this access token to make requests to the provider's API. This same information can also be sent to the browser for use.
7. With Feathers, there is an additional step. After logging in, a JWT access token is stored in a cookie and sent to the browser. The client uses the JWT to authenticate with the server on subsequent requests.

Implementing OAuth with Feathers

The `Feathers-cli` allows you to easily setup a new application with OAuth. Here are the steps to generate an application:

1. `npm install -g feathers-cli`
   or
   `yarn global feathers-cli`
2. `mkdir feathers-demo-oauth; cd feathers-demo-oauth`
   or a folder name you prefer.
3. `feathers generate app`
   use the default prompts.

4. `feathers generate authentication`
   - Select `Facebook`, `GitHub`, or `Google` when prompted for a provider.
     This guide will show how to use GitHub.
   - Select the defaults for the remaining prompts.

Setting up the OAuth Provider

To setup the provider, you use each provider's website. Here are links to common providers:

- Facebook
- GitHub
- Google

Once your app is setup, the OAuth provider will give you a `Client ID` and `Client Secret`.

Configuring Your Application

Once you have your app's `Client ID` and `Client Secret`, it's time to setup the app to communicate with the provider. Open the `default.json` configuration file. The generator added a key to the config for the provider you selected. The below configuration example has a `github` config. Copy over and replace the placeholders with the `clientID` and `clientSecret`.

config/default.json

```
{
  "host": "localhost",
  "port": 3030,
  "public": "../public/",
  "paginate": {
    "default": 10,
    "max": 50
  },
  "authentication": {
    "secret": "cc71e4f97a80c878491197399aabf74e9c0b115c9f8071e75b306c99c891a54b7171852f8c5508e1fe4dcfaedbb60317
8b0935261928592e487e628f2f669f3a752f2beb3661b29d521b36c8a39e1be6823c0362df5ef1e212d7f2daae789df1065293b98ec9b43
309ffe24dba3a2ec2362c5ce5c9155c6438ec380bc7c56d6a169988c0f6754077c5129e8a0ee5fd85b2182d87f84312387e1bbefebe49ad
1bf2dcf783e7d8cbee40272b141358b8e23150eee5ea8fc04b2a0f3d824e7fa9d46c025c619c3281af91b7a19fd760bccedae379b735c85
024b25a9c91749935b2f29d5b69b2c1ff29368b4aa9cf426d9960302e5e7b903d53e18ccbe2325cf3b6",
    "strategies": [
      "jwt"
    ],
    "path": "/authentication",
    "service": "users",
    "jwt": {
      "header": {
        "typ": "access"
      },
      "audience": "https://yourdomain.com",
      "subject": "anonymous",
      "issuer": "feathers",
      "algorithm": "HS256",
      "expiresIn": "1d"
    },
    "github": {
      "clientID": "your github client id", // Replace this with your app's Client ID
      "clientSecret": "your github client secret", // Replace this with your app's Client Secret
      "successRedirect": "/"
    },
    "cookie": {
      "enabled": true,
```

```
      "name": "feathers-jwt",
      "httpOnly": false,
      "secure": false
    }
  },
  "nedb": "../data"
}
```

Test Login with OAuth

Your app is ready for OAuth logins. We've made it that simple! Let's try it out. Open the file `public/index.html` and scroll to the bottom. Add the following code just under the `h2`:

```
<p class="center-text"><br/>
  <a href="/auth/github">Login With GitHub</a>
</p>
```

Now add the following code to the same page. The first script tag loads Feathers Client from a CDN. The second script loads Socket.io. The third script creates a Feathers Client instance and attempts to authenticate with the JWT strategy upon page load. The authentication client plugin has been configured with a `cookie` value of `feathers-jwt`.

> Note: This code loads the `feathers-client` package from a CDN. This is **not** the recommended usage for most apps, but is good for demonstration purposes. We recommend using a bundler as described in the Feathers Client API docs.

```
<script src="//unpkg.com/feathers-client@2.0.0/dist/feathers.js"></script>
<script src="//unpkg.com/socket.io-client@1.7.3/dist/socket.io.js"></script>
<script>
  // Socket.io is exposed as the `io` global.
  var socket = io('http://localhost:3030', {transports: ['websocket']});
  // feathers-client is exposed as the `feathers` global.
  var feathersClient = feathers()
    .configure(feathers.hooks())
    .configure(feathers.socketio(socket))
    .configure(feathers.authentication({
      cookie: 'feathers-jwt'
    }));

  feathersClient.authenticate()
    .then(response => {
      console.info('Feathers Client has Authenticated with the JWT access token!');
      console.log(response);
    })
    .catch(error => {
      console.info('We have not logged in with OAuth, yet.  This means there\'s no cookie storing the accessTok
en.  As a result, feathersClient.authenticate() failed.');
      console.log(error);
    });
</script>
```

Now, run the server, open `http://localhost:3030`. Before you click the "Login with GitHub" link, open the console. If you refresh you'll see the message in the catch block. Since we haven't logged in, yet, we don't have a stored JWT access token. Now, click the `Login with GitHub` button. Assuming you haven't logged in to Github with this application, before, you'll see a GitHub login page. Once you login to GitHub, you'll be redirected back to `http://localhost:3030`. Now, if you look at your console, you should see a success message.

What just happened? When you clicked on that link, it opened the `/auth/github` link, which is just a shortcut for redirecting to GitHub with your `Client ID`. The entire OAuth process that we described earlier took place. The browser received a `feathers-jwt` cookie from the server. Finally the script that we added in the last step used the

`feathers-authentication-client` to authenticate using the JWT returned from the server. There were a lot of steps that happened in a very short time. The best news is that you're authenticated with OAuth.

Wrapping Up

You've now seen how OAuth login is greatly simplified with the new Feathers Authentication plugins. Having plugins built on top of PassportJS allows for a lot of flexibility. You can now build nearly any authentication experience imaginable. In the final part of this guide, you were able to authenticate the Feathers client. Hopefully this will get you started integrating OAuth into your application.

FeathersJS Auth Recipe: Custom Auth Strategy

The Auk release of FeathersJS includes a powerful new authentication suite built on top of PassportJS. The new plugins are very flexible, allowing you to customize nearly everything. We can leverage this to create completely custom authentication strategies using Passport Custom. Let's take a look at two such examples in this guide.

Setting up the basic app

Let's first start by creating a basic server setup.

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const auth = require('@feathersjs/authentication');
const jwt = require('@featehrsjs/authentication-jwt');
const memory = require('feathers-memory');

const app = express(feathers());

app.configure(express.rest());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.configure(auth({ secret: 'secret' }));
app.configure(jwt());
app.use('/users', memory());

app.hooks({
  before: {
    all: [auth.hooks.authenticate('jwt')]
  }
});

app.listen(8080);
```

Creating a Custom API Key Auth Strategy

The first custom strategy example we can look at is an API Key Strategy. Within it, we'll check if there is a specific header in the request containing a specific API key. If true, we'll successfully authorize the request.

First let's make the strategy using `passport-custom` npm package.

```
const Strategy = require('passport-custom');

module.exports = opts => {
  return function() {
    const verifier = (req, done) => {

      // get the key from the request header supplied in opts
      const key = req.params.headers[opts.header];

      // check if the key is in the allowed keys supplied in opts
      const match = opts.allowedKeys.includes(key);

      // user will default to false if no key is present
      // and the authorization will fail
      const user = match ? 'api' : match;
      return done(null, user);
    };
```

```
    // register the strategy in the app.passport instance
    this.passport.use('apiKey', new Strategy(verifier));
  };
};
```

Next let's add this to our server setup

```
const apiKey = require('./apiKey');

app.configure(
  apiKey({
    // which header to look at
    header: 'x-api-key',
    // which keys are allowed
    allowedKeys: ['opensesame']
  })
);
```

Next let's create a custom authentication hook that conditionally applies auth for all external requests.

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

const authenticate = () =>
  commonHooks.iff(
    // if and only if the request is external
    commonHooks.every(commonHooks.isProvider('external')),
    commonHooks.iffElse(
      // if the specific header is included
      ctx => ctx.params.headers['x-api-key'],
      // authentication with this strategy
      auth.hooks.authenticate('apiKey'),
      // else fallback on the jwt strategy
      auth.hooks.authenticate(['jwt'])
    )
  );

app.hooks({
  before: {
    all: [authenticate()]
  }
});
```

Finally our `server.js` looks like this:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const auth = require('@feathersjs/authentication');
const jwt = require('@featehrsjs/authentication-jwt');
const memory = require('feathers-memory');
const commonHooks = require('feathers-hooks-common');

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

const app = express(feathers());

app.configure(express.rest());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.configure(auth({ secret: 'secret' }));
app.configure(jwt());
app.configure(
  apiKey({
```

```
    header: 'x-api-key',
    allowedKeys: ['opensesame']
  })
);

app.use('/users', memory());

const authenticate = () =>
  commonHooks.iff(
    commonHooks.every(commonHooks.isProvider('external')),
    commonHooks.iffElse(
      ctx => ctx.params.headers['x-api-key'],
      auth.hooks.authenticate('apiKey'),
      auth.hooks.authenticate(['jwt'])
    )
  );

app.hooks({
  before: {
    all: [authenticate()]
  }
});

app.listen(8080);
```

Now any request with a header `x-api-key` and the value `opensesame` will be authenticated by the server.

Creating an Anonymous User Strategy

The second strategy we'll look at is for an anonymous user. For this specific flow we'll expect the client to call the `/authentication` endpoint letting us know that it wants to authenticate anonymously. The server will then create a new user and return a new JWT token that the client will have to use from that point onwards.

First let's create the strategy using `passport-custom`

```
const Strategy = require('passport-custom');

module.exports = opts => {
  return function() {
    const verifier = async (req, done) => {
      // create a new user in the user service
      // mark this user with a specific anonymous=true attribute
      const user = await this.service(opts.userService).create({
        anonymous: true
      });

      // authenticate the request with this user
      return done(null, user, {
        userId: user.id
      });
    };

    // register the strategy in the app.passport instance
    this.passport.use('anonymous', new Strategy(verifier));
  };
};
```

Next let's update our `server.js` to use this strategy.

```
const anonymous = require('./anonymous');

app.configure(
  anonymous({
```

```
    // the user service
    userService: 'users'
  })
);

const authenticate = () =>
  commonHooks.iff(
    commonHooks.every(commonHooks.isProvider('external')),
    commonHooks.iffElse(
      ctx => ctx.params.headers['x-api-key'],
      auth.hooks.authenticate('apiKey'),
      // add the additional anonymous strategy
      auth.hooks.authenticate(['jwt', 'anonymous'])
    )
  );
```

Finally our `server.js` looks like this:

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const auth = require('@feathersjs/authentication');
const jwt = require('@featehrsjs/authentication-jwt');
const memory = require('feathers-memory');
const commonHooks = require('feathers-hooks-common');

const apiKey = require('./apiKey');
const anonymous = require('./anonymous');

const app = express(feathers());

app.configure(express.rest());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.configure(auth({ secret: 'secret' }));
app.configure(jwt());
app.configure(
  apiKey({
    header: 'x-api-key',
    allowedKeys: ['opensesame']
  })
);
app.configure(
  anonymous({
    userService: 'users'
  })
);

app.use('/users', memory());

const authenticate = () =>
  commonHooks.iff(
    commonHooks.every(commonHooks.isProvider('external')),
    commonHooks.iffElse(
      ctx => ctx.params.headers['x-api-key'],
      auth.hooks.authenticate('apiKey'),
      auth.hooks.authenticate(['jwt', 'anonymous'])
    )
  );

app.hooks({
  before: {
    all: [authenticate()]
  }
});

app.listen(8080);
```

Now any such request will return a valid JWT token:

```
POST /authentication

{
  strategy: 'anonymous'
}
```

Note that this looks very similar to a request body for `local` strategy:

```
POST /authentication

{
  strategy: 'local',
  username: 'admin',
  password: 'password'
}
```

So for any new strategy we register, we can call the `/authentication` endpoint with a specific body and expect a valid JWT in return, which we can use from thereon.

As we can see it's very easy to create a completely custom auth strategy in a standard passport way using `passport-custom`.

Happy Hacking!!

Advanced guides

In this section you can find some guides for advanced topics once you have learned the basics and created your first app.

- Debugging
- File uploads
- Creating a Feathers plugin
- Using a view engine
- Scaling
- Local Authentication Management
- Offline first

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 were struggling to find a way to upload files without 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 to accomplish a few important things:

1. It has to handle large files (+10MB).
2. It needs to work with the app's authentication and authorization.
3. The files need 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. It has to show the upload progress.

The plan is to upload the files to a feathers service so we can take advantage of hooks for authentication, authorization and validation, and for service events.

Fortunately, there exists a file storage service: feathers-blob. With it we can meet our goals, but (spoiler alert) it isn't an ideal solution. We discuss some of its problems 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('@feathersjs/feathers');
const express = require('@feathersjs/express');
const socketio = require('feathers-socketio');

// 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 = express(feathers());

// Parse HTTP JSON bodies
app.use(express.json());
// Parse URL-encoded params
app.use(express.urlencoded({ extended: true }));
// Add REST API support
app.configure(express.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(express.errorHandler());

// Start the server
app.listen(3030, function(){
    console.log('Feathers app started at localhost:3030')
});
```

Let's look at this implmented in the `feathers-cli` generated server code:

```
/* --- /src/services/uploads/uploads.service.js --- */

// Initializes the `uploads` service on path `/uploads'


// Here we used the nedb database, but you can
// use any other ORM database.
const createService = require('feathers-nedb');

const createModel = require('../../models/uploads.model');
const hooks = require('./uploads.hooks');
const filters = require('./uploads.filters');


// 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');


// File storage location. Folder must be created before upload.
// Example: './uploads' will be located under feathers app top level.
const blobStorage = fs('./uploads');

module.exports = function() {
  const app = this;
  const Model = createModel(app);
  const paginate = app.get('paginate');

  // Initialize our service with any options it requires
  app.use('/uploads', blobService({ Model: blobStorage}));

  // Get our initialized service so that we can register hooks and filters
  const service = app.service('uploads');

  service.hooks(hooks);

  if (service.filter) {
    service.filter(filters);
  }
};
```

`feathers-blob` works over abstract-blob-store, which 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': '
vrvYvntYvnvY/ruZPrwZPfsZPjsZfjtZvfsZvHmY/zxavftaPrvavjuafzxbfnua/jta/ftbP3yb/zzcPvwb/zzcfvxcfzxc/3zdf3zdv70efvw
d/rwd/vwefftd/3yfPvxfP70f/zzfvnwffvzf/rxf/rxgPjvgPjvgfnwhPvzhvjvhv71jfz0kPrykvz0mv72nvblTPnnUPjoUPrpUvnnUfnpUvX
lUfnpU/npVPnqVPfnU/3uVvvsWPfpVvnqWfrrXPLiW/nrX/vtYv7xavrta/Hlcvnuf/Pphvbsif3zk/zzlPzylfjuk/z0o/LqnvbhSPbhSfjiS/
jlS/jjTPfhTfjlTubUU+/iiPPokfrvl/Dll/ftovLWPfHXPvHZP/PbQ/bcRuDJP/PaRvjgSffdSe3ddu7fge7fi+zkuO7NMvPTOt2/Nu7SO+3OO
/PWQdnGbOneqeneqvDqyu3JMuvJMu7KNfHNON7GZdnEbejanObXnOW8JOa9KOvCLOnBK9+4Ku3FL9ayKuzEMcenK9e+XODOiePSkODOkOW3Itis
I9yxL+a9NtGiHr+VH5h5JsSfNM2bGN6rMJt4JMOYL5h4JZl5Jph3Jpl4J5h5J5h3KJl4KZp5Ks+sUN7Gi96lLL+PKMmbMZt2Jpp3Jpt3KZl4K7q
FFdyiKdufKsedRdm7feOpQN2QKMKENrpvJbFfIrNjJL1mLMBpLr9oLrFhK69bJFkpE1kpFYNeTqFEIlsoFbmlnlsmFFwpGFkoF/////7+/v///w
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACH5BAEAANAALAAAAAATABMAAAj/AKEJHCgo
kKJKlhThGciQYSIva7r8SHPFzqGGAwPd4bKlh5YsPKy0qFLnT0NAaHTcsIHDho0aKkaAwGCGEkM1NmSkIjWLBosVJT6cOjUrzsBKPl54KmYsACo
TMmk1WwaA1CRoeM7siJEqmTIAsjp40ICK2bEApfZcsoQlxwxRzgI8W8XhgoVYA+Kq6sMK0QEYKVCUkoVqQwQJFTwFEAAAFZ9PlFy4OEEiRIYJD5
5EodDA1ClTbPp0okRFxBQDBRgskAKhiRMlc+Sw4SNpFCIoBBwkUMBkCBIiY8qAgcPG0KBHrBTFQbCEV5EjQYQACfNFjp5CgxpxagVtUhIjwzaJY
SHzhQ4cP3ryQHLEqJbASnu+6EIW6o2b2X0ISXK0CFSugazs0YYmwQhziyuE2PLLIv3h0hArkRhiCCzAENOLL7tgAoqDGLXSSSaPMLIIJpmAUst/
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='sha256-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='//unpkg.com/feathers-client@^2.0.0/dist/feathers.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(percentComplete * 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 something 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 process, 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 reassembling 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 prefer.

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(context) {
            if (!context.data.uri && context.params.file){
                const file = context.params.file;
                const uri = dauria.getBase64DataURI(file.buffer, file.mimetype);
                context.data = {uri: uri};
            }
        }
    ]
});
```

Et voilà!. Now we have a FeathersJS 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, which may not be necessary 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, that's 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='//unpkg.com/feathers-client@^2.0.0/dist/feathers.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!

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.

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 `context.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('@feathersjs/client');
const socketio = require('@feathersjs/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.

Scaling

Depending on your requirements, your feathers application may need to provide high availability. Feathers is designed to scale.

The types of transports used in a feathers application will impact the scaling configuration. For example, a feathers app that uses the `feathers-rest` adapter exclusively will require less scaling configuration because HTTP is a stateless protocol. If using websockets (a stateful protocol) through the `feathers-socketio` or `feathers-primus` adapters, configuration may be more complex to ensure websockets work properly.

Horizontal Scaling

Scaling horizontally refers to either:

- setting up a cluster, or
- adding more machines to support your application

To achieve high availability, varying combinations of both strategies may be used.

Cluster configuration

Cluster support is built into core NodeJS. Since NodeJS is single threaded, clustering allows you to easily distribute application requests among multiple child processes (and multiple threads). Clustering is a good choice when running feathers in a multi-core environment.

Below is an example of adding clustering to feathers with the `feathers-socketio` provider. By default, websocket connections begin via a handshake of multiple HTTP requests and are upgraded to the websocket protocol. However, when clustering is enabled, the same worker will not process all HTTP requests for a handshake, leading to HTTP 400 errors. To ensure a successful handshake, force a single worker to process the handshake by disabling the http transport and exclusively using the `websocket` transport.

There are notable side effects to be aware of when disabling the HTTP transport for websockets. While all modern browsers support websocket connections, there is no websocket support for IE <=9 and Android Browser <=4.3. If you must support these browsers, use alternative scaling strategies.

```
import cluster from 'cluster';
import feathers from '@feathersjs/feathers';
import socketio from '@feathersjs/socketio';

const CLUSTER_COUNT = 4;

if (cluster.isMaster) {
  for (let i = 0; i < CLUSTER_COUNT; i++) {
    cluster.fork();
  }
} else {
  const app = feathers();
  // ensure the same worker handles websocket connections
  app.configure(socketio({
    transports: ['websocket']
  }));
  app.listen(4000);
}
```

In your feathers client code, limit the socket.io-client to the `websocket` transport and disable `upgrade`.

```
import feathers from '@feathersjs/client';
import io from 'socket.io-client';
import socketio from 'feathers-socketio/client';

const app = feathers()
  .configure(socketio(
    io('http://api.feathersjs.com', {
      transports: ['websocket'],
      upgrade: false
    })
  ));
```

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 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.

API

This section describes all the APIs of Feathers and its individual modules.

- **Core:** Feathers core functionality
  - Application - The main Feathers application API
  - Services - Service objects and their methods and Feathers specific functionality
  - Hooks - Pluggable middleware for service methods
  - Events - Events sent by Feathers service methods
  - Channels - Decide what events to send to connected real-time clients
  - Errors - A collection of error classes used throughout Feathers
  - Configuration - A node-config wrapper to initialize configuraiton of a server side application.
- **Transports:** Expose a Feathers application as an API server
  - Express - Feathers Express framework bindings, REST API provider and error middleware.
  - Socket.io - The Socket.io real-time transport provider
  - Primus - The Primus real-time transport provider
- **Client:** More details on how to use Feathers on the client
  - Usage - Feathers client usage in Node, React Native and the browser (also with Webpack and Browserify)
  - REST - Feathers client and direct REST API server usage
  - Socket.io - Feathers client and direct Socket.io API server usage
  - Primus - Feathers client and direct Primus API server usage
- **Authentication:** Feathers authentication mechanism
  - Server - The main authentication server configuration
  - Client - A client for a Feathers authentication server
  - Local - Local email/password authentication
  - JWT - JWT authentication
  - OAuth1 - Obtain a JWT through oAuth1
  - OAuth2 - Obtain a JWT through oAuth2
- **Database:** Feathers common database adapter API and querying mechanism
  - Adapters - A list of supported database adapters
  - Common API - Database adapter common initialization and configuration API
  - Querying - The common querying mechanism

Application


```
$ npm install @feathersjs/feathers --save
```

The core `@feathersjs/feathers` module provides the ability to initialize new Feathers application instances. It works in Node, React Native and the browser (see the client chapter for more information). Each instance allows for registration and retrieval of services, hooks, plugin configuration, and getting and setting configuration options. An initialized Feathers application is referred to as the **app object**.

```
const feathers = require('@feathersjs/feathers');

const app = feathers();
```

.use(path, service)

`app.use(path, service) -> app` allows registering a service object on a given `path`.

```
// Add a service.
app.use('/messages', {
  get(id) {
    return Promise.resolve({
      id,
      text: `This is the ${id} message!`
    });
  }
});
```

.service(path)

`app.service(path) -> service` 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 `app.service(path)` will provide the same methods and functionality as your original service object but also functionality added by Feathers and its plugins like service events and additional methods. `path` can be the service name with or without leading and trailing slashes.

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

messageService.get('test').then(message => console.log(message));

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

const todoService = app.service('my/todos');
// todoService is an event emitter
todoService.on('created', todo =>
  console.log('Created todo', todo)
);
```

.mixins

`app.mixins` contains a list of service mixins. A mixin is a callback (`(service, path) => {}`) that gets run for every service that is being registered. Adding your own mixins allows to add functionality to every registered service.

```
const feathers = require('@feathersjs/feathers');
const app = feathers();

// Mixins have to be added before registering any services
app.mixins.push((service, path) => {
  service.sayHello = function() {
    return `Hello from service at '${path}'`;
  }
});

app.use('/todos', {
  get(id) {
    return Promise.resolve({ id });
  }
});

app.service('todos').sayHello();
// -> Hello from service at 'todos'
```

.hooks(hooks)

`app.hooks(hooks) -> app` allows registration of application-level hooks. For more information see the application hooks section in the hooks chapter.

.publish([event,] publisher)

`app.publish([event, ] publisher) -> app` registers a global event publisher. For more information see the channels publishing chapter.

.configure(callback)

`app.configure(callback) -> app` runs a `callback` function that gets passed the application object. It is used to initialize plugins or services.

```
function setupService(app) {
  app.use('/todos', todoService);
}

app.configure(setupService);
```

.listen(port)

`app.listen([port]) -> HTTPServer` starts the application on the given port. It will set up all configured transports (if any) and then run `app.setup(server)` (see below) with the server object and then return the server object.

`listen` will only be available if a server side transport (REST, Socket.io or Primus) has been configured.

.setup([server])

`app.setup([server]) -> app` 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 it needs to be called explicitly.

.set(name, value)

`app.set(name, value) -> app` assigns setting `name` to `value`.

.get(name)

`app.get(name) -> value` retrieves the setting `name`. For more information on server side Express settings see the Express documentation.

```
app.set('port', 3030);

app.listen(app.get('port'));
```

.on(eventname, listener)

Provided by the core NodeJS EventEmitter .on. Registers a `listener` method (`function(data) {}`) for the given `eventname`.

```
app.on('login', user => console.log('Logged in', user));
```

.emit(eventname, data)

Provided by the core NodeJS EventEmitter .emit. Emits the event `eventname` to all event listeners.

```
app.emit('myevent', {
  message: 'Something happened'
});

app.on('myevent', data => console.log('myevent happened', data));
```

.removeListener(eventname, [listener])

Provided by the core NodeJS EventEmitter .removeListener. Removes all or the given listener for `eventname`.

Services

Services are the heart of every Feathers application and JavaScript objects (or instances of ES6 classes) that implements certain methods. Feathers itself will also add some additional methods and functionality to its services.

Service methods

Service methods are pre-defined CRUD methods that your service object can implement (or that has already been implemented by one of the database adapters). Below is a complete example of the Feathers *service interface* as a normal JavaScript object returning eihter a Promise or using async/await:

Promise
async/await

```
const myService = {
  find(params) {
    return Promise.resolve([]);
  },
  get(id, params) {},
  create(data, params) {},
  update(id, data, params) {},
  patch(id, data, params) {},
  remove(id, params) {},
  setup(app, path) {}
}

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

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

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

Services can also be an instance of an ES6 class:

Promise
async/await

```
class MyService {
  find(params) {
    return Promise.resolve([]);
  }
  get(id, params) {}
  create(data, params) {}
  update(id, data, params) {}
  patch(id, data, params) {}
  remove(id, params) {}
  setup(app, path) {}
}
```

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

```
class MyService {
  async find(params) {
    return [];
  }
  async get(id, params) {}
  async create(data, params) {}
  async update(id, data, params) {}
  async patch(id, data, params) {}
  async remove(id, params) {}
  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 have to return a Promise or be declared as `async` and have the following parameters:

- `id` - the identifier for the resource. A resource is the data identified by a unique id.
- `data` - the resource data.
- `params` - can contain any extra parameters, for example the authenticated user.

> **Important:** `params.query` contains the query parameters from the client, either passed as URL query parameters (see the REST chapter) or through websockets (see Socket.io or Primus).

Once registered the service can be retrieved and used via app.service():

```
const myService = app.service('my-service');

myService.find().then(items => console.log('.find()', items));
myService.get(1).then(item => console.log('.get(1)', items));
```

Keep in mind that services don't have to use databases. You could easily replace the database in the example with a package that uses some API, like pulling in GitHub stars or stock ticker data.

> **Important:** This section describes the general use of service methods and how to implement them. They are already implemented by Feathers official database adapters. For specifics on how to use the database adapters see the database adapters common API.

.find(params)

`service.find(params) -> Promise` - retrieves a list of all resources from the service. Provider parameters will be passed as `params.query` .

```
app.use('/messages', {
  find(params) {
    return Promise.resolve([
      {
        id: 1,
        text: 'Message 1'
      }, {
        id: 2,
        text: 'Message 2'
      }
    ]);
  }
});
```

> **Note:** `find` does not have to return an array it can also return an object. The database adapters already do this for pagination.

.get(id, params)

`service.get(id, params) -> Promise` - retrieves a single resource with the given `id` from the service.

```
app.use('/messages', {
  get(id, params) {
    return Promise.resolve({
      id,
      text: `You have to do ${id}!`
    });
  }
});
```

.create(data, params)

`service.create(data, params) -> Promise` - creates a new resource with `data`. The method should return a Promise with the newly created data. `data` may also be an array.

```
app.use('/messages', {
  messages: [],

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

    return Promise.resolve(data);
  }
});
```

> **Important:** A successful `create` method call emits the `created` service event.

.update(id, data, params)

`service.update(id, data, params) -> Promise` - 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 with `params.query` containing the query criteria.

> **Important:** A successful `update` method call emits the `updated` service event.

.patch(id, data, params)

`patch(id, data, params) -> Promise` - 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 with `params.query` containing the query criteria.

The method should return with the complete updated resource data. Implement `patch` additionally (or instead of) `update` if you want to separate between partial and full updates and support the `PATCH` HTTP method.

> **Important:** A successful `patch` method call emits the `patched` service event.

.remove(id, params)

`service.remove(id, params) -> Promise` - 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 with `params.query` containing the query criteria.

> **Important:** A successful `remove` method call emits the `removed` service event.

.setup(app, path)

`service.setup(app, path) -> Promise` is a special method that initializes the service, passing an instance of the Feathers application and the path it has been registered on.

For services registered before `app.listen` is invoked, the `setup` function of each registered service is called upon invoking `app.listen`. For services registered after `app.listen` is invoked, `setup` 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('@feathersjs/feathers');
const rest = require('@feathersjs/express/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);
```

Feathers functionality

When registering a service, Feathers (or its plugins) can also add its own methods to a service. Most notably, every service will automatically become an instance of a NodeJS EventEmitter.

.hooks(hooks)

Register hooks for this service.

.publish([event,] publisher)

Register an event publishing callback. For more information see the channels chapter.

.mixin(mixin)

`service..mixin(mixin) -> service` allows to extend the functionality of a service. For more information see the Uberproto project page.

.on(eventname, listener)

Provided by the core NodeJS EventEmitter .on. Registers a `listener` method (`function(data) {}`) for the given `eventname` .

> **Important:** For more information about service event see the Events chapter.

.emit(eventname, data)

Provided by the core NodeJS EventEmitter .emit. Emits the event `eventname` to all event listeners.

> **Important:** For more information about service event see the Events chapter.

.removeListener(eventname, [listener])

Provided by the core NodeJS EventEmitter .removeListener. Removes all or the given listener for `eventname` .

> **Important:** For more information about service event see the Events chapter.

Hooks

Hooks are pluggable middleware functions that can be registered **before**, **after** or on **error**s of a service method. You can register a single hook function or create a chain of them to create complex work-flows. Most of the time multiple hooks are registered so the examples show the "hook chain" array style registration.

A hook is **transport independent**, which means it does not matter if it has been called through HTTP(S) (REST), Socket.io, Primus or any other transport 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.

Hooks are commonly used to handle things like validation, logging, populating related entities, sending notifications and more. This pattern keeps your application logic flexible, composable, and much easier to trace through and debug. For more information about the design patterns behind hooks see this blog post.

Quick Example

The following example adds a `createdAt` and `updatedAt` property before saving the data to the database and logs any errors on the service:

```
const feathers = require('@feathersjs/feathers');

const app = feathers();

app.service('messages').hooks({
  before: {
    create(context) {
      context.data.createdAt = new Date();
    },

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

    patch(context) {
      context.data.updatedAt = new Date();
    }
  },

  error(context) {
    console.error(`Error in ${context.path} calling ${context.method} method`, context.error);
  }
});
```

Hook functions

A hook function can be a normal or `async` function or arrow function that takes the hook context as the parameter and can

- return a `context` object
- return nothing (`undefined`)
- `throw` an error
- for asynchronous operations return a Promise that
  - resolves with a `context` object
  - resolves with `undefined`

- rejects with an error

```
// normal hook function
function(context) {
  return context;
}

// asynchronous hook function with promise
function(context) {
  return Promise.resolve(context);
}

// async hook function
async function(context) {
  return context;
}

// normal arrow function
context => {
  return context;
}

// asynchronous arrow function with promise
context => {
  return Promise.resolve(context);
}

// async arrow function
async context => {
  return context;
}
```

When an error is thrown (or the promise is rejected), all subsequent hooks - and the service method call if it didn't run already - will be skipped and only the error hooks will run.

The following example throws an error when the text for creating a new message is empty. You can also create very similar hooks to use your Node validation library of choice.

```
app.service('messages').hooks({
  before: {
    create: [
      function(context) {
        if(context.data.text.trim() === '') {
          throw new Error('Message text can not be empty');
        }
      }
    ]
  }
});
```

Hook context

The hook `context` is passed to a hook function and contains information about the service method call. It has **read only** properties that should not be modified and ***writeable*** properties that can be changed for subsequent hooks.

> **Pro Tip:** The `context` object is the same throughout a service method call so it is possible to add properties and use them in other hooks at a later time.

context.app

`context.app` is a *read only* property that contains the Feathers application object. This can be used to retrieve other services (via `context.app.service('name')`) or configuration values.

context.service

`context.service` is a *read only* property and contains the service this hook currently runs on.

context.path

`context.path` is a *read only* property and contains the service name (or path) without leading or trailing slashes.

context.method

`context.method` is a *read only* property with the name of the service method (one of `find` , `get` , `create` , `update` , `patch` , `remove`).

context.type

`context.type` is a *read only* property with the hook type (one of `before` , `after` or `error`).

context.params

`context.params` is a **writeable** property that contains the service method parameters (including `params.query`).

Important properties that usually are available in `params` :

- `context.params.query` - The query from the client
- `context.params.provider` - The transport (`rest` , `socketio` or `primus`) used for this service call. Will be `undefined` for internal calls from the server.

context.id

`context.id` is a **writeable** property and the `id` for a `get` , `remove` , `update` and `patch` service method call. For `remove` , `update` and `patch` `context.id` can also be `null` when modifying multiple entries. In all other cases it will be `undefined` .

> **Note:** `context.id` is only available for method types `get` , `remove` , `update` and `patch` .

context.data

`context.data` is a **writeable** property containing the data of a `create` , `update` and `patch` service method call.

> **Note:** `context.data` will only be available for method types `create` , `update` and `patch` .

context.error

`context.error` is a **writeable** property with the error object that was thrown in a failed method call. It is only available in `error` hooks.

> **Note:** `context.error` will only be available if `context.type` is `error` .

context.result

`context.result` is a **writeable** property containing the result of the successful service method call. It is only available in `after` hooks. `context.result` can also be set in

- A `before` hook to skip the actual service method (database) call
- An `error` hook to swallow the error and return a result instead

> **Note:** `context.result` will only be available if `context.type` is `after` or if `context.result` has been set.

context.dispatch

`context.dispatch` is a **writeable, optional** property and contains a "safe" version of the data that should be sent to any client. If `context.dispatch` has not been set `context.result` will be sent to the client instead.

> **Note:** `context.dispatch` only affects the data sent through a Feathers Transport like REST or Socket.io. An internal method call will still get the data set in `context.result`.

Asynchronous hooks

When the hook function is `async` or a Promise is returned it will wait until all asynchronous operations resolve or reject before continuing to the next hook.

> **Important:** As stated in the hook functions section the promise has to either resolve with the `context` object (usually done with `.then(() => context)` at the end of the promise chain) or with `undefined`.

async/await

When using Node v8.0.0 or later the use of async/await is highly recommended. This will avoid many common issues when using Promises and asynchronous hook flows. Any hook function can be `async` in which case it will wait until all `await` operations are completed. Just like a normal hook it should return the `context` object or `undefined`.

The following example shows an async/await hook that uses another service to retrieve and populate the messages `user` when getting a single message:

```
app.service('messages').hooks({
  after: {
    get: [
      async function(context) {
        const userId = context.result.userId;

        // Since context.app.service('users').get returns a promise we can `await` it
        const user = await context.app.service('users').get(userId);

        // Update the result (the message)
        context.result.user = user;

        // Returning will resolve the promise with the `context` object
        return context;
      }
    ]
  }
});
```

Returning promises

The following example shows an asynchronous hook that uses another service to retrieve and populate the messages `user` when getting a single message.

```
app.service('messages').hooks({
  after: {
    get: [
      function(context) {
```

```
        const userId = context.result.userId;

        // context.app.service('users').get returns a Promise already
        return context.app.service('users').get(userId).then(user => {
          // Update the result (the message)
          context.result.user = user;

          // Returning will resolve the promise with the `context` object
          return context;
        });
      }
    ]
  }
});
```

> **Note:** A common issue when hooks are not running in the expected order is a missing `return` statement of a promise at the top level of the hook function.
>
> **Important:** Most Feathers service calls and newer Node packages already return Promises. They can be returned and chained directly. There is no need to instantiate your own `new` Promise instance in those cases.

Converting callbacks

When the asynchronous operation is using a *callback* instead of returning a promise you have to create and return a new Promise (`new Promise((resolve, reject) => {})`) or use util.promisify.

The following example reads a JSON file converting fs.readFile with `util.promisify` :

```
const fs = require('fs');
const utils = require('utils');
const readFile = utils.promisify(fs.readFile);

app.service('messages').hooks({
  after: {
    get: [
      function(context) {
        return readFile('./myfile.json').then(data => {
          context.result.myFile = data.toString();

          return context;
        });
      }
    ]
  }
});
```

> **Pro Tip:** Other tools like Bluebird also help converting between callbacks and promises.

Registering hooks

Hook functions are registered on a service through the `app.service(<servicename>).hooks(hooks)` method. There are several options for what can be passed as `hooks` :

```
// The standard all at once way (also used by the generator)
// an array of functions per service method name (and for `all` methods)
app.service('servicename').hooks({
  before: {
    all: [
      // Use normal functions
      function(context) { console.log('before all hook ran'); }
    ],
```

```
    find: [
      // Use ES6 arrow functions
      context => console.log('before find hook 1 ran'),
      context => console.log('before find hook 2 ran')
    ],
    get: [ /* other hook functions here */ ],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
  after: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
  error: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  }
});

// Register a single hook before, after and on error for all methods
app.service('servicename').hooks({
  before(context) {
    console.log('before all hook ran');
  },
  after(context) {
    console.log('after all hook ran');
  },
  error(context) {
    console.log('error all hook ran');
  }
});
```

> **Pro Tip:** When using the full object, `all` is a special keyword meaning this hook will run for all methods. `all` hooks will be registered before other method specific hooks.
>
> **Pro Tip:** `app.service(<servicename>).hooks(hooks)` can be called multiple times and the hooks will be registered in that order. Normally all hooks should be registered at once however to see at a glance what what the service is going to do.

Application hooks

To add hooks to every service `app.hooks(hooks)` can be used. Application hooks are registered in the same format as service hooks and also work exactly the same. Note when application hooks will be executed however:

- `before` application hooks will always run *before* all service `before` hooks
- `after` application hooks will always run *after* all service `after` hooks
- `error` application hooks will always run *after* all service `error` hooks

Here is an example for a very useful application hook that logs every service method error with the service and method name as well as the error stack.

```
app.hooks({
  error(context) {
    console.error(`Error in '${context.path}' service method '${context.method}`, context.error.stack);
  }
});
```

Events

Events are the key part of Feathers real-time functionality. All events in Feathers are provided through the NodeJS EventEmitter interface. This section describes

- A quick overview of the NodeJS EventEmitter interface
- The standard service events
- How to allow sending custom events from the server to the client

> **Important:** For more information on how to send real-time events to clients, see the Channels chapter.

EventEmitters

Once registered, any service gets turned into a standard NodeJS EventEmitter and can be used accordingly.

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

// Listen to a normal service event
messages.on('patched', message => console.log('message patched', message));

// Only listen to an event once
messsages.once('removed', message =>
  console.log('First time a message has been removed', message)
);

// A reference to a handler
const onCreatedListener = message => console.log('New message created', message);

// Listen `created` with a handler reference
messages.on('created', onCreatedListener);

// Unbind the `created` event listener
messages.removeListener('created', onCreatedListener);

// Send a custom event
messages.emit('customEvent', {
  type: 'customEvent',
  data: 'can be anything'
});
```

Service Events

Any service automaticaly emits `created`, `updated`, `patched` and `removed` events when the respective service method returns successfully. This works on the client as well as on the server. When the client is using Socket.io or Primus, events will be pushed automatically from the server to all connected client. This is essentially how Feathers does real-time.

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

> **Important:** For information on how those events are published for real-time updates to connected clients, see the channel chapter.

Additionally to the event `data`, all events also get the hook context from their method call passed as the second parameter.

created

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

```
const feathers = require('@feathersjs/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, context) => 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('@feathersjs/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, context) => 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('@feathersjs/feathers');
const app = feathers();

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

```
  }
});

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

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

Custom events

By default, real-time clients will only receive the standard events. However, it is possible to define a list of custom events on a service as `service.events` that should also be passed. The `context` for custom events won't be a full hook context but just an object containing `{ app, service, path, result }`.

> **Important:** The database adapters also take a list of custom events as an initialization option.

> **Important:** Custom events can only be sent from the server to the client, not the other way (client to server). Learn more

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' });
      });
    });
  }
}
```

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

Event channels

On a Feathers server with a real-time transport (Socket.io or Primus) set up, event channels determine which connected clients to send real-time events to and how the sent data should look like.

This chapter describes:

- Real-time Connections and how to acces them
- Channel usage and how to retrieve, join and leave channels
- Publishing events to channels

> **Important:** If you are not using a real-time transport server (e.g. when making a REST only API or using Feathers on the client), channel functionality is not going to be available.

Some examples where channels are used:

- Real-time events should only be sent to authenticated users
- Users should only get updates about messages if they joined a certain chat room
- Only users in the same organization should receive real-time updates about their data changes
- Only admins should be notified when new users are created
- When a user is created, modified or removed, non-admins should only receive a "safe" version of the user object (e.g. only `email`, `id` and `avatar`)

Example

The example below shows the generated `channels.js` file illustrating how the different parts fit together:

```
module.exports = function(app) {
  app.on('connection', connection => {
    // On a new real-time connection, add it to the
    // anonymous channel
    app.channel('anonymous').join(connection);
  });

  app.on('login', (user, { connection }) => {
    // connection can be undefined if there is no
    // real-time connection, e.g. when logging in via REST
    if(connection) {
      // The connection is no longer anonymous, remove it
      app.channel('anonymous').leave(connection);

      // Add it to the authenticated user channel
      app.channel('authenticated').join(connection);

      // Channels can be named anything and joined on any condition
      // E.g. to send real-time events only to admins use

      // if(user.isAdmin) { app.channel('admins').join(connection); }

      // If the user has joined e.g. chat rooms

      // user.rooms.forEach(room => app.channel(`rooms/${room.id}`).join(channel))
    }
  });

  // A global publisher that sends all events to all authenticated clients
  app.publish((data, context) => {
    return app.channel('authenticated');
  });
```

```
};
```

Connections

A connection is an object that represents a real-time connection. It is the same object as `socket.feathers` in a Socket.io and `socket.request.feathers` in a Primus middleware. You can add any kind of information to it but most notably, when using authentication, it will contain the authenticated user. By default it is located in `connection.user` once the client has authenticated on the socket (usually by calling `app.authenticate()` on the client).

We can get access to the `connection` object by listening to `app.on('connection', connection => {})` or `app.on('login', (user, { connection }) => {})`.

> **Note:** When a connection is terminated it will be automatically removed from all channels.

app.on('connection')

`app.on('connection', connection => {})` is fired every time a new real-time connection is established. This is a good place to add the connection to a channel for anonymous users (in case we want to send any real-time updates to them):

```
app.on('connection', connection => {
  // On a new real-time connection, add it to the
  // anonymous channel
  app.channel('anonymous').join(connection);
});
```

app.on('login')

`app.on('login', (user, info) => {})` is sent by the authentication module also contains the connection in the `meta` object that is passed as the second parameter. Note that it can also be `undefined` if the login happened through e.g. REST which does not support real-time connectivity.

This is a good place to add the connection to channels related to the user (e.g. chat rooms, admin status etc.)

```
app.on('login', (user, { connection }) => {
  // connection can be undefined if there is no
  // real-time connection, e.g. when logging in via REST
  if(connection) {
    // The connection is no longer anonymous, remove it
    app.channel('anonymous').leave(connection);

    // Add it to the authenticated user channel
    app.channel('authenticated').join(connection);

    // Channels can be named anything and joined on any condition
    // E.g. to send real-time events only to admins use
    if(user.isAdmin) {
      app.channel('admins').join(connection);
    }

    // If the user has joined e.g. chat rooms
    user.rooms.forEach(room => {
      app.channel(`rooms/${room.id}`).join(channel);
    });
  }
});
```

> **Note:** `(user, { connection })` is an ES6 shorthand for `(user, meta) => { const connection = meta.connection; }`, see Destructuring assignment.

Channels

A channel is an object that contains a number of connections. It can be created via `app.channel` and allows a connection to join or leave it.

app.channel(...names)

`app.channel(name) -> Channel`, when given a single name, returns an existing or new named channel:

```
app.channel('admins') // the admin channel
app.channel('authenticated') // the authenticated channel
```

`app.channel(name1, name2, ... nameN) -> Channel`, when given multiples names, will return a combined channel. A combined channel contains a list of all connections (without duplicates) and re-directs `channel.join` and `channel.leave` calls to all its child channels.

```
// Combine the anonymous and authenticated channel
const combinedChannel = app.channel('anonymous', 'authenticated')

// Join the `admins` and `chat` channel
app.channel('admins', 'chat').join(connection);

// Leave the `admins` and `chat` channel
app.channel('admins', 'chat').leave(connection);

// Make user with `_id` 5 leave the admins and chat channel
app.channel('admins', 'chat').leave(connection => {
  return connection.user._id === 5;
});
```

app.channels

`app.channels -> [string]` returns a list of all existing channel names.

```
app.channel('authenticated');
app.channel('admins', 'users');

app.channels // [ 'authenticated', 'admins', 'users' ]

app.channel(app.channels) // will return a channel with all connections
```

This is useful to e.g. remove a connection from all channels:

```
// When a user is removed, make all their connections leave every channel
app.service('users').on('removed', user => {
  app.channel(app.channels).leave(connection => {
    return user._id === connection.user._id;
  });
});
```

channel.join(connection)

`channel.join(connection) -> Channel` adds a connection to this channel. If the channel is a combined channel, add the connection to all its child channels. If the connection is already in the channel it does nothing. Returns the channel object.

```
app.on('login', (user, { connection }) => {
  if(connection && user.isAdmin) {
    // Join the admins channel
    app.channel('admins').join(connection);

    // Calling a second time will do nothing
    app.channel('admins').join(connection);
  }
});
```

channel.leave(connection|fn)

`channel.leave(connection|fn) -> Channel` removes a connection from this channel. If the channel is a combined channel, remove the connection from all its child channels. Also allows to pass a callback that is run for every connection and returns if the connection should be removed or not. Returns the channel object.

```
// Make the user with `_id` 5 leave the `admins` channel
app.channel('admins').leave(connection => {
  return connection.user._id === 5;
});
```

channel.filter(fn)

`channel.filter(fn) -> Channel` returns a new channel filtered by a given function which gets passed the connection.

```
// Returns a new channel with all connections of the user with `_id` 5
const userFive = app.channel(app.channels)
  .filter(connection => connection.user._id === 5);
```

channel.send(data)

`channel.send(data) -> Channel` returns a copy of this channel with customized data that should be sent for this event. Usually this should be handled by modifying either the service method result or setting client "safe" data in `context.dispatch` but in some cases it might make sense to still change the event data for certain channels.

What data will be sent as the event data will be determined by the first available in the following order:

1. `data` from `channel.send(data)`
2. `context.dispatch`
3. `context.result`

```
app.on('connection', connection => {
  // On a new real-time connection, add it to the
  // anonymous channel
  app.channel('anonymous').join(connection);
});

// Send the `users` `created` event to all anonymous
// users but use only the name as the payload
app.service('users').publish('created', data => {
  return app.channel('anonymous').send({
    name: data.name
  });
});
```

> **Note:** If a connection is in multiple channels (e.g. `users` and `admins`) it will get the data from the *first* channel that it is in.

channel.connections

`channel.connections -> [ object ]` contains a list of all connections in this channel.

channel.length

`channel.length -> integer` returns the total number of connections in this channel.

Publishing

Publishers are callback functions that return which channel(s) to send an event to. They can be registered at the application and the service level and for all or specific events. A publishing function gets the event data and context object (`(data, context) => {}`) and returns a named or combined channel or an array of channels. Multiple publishers can be registered. Besides the standard service event names an event name can also be a custom event. `context` is the context object from the service call or an object containing `{ path, service, app, result }` for custom events.

service.publish([event,] fn)

`service.publish([event,] fn) -> service` registers a publishing function for a specific service for a specific event or all events if no event name was given.

```
app.on('login', (user, { connection }) => {
  // connection can be undefined if there is no
  // real-time connection, e.g. when logging in via REST
  if(connection && user.isAdmin) {
    app.channel('admins').join(connection);
  }
});

// Publish all messages service events only to its room channel
app.service('messages').publish((data, context) => {
  return app.channel(`rooms/${data.roomId}`);
});

// Publish the `created` event only to admins
app.service('users').publish('created', (data, context) => {
  return app.channel('admins');
});
```

app.publish([event,] fn)

`app.publish([event,] fn) -> app` registers a publishing function for all services for a specific event or all events if no event name was given.

```
app.on('login', (user, { connection }) => {
  // connection can be undefined if there is no
  // real-time connection, e.g. when logging in via REST
  if(connection) {
    app.channel('authenticated').join(connection);
  }
});
```

```
// Publish all events to all authenticated users
app.publish((data, context) => {
  return app.channel('authenticated');
});

// Publish the `log` custom event to all connections
app.publish('log', (data, context) => {
  return app.channel(app.channels);
});
```

Errors


```
$ npm install @feathersjs/errors --save
```

The `@feathersjs/errors` module contains a set of standard error classes used by all other Feathers modules as well as an Express error handler to format those - and other - errors and setting the correct HTTP status codes for REST calls.

Feathers errors

The following error types, all of which are instances of `FeathersError` are available:

> **ProTip:** All of the Feathers plugins will automatically emit the appropriate Feathers errors for you. For example, most of the database adapters will already send `Conflict` or `Unprocessable` errors with the validation errors from the ORM.

- `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

Feathers errors are pretty flexible. They contain the following fields:

- `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.

> **ProTip:** To convert a Feathers error back to an object call `error.toJSON()` . A normal `console.log` of a JavaScript Error object will not automatically show those additional properties described above (even though they can be accessed directly).

Examples

Here are a few ways that you can use them:

```
const errors = require('@feathersjs/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'
  }
});
```

Server Side Errors

Promises swallow errors if you forget 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);
});
```

Configuration


```
$ npm install @feathersjs/configuration --save
```

`@feathersjs/configuration` is a wrapper for node-config which allows to configure a server side Feathers application.

By default this implementation will look in `config/*` for `default.json` which retains convention. As per the config docs you can organize *"hierarchical configurations for your app deployments"*. See the usage section below for better information how to implement this.

Usage

The `@feathersjs/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('@feathersjs/feathers');
const configuration = require('@feathersjs/configuration');

// Use the application roo and `config/` as the configuration folder
let app = feathers().configure(configuration())
```

Variable types

`@feathersjs/configuration` uses the following variable mechanisms:

- 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
- 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 starts with `./` or `../` turn it into an absolute path relative to the configuration file path
  - If the value is escaped (starting with a `\`) always use that value (e.g. `\\NODE_ENV` will become `NODE_ENV`)
- Both `default` and `<env>` configurations can be modules which provide their computed settings with `module.exports = {...}` and a `.js` file suffix. See `test/config/testing.js` for an example.
  All rules listed above apply for `.js` modules.

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('@feathersjs/feathers');
const configuration = require('@feathersjs/configuration');

let conf = configuration();

let app = feathers()
  .configure(conf);

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'));
console.log(conf());
```

If you now run

```
node app
// -> path/to/app/public
// -> localhost
// -> 3030
// -> mongodb://localhost:27017/myapp
// -> path/to/templates
```

Or via custom environment variables by setting them in `config/custom-environment-variables.json`:

```
{
  "port": "PORT",
  "mongodb": "MONGOHQ_URL"
}
```

```
$ PORT=8080 MONGOHQ_URL=mongodb://localhost:27017/production NODE_ENV=production node app
// -> path/to/app/public/dist
// -> myapp.com
// -> 8080
// -> mongodb://localhost:27017/production
// -> path/to/templates
```

You can also override these variables with arguments. Read more about how with node-config

Express


```
$ npm install @feathersjs/express --save
```

The `@feathersjs/express` module contains Express framework integrations for Feathers:

- The Express framework bindings to make a Feathers application Express compatible
- An Express based transport to expose services throuhg a REST API
- An Express error handler for Feathers errors

```
const express = require('@feathersjs/express');
```

> **Very Important:** This page describes how to set up an Express server and REST API. See the REST client chapter how to use this server on the client.
>
> **Important:** This chapter assumes that you are familiar with Express.

express(app)

`express(app) -> app` is a function that turns a Feathers application into a fully Express (4+) compatible application that additionally to Feathers functionality also lets you use the Express API.

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');

// Create an app that is a Feathers AND Express application
const app = express(feathers());
```

Note that `@feathersjs/express` (`express`) also exposes the standard Express middleware:

- `express.json` - A JSON body parser
- `express.urlencoded` - A URL encoded form body parser
- `express.static` - To statically host files in a folder
- `express.Router` - Creates an Express router object

app.use(path, service|mw)

`app.use(path, service|mw) -> app` registers either a service object or an Express middleware on the given path. If a service object is passed it will use Feathers registration mechanism, for a middleware function Express.

```
// Register a service
app.use('/todos', {
  get(id) {
    return Promise.resovle({ id });
  }
});

// Register an Express middleware
app.use('/test', (req, res) => {
  res.json({
```

```
    message: 'Hello world from Express middleware'
  });
});
```

app.listen(port)

`app.listen(port) -> HttpServer` will first call Express app.listen and then internally also call the Feathers app.setup(server).

```
// Listen on port 3030
const server = app.listen(3030);

server.on('listening', () => console.log('Feathers application started'));
```

app.setup(server)

`app.setup(server) -> app` is usually called internally by `app.listen` but in the cases described below needs to be called explicitly.

Sub-Apps

When registering an application as a sub-app, `app.setup(server)` has to be called to initialize the sub-apps services.

```
const express = require('express');
const feathers = require('@feathersjs/feathers');
const feathersExpress = require('@feathersjs/express');

const api = feathersExpress(feathers())
  .configure(feathersExpress.rest())
  .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);
```

> **ProTip:** We recommend avoiding complex sub-app setups because websockets and Feathers built in authentication are not fully sub-app aware at the moment.

HTTPS

HTTPS requires creating a separate server in which case `app.setup(server)` also has to be called explicitly.

```
const fs = require('fs');
const https  = require('https');

const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');

const app = express(feathers());

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

The vhost Express middleware can be used to run a Feathers application on a virtual host but again requires `app.setup(server)` to be called explicitly.

```
const express = require('express');
const vhost = require('vhost');

const feathers = require('@feathersjs/feathers');
const feathersExpress = require('@feathersjs/express');

const app = feathersExpress(feathers());

app.use('/todos', todoService);

const host = express().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);
```

express.rest()

`express.rest` registers a Feathers transport mechanism that allows you to expose and consume services through a RESTful API. This means that you can call a service method through the `GET`, `POST`, `PUT`, `PATCH` and `DELETE` HTTP methods:

Service method	HTTP method	Path
.find()	GET	/messages
.get()	GET	/messages/1
.create()	POST	/messages
.update()	PUT	/messages/1
.patch()	PATCH	/messages/1
.remove()	DELETE	/messages/1

To expose services through a RESTful API we will have to configure `express.rest` and provide our own body parser middleware (usually the standard Express 4 body-parser) to make REST `.create`, `.update` and `.patch` calls parse the data in the HTTP body. If you would like to add other middleware *before* the REST handler, call `app.use(middleware)` before registering any services.

> **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.

app.configure(express.rest())

Configures the transport provider with a standard formatter sending JSON response via res.json.

```
const feathers = require('@feathersjs/feathers');
```

```
const express = require('@feathersjs/express');

// Create an Express compatible Feathers application
const app = express(feathers());

// Turn on JSON parser for REST services
app.use(express.json())
// Turn on URL-encoded parser for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport
app.configure(express.rest())
```

app.configure(express.rest(formatter))

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 pass a `formatter(req, res)` function. This middleware will have access to `res.data` which is the data returned by the service. res.format can be used for content negotiation.

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');

const app = feathers();

// Turn on JSON parser for REST services
app.use(express.json())
// Turn on URL-encoded parser for REST services
app.use(express.urlencoded({ extended: true }));
// Set up REST transport
app.configure(express.rest(function(req, res) {
  // Format the message as text/plain
  res.format({
    'text/plain': function() {
      res.end(`The Message is: "${res.data.text}"`);
    }
  });
}))
```

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 has the service call information available as

- `res.data` - The data that will be sent
- `res.hook` - The hook context of the service method call

For example `updateData` could look like this:

```
function updateData(req, res, next) {
```

```
  res.data.updateData = true;
  next();
}
```

> **ProTip:** If you run `res.send` in a custom middleware after the service and don't call `next`, other middleware (like the REST formatter) will be skipped. This can be used to e.g. render different views for certain service method calls.

params

All middleware registered after the REST transport will have access to the `req.feathers` object to set properties on the service method `params`:

```
const app = require('@feathersjs/feathers')();
const rest = require('@feathersjs/express/rest');
const bodyParser = require('body-parser');

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}!`
    });
  }
});

app.listen(3030);
```

You can see the parameters set by running the example and visiting `http://localhost:3030/todos/test`.

Avoid 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.

> **Very important:** Since the order of Express middleware matters, any middleware that sets service parameters has to be registered *before* your services (in a generated application before `app.configure(services)` or in `middleware/index.js`).
>
> **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.

params.query

`params.query` will contain the URL query parameters sent from the client. For the REST transport the query string is parsed using the qs module. For some query string examples see the database querying chapter.

> **Important:** Only `params.query` is passed between the server and the client, other parts of `params` are not. This is for security reasons so that a client can't set things like `params.user` or the database options. You can always map from `params.query` to other `params` properties in a before hook.

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.
>
> **Note:** If an array in your request consists of more than 20 items, the qs parser implicitly converts it to an object with indices as keys. To extend this limit, you can set a custom query parser: `app.set('query parser', str => qs.parse(str, {arrayLimit: 1000}))`

`params.provider`

For any service method call made through REST `params.provider` will be set to `rest`. In a hook this can for example be used to prevent external users from making a service method call:

```
app.service('users').hooks({
  before: {
    remove(context) {
      // check for if(context.params.provider) to prevent any external call
      if(context.params.provider === 'rest') {
        throw new Error('You can not delete a user via REST');
      }
    }
  }
});
```

params.route

See the routing section.

express.notFound()

`express.notFound()` returns middleware that returns a `NotFound` (404) Feathers error. It should be used as the last middleware **before** the error handler.

express.errorHandler()

`expres.errorHandler` is an Express error handler middleware that formats any error response to a REST call as JSON (or HTML if e.g. someone hits our API directly in the browser) and sets the appropriate error code.

> **ProTip:** You can still use any other Express compatible error middleware with Feathers. In fact, the `express.errors` is just a slightly customized one. **Very Important:** Just as in Express, the error handler has to be registered *after* all middleware and services.

`app.use(express.errorHandler())`

Set up the error handler with the default configuration.

```
const errorHandler = require('@feathersjs/express/errors');
const app = feathers();

// before starting the app
app.use(errorHandler())
```

`app.use(express.errorHandler(options))`

```
const error = require('@feathersjs/errors');
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.

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.

Routing

Express route placeholders in a service URL will be added to the services `params.route`.

> **Important:** See the FAQ entry on nested routes for more details on when and when not to use nested routes.

```
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.route.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);
```

Socket.io


```
$ npm install @feathersjs/socketio --save
```

The @feathersjs/socketio module allows to call service methods and receive real-time events via Socket.io, a NodeJS library which enables real-time bi-directional, event-based communication.

> **Important:** This page describes how to set up a Socket.io server. The Socket.io client chapter shows how to connect to this server on the client and the message format for service calls and real-time events.

Configuration

`@feathersjs/socketio` can be used standalone or together with a Feathers framework integration like Express.

app.configure(socketio())

Sets up the Socket.io transport with the default configuration using either the server provided by app.listen or passed in app.setup(server).

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio');

const app = feathers();

app.configure(socketio());

app.listen(3030);
```

> **Pro tip:** Once the server has been started with `app.listen()` or `app.setup(server)` the Socket.io object is available as `app.io`.

app.configure(socketio(callback))

Sets up the Socket.io transport with the default configuration and call `callback` with the Socket.io server object. This is a good place to listen to custom events or add authorization:

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio');

const app = feathers();

app.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);
```

app.configure(socketio(options [, callback]))

Sets up the Socket.io transport with the given Socket.io options object and optionally calls the callback described above.

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('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio');

const app = feathers();

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

app.listen(3030);
```

app.configure(socketio(port, [options], [callback]))

Creates a new Socket.io server on a separate port. Options and a callback are optional and work as described above.

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio');

const app = feathers();

app.configure(socketio(3031));
app.listen(3030);
```

params

Socket.io middleware can modify the `feathers` property on the `socket` which will then be used as the service call `params` :

```
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' }
  }
});
```

params.provider

For any service method call made through Socket.io `params.provider` will be set to `socketio`. In a hook this can for example be used to prevent external users from making a service method call:

```
app.service('users').hooks({
  before: {
    remove(context) {
      // check for if(context.params.provider) to prevent any external call
      if(context.params.provider === 'socketio') {
        throw new Error('You can not delete a user via Socket.io');
      }
    }
  }
});
```

params.query

`params.query` will contain the query parameters sent from the client.

> **Important:** Only `params.query` is passed between the server and the client, other parts of `params` are not. This is for security reasons so that a client can't set things like `params.user` or the database options. You can always map from `params.query` to `params` in a before hook.

uWebSocket

The options can also be used to initialize uWebSocket which is a WebSocket server implementation that provides better performace and reduced latency.

```
$ npm install uws --save
```

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio');

const app = feathers();

app.configure(socketio({
  wsEngine: 'uws'
}));

app.listen(3030);
```

Primus


```
$ npm install @feathersjs/primus --save
```

The @feathersjs/primus module allows to call service methods and receive real-time events via Primus, a universal wrapper for real-time frameworks that supports Engine.IO, WebSockets, Faye, BrowserChannel, SockJS and Socket.IO.

> **Important:** This page describes how to set up Primus server. The Primus client chapter shows how to connect to this server on the client and the message format for service calls and real-time events.

Configuration

Additionally to `@feathersjs/primus` your websocket library of choice also has to be installed.

```
$ npm install ws --save
```

app.configure(primus(options))

Sets up the Primus transport with the given Primus options.

> **Pro tip:** Once the server has been started with `app.listen()` or `app.setup(server)` the Primus server object is available as `app.primus`.

```
const feathers = require('@feathersjs/feathers');
const primus = require('@feathersjs/primus');

const app = feathers();

// Set up Primus with SockJS
app.configure(primus({ transformer: 'ws' }));

app.listen(3030);
```

app.configure(primus(options, callback))

Sets up the Primus transport with the given Primus options and calls the callback with the Primus server instance.

```
const feathers = require('@feathersjs/feathers');
const primus = require('@feathersjs/primus');

const app = feathers();

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

app.listen(3030);
```

params

The Primus request object has a `feathers` property that can be extended with additional service `params` during authorization:

```
app.configure(primus({
  transformer: 'ws'
}, function(primus) {
  // Do something with primus
  primus.use('feathers-referrer', function(req, res){
    // Exposing a request property to services and hooks
    req.feathers.referrer = request.referrer;
  });
}));

app.use('messages', {
  create(data, params, callback) {
    // When called via Primus:
    params.referrer // referrer from request
  }
});
```

params.provider

For any service method call made through a Primus socket `params.provider` will be set to `primus`. In a hook this can for example be used to prevent external users from making a service method call:

```
app.service('users').hooks({
  before: {
    remove(context) {
      // check for if(context.params.provider) to prevent any external call
      if(context.params.provider === 'primus') {
        throw new Error('You can not delete a user via Primus');
      }
    }
  }
});
```

params.query

`params.query` will contain the query parameters sent from the client.

> **Important:** Only `params.query` is passed between the server and the client, other parts of `params` are not. This is for security reasons so that a client can't set things like `params.user` or the database options. You can always map from `params.query` to `params` in a before hook.

Feathers Client

One of the most notable features of Feathers is that it can also be used as the client. The difference to many other frameworks is that it isn't a separate library, you instead get the exact same functionality as on the server. This means you can use services and hooks and configure plugins. By default a Feathers client automatically creates services that talk to a Feathers server.

In order to connect to a Feathers server, a client creates Services that use a REST or websocket connection to relay method calls and allow listening to events on the server. This means the Feathers application instance instance usable the exact same way as you would on the server.

Modules relevant for use on the client are

- `@feathersjs/feathers` to initialize a new Feathers application
- @feathersjs/rest-client.
- @feathersjs/socketio-client to connect to services through Socket.io.
- @feathersjs/primus-client to connect to services through Primus.
- @feathersjs/authentication-client to authenticate a client

> **Important:** You do not have to use Feathers on the client to connect to a Feathers server. The client chapters above also describe how to use a REST HTTP, Socket.io or Primus connection directly without Feathers on the client side. For more information regarding authentication see the Authentication client chapter.

This chapter describes how to set up Feathers as the client in Node, React Native and in the browser with a module loader like Webpack or Browserify or through a `<script>` tag. The examples are using the Socket.io client. For other connection methods see the chapters linked above.

> **Important:** Feathers can be used on the client throught the individual modules or the @feathersjs/client module which combines the modules mentioned above into a single, ES5 transpiled version.

Node

To connect to a Feathers server in NodeJS install the client connection library needed (here `socket.io-client`), Feathers core and appropriate client library:

```
npm install @feathersjs/feathers @feathersjs/socketio-client socket.io-client --save
```

Then initialize like this:

```
const io = require('socket.io-client');
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio-client');

const socket = io('http://api.my-feathers-server.com', {
  transports: ['websocket'],
  forceNew: true
});
const client = feathers();

client.configure(socketio(socket));

const messageService = client.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'
});
```

React Native

React Native usage is the same as for the Node client. Install the required packages into your React Native project.

```
$ npm install @feathers/feathers @feathers/socketio-client socket.io-client
```

Then in the main application file:

```
import io from 'socket.io-client';
import feathers from '@feathersjs/feathers';
import socketio from '@feathersjs/socketio-client';

const socket = io('http://api.my-feathers-server.com', {
  transports: ['websocket'],
  forceNew: true
});
const client = feathers();

client.configure(socketio(socket));

const messageService = client.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'
});
```

As React Native for Android doesn't handle well timeouts longer than a minute, you might want to set lower values for `pingInterval` and `pingTimeout` of `feathers-socketio` on your server, which will stop warnings related to this issue. For example:

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

app.configure(socketio({
  pingInterval: 10000,
  pingTimeout: 50000
}));
```

Module loaders

All modules in the `@feathersjs` namespace are using ES6 and have to be transpiled for target browsers without full ES6 support. Most client side module loader exclude the `node_modules` folder from being transpiled and have to be configured to include modules in the `@feathersjs` namespace.

Webpack

For Webpack, the recommended `babel-loader` rule normally excludes everything in `node_modules`. It has to be adjusted to skip `node_modules/@feathersjs`. In the `module` `rules` in your `webpack.config.js` update the `babel-loader` section to this:

```
{
  test: /\.jsx?$/,
  exclude: /node_modules(\/|\\)(?!(@feathersjs))/,
  loader: 'babel-loader'
}
```

create-react-app

create-react-app uses Webpack but does not allow to modify the configuration unless you eject. If you do not want to eject, use the @feathersjs/client module instead.

Browserify

In Browserify the babelify transform has to be used. All Feathers packages indicate that they need the transform and will be then transpiled automatically.

Others

As mentioned above, `node_modules/@feathersjs` and all its subfolders have to be included in the ES6+ transpilation step when using any module loader that is using a transpiler. For non-CommonJS formats (like AMD) and an ES5 compatible version of Feathers and its client modules you can use the @feathersjs/client module.

@feathersjs/client


```
$ npm install @feathersjs/client --save
```

`@feathersjs/client` is a module that bundles the separate Feathers client side modules into one providing the code as ES5 which is compatible with modern browsers (IE10+). It can also be used directly int the browser through a `<script>` tag. Here is a table of which Feathers client module is included:

Feathers module	@feathersjs/client
@feathers/feathers	feathers (default)
@feathers/errors	feathers.errors
@feathersjs/rest-client	feathers.rest
@feathersjs/socketio-client	feathers.socketio
@feathersjs/primus-client	feathers.primus
@feathersjs/authentication-client	feathers.authentication

> **Important:** The Feathers client libraries come transpiled to ES5 and 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>`

> **Important:** When you are loading @feathersjs/client you do **not** have install or load any of the other modules listed in the table above.

When to use

`@feathersjs/client` can be used directly in the browser using a `<script>` tag without a module loader as well as with module loaders that do not support CommonJS (like RequireJS) or React applications created with a default `create-react-app`.

If you are using the Feathers client with Node or React Native you should follow the steps described in the Node and React Native sections and **not** use `@feathersjs/client`.

> **Note:** All Feathers client examples show direct usage and usage with `@feathersjs/client`.

Load with a module loader

You can use `@feathersjs/client` with other browser module loaders/bundlers (instead of using the modules directly) but it may include packages you may not use and result in a slightly larger bundle size.

```
import feathers from '@feathersjs/feathers';

// Socket.io is exposed as the `io` global.
const socket = io('http://localhost:3030', {
  transports: ['websocket']
});
// @feathersjs/client is exposed as the `feathers` global.
const app = feathers();

app.configure(feathers.socketio(socket));
app.configure(feathers.authentication());

app.service('messages').create({
  text: 'A new message'
});

// feathers.errors is an object with all of the custom error types.
```

Load from CDN with `<script>`

Below is an example of the scripts you would use to load `@feathersjs/client` from unpkg.com.

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
<script src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
<script src="//unpkg.com/socket.io-client@1.7.3/dist/socket.io.js"></script>
<script>
  // Socket.io is exposed as the `io` global.
  var socket = io('http://localhost:3030', {
    transports: ['websocket']
  });
  // @feathersjs/client is exposed as the `feathers` global.
  var app = feathers();

  app.configure(feathers.socketio(socket));
  app.configure(feathers.authentication());

  app.service('messages').create({
    text: 'A new message'
  });

  // feathers.errors is an object with all of the custom error types.
</script>
```

RequireJS

Here is an example of loading feathers-client using RequireJS Syntax:

```
define(function (require) {
  const feathers = require('@feathersjs/client');
  const { socketio, authentication } = feathers;
  const io = require('socket.io-client');

  const socket = io('http://localhost:3030', {
    transports: ['websocket']
  });
  // @feathersjs/client is exposed as the `feathers` global.
  const app = feathers();

  app.configure(socketio(socket));
  app.configure(authentication());

  app.service('messages').create({
    text: 'A new message'
  });

  return const;
});
```

REST Client

> **Note:** For directly using a Feathers REST API (via HTTP) without using Feathers on the client see the HTTP API section.

@feathersjs/rest-client


```
$ npm install @feathersjs/rest-client --save
```

`@feathersjs/rest-client` allows to connect to a service exposed through the Expres REST API using jQuery, request, Superagent, Axios or Fetch as the AJAX 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.

> **Note:** A client application can only use a single transport (either REST, Socket.io or Primus). Using two transports in the same client application is normally not necessary.

rest([baseUrl])

REST client services can be initialized by loading `@feathersjs/rest-client` and initializing a client object with a base URL:

Modular
@feathersjs/client

```
const feathers = require('@feathersjs/feathers');
const rest = require('@feathersjs/rest-client');

const app = feathers();

// Connect to the same as the browser URL (only in the browser)
const restClient = rest();

// Connect to a different URL
const restClient = rest('http://feathers-api.com')

// Configure an AJAX library (see below) with that client
app.configure(restClient.fetch(window.fetch));

// Connect to the `http://feathers-api.com/messages` service
const messages = app.service('messages');
```

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
<script src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
<script>
  var app = feathers();
  // Connect to a different URL
  var restClient = feathers.rest('http://feathers-api.com')

  // Configure an AJAX library (see below) with that client
  app.configure(restClient.fetch(window.fetch));

  // Connect to the `http://feathers-api.com/messages` service
```

```
  const messages = app.service('messages');
</script>
```

> **ProTip:** In the browser, 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')`.

params.headers

Request specific headers can be through `params.headers` in a service call:

```
app.service('messages').create({
  text: 'A message from a REST client'
}, {
  headers: { 'X-Requested-With': 'FeathersJS' }
});
```

jQuery

Pass the instance of jQuery (`$`) to `restClient.jquery`:

```
app.configure(restClient.jquery(window.jQuery));
```

Or with a module loader:

```
import $ from 'jquery';

app.configure(restClient.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 request = require('request');
const requestClient = request.defaults({
  'auth': {
    'user': 'username',
    'pass': 'password',
    'sendImmediately': false
  }
});

app.configure(restClient.request(requestClient));
```

Superagent

Superagent currently works with a default configuration:

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

app.configure(restClient.superagent(superagent));
```

Axios

Axios currently works with a default configuration:

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

app.configure(restClient.axios(axios));
```

Fetch

Fetch also uses a default configuration:

```
// In Node
const fetch = require('node-fetch');

app.configure(restClient.fetch(fetch));

// In modern browsers
app.configure(restClient.fetch(window.fetch));
```

HTTP API

You can communicate with a Feathers REST API using any other 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 `Accept` header to `application/json`.

Authentication

Authenticating HTTP (REST) requests is a two step process. Frist you have to obtain a JWT from the authentication service by POSTing the strategy you want to use:

```
// POST /authentication the Content-Type header set to application/json
{
  "strategy": "local",
  "email": "your email",
  "password": "your password"
}
```

Here is what that looks like with curl:

```
curl -H "Content-Type: application/json" -X POST -d '{"strategy":"local","email":"your email","password":"your
password"}' http://localhost:3030/authentication
```

Then to authenticate subsequent requests, add the returned `accessToken` to the `Authorization` header:

```
curl -H "Content-Type: application/json" -H "Authorization: <your access token>" -X POST http://localhost:3030/
authentication
```

Also see the JWT and local authentication chapter.

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.

If you want to use any of the built-in find operands ($le, $lt, $ne, $eq, $in, etc.) the general format is as follows:

```
GET /messages?field[$operand]=value&field[$operand]=value2
```

For example, to find the records where field *status* is not equal to **active** you could do

```
GET /messages?status[$ne]=active
```

More information about the possible parameters for official database adapters can be found in the database querying section.

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.

Socket.io Client

> **Note:** We recommend using Feathers and the `@feathersjs/socketio-client` module on the client if possible. To use a direct Socket.io connection without using Feathers on the client however see the Direct connection section.

@feathersjs/socketio-client


```
$ npm install @feathersjs/socketio-client --save
```

The `@feathersjs/socketio-client` module allows to connect to services exposed through the Socket.io server via a Socket.io socket.

> **Important:** Socket.io is also used to *call* service methods. Using sockets for both, calling methods and receiving real-time events is generally faster than using REST and there is no need to use both, REST and Socket.io in the same client application at the same time.

socketio(socket)

Initialize the Socket.io client using a given socket and the default options.

Modular
@feathersjs/client

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio-client');
const io = require('socket.io-client');

const socket = io('http://api.feathersjs.com');
const app = feathers();

// Set up Socket.io client with the socket
app.configure(socketio(socket));

// Receive real-time events through Socket.io
app.service('messages')
  .on('created', message => console.log('New message created', message));

// Call the `messages` service
app.service('messages').create({
  text: 'A message from a REST client'
});
```

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
<script src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
<script src="//unpkg.com/socket.io-client@1.7.3/dist/socket.io.js"></script>
<script>
  // Socket.io is exposed as the `io` global.
  var socket = io('http://api.feathersjs.com');
  // @feathersjs/client is exposed as the `feathers` global.
  var app = feathers();

  // Set up Socket.io client with the socket
  app.configure(feathers.socketio(socket));
```

```
  // Receive real-time events through Socket.io
  app.service('messages')
    .on('created', message => console.log('New message created', message));

  // Call the `messages` service
  app.service('messages').create({
    text: 'A message from a REST client'
  });

  // feathers.errors is an object with all of the custom error types.
</script>
```

socketio(socket, options)

Initialize the Socket.io client using a given socket and the given options.

Options can be:

- `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.

```
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio-client');
const io = require('socket.io-client');

const socket = io('http://api.feathersjs.com');
const app = feathers();

// Set up Socket.io client with the socket
// And a timeout of 2 seconds
app.configure(socketio(socket, {
  timeout: 2000
}));
```

To set a service specific timeout you can use:

```
app.service('messages').timeout = 3000;
```

Direct connection

Feathers sets up a normal Socket.io server that you can connect to with any Socket.io compatible client, usually 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 an inherent cross-origin restriction in the browser so it is possible to connect to any Feathers server.

> **ProTip**: The socket connection URL has to point to the server root which is where Feathers will set up Socket.io.

```
<!-- Connecting to the same URL -->
<script src="/socket.io/socket.io.js">
<script>
  var socket = io();
</script>

<!-- Connecting to a different server -->
<script src="http://localhost:3030/socket.io/socket.io.js">
<script>
  var socket = io('http://localhost:3030/');
</script>
```

Service methods can be called by emitting a `<methodname>` event followed by the service path and method parameters. The service path 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.

If the service path or method does not exist an appropriate Feathers error will be returned.

Authentication

Sockets can be authenticated by sending the `authenticate` event with the `strategy` and the payload. For specific examples see the "Direct Connection" section in the local and jwt authentication chapters.

```
const io = require('socket.io-client');
const socket = io('http://localhost:3030');

socket.emit('authenticate', {
  strategy: 'strategyname',
  ... otherData
}, function(message, data) {
  console.log(message); // message will be null
  console.log(data); // data will be {"accessToken": "your token"}
  // You can now send authenticated messages to the server
});
```

find

Retrieves a list of all matching resources from the service

```
socket.emit('find', 'messages', { status: 'read', user: 10 }, (error, data) => {
  console.log('Found all messages', data);
});
```

Will call `app.service('messages').find({ query: { status: 'read', user: 10 } })` on the server.

get

Retrieve a single resource from the service.

```
socket.emit('get', 'messages', 1, (error, message) => {
  console.log('Found message', message);
});
```

Will call `app.service('messages').get(1, {})` on the server.

```
socket.emit('get', 'messages', 1, { fetch: 'all' }, (error, message) => {
  console.log('Found message', message);
});
```

Will call `app.service('messages').get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
socket.emit('create', 'messages', {
  text: 'I really have to iron'
}, (error, message) => {
  console.log('Todo created', message);
});
```

Will call `app.service('messages').create({ text: 'I really have to iron' }, {})` on the server.

```
socket.emit('create', 'messages', [
  { text: 'I really have to iron' },
  { text: 'Do laundry' }
]);
```

Will call `app.service('messages').create` with the array.

update

Completely replace a single or multiple resources.

```
socket.emit('update', 'messages', 2, {
  text: 'I really have to do laundry'
}, (error, message) => {
  console.log('Todo updated', message);
});
```

Will call `app.service('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('update', 'messages', null, {
  complete: true
}, { complete: false });
```

Will call `app.service('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('patch', 'messages', 2, {
  read: true
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `app.service('messages').patch(2, { read: true }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
socket.emit('patch', 'messages', null, {
  complete: true
}, {
  complete: false
}, (error, message) => {
```

```
  console.log('Patched message', message);
});
```

Will call `app.service('messages').patch(null, { complete: true }, { query: { complete: false } })` on the server to change the status for all read app.service('messages').

This is supported out of the box by the Feathers database adapters

remove

Remove a single or multiple resources:

```
socket.emit('remove', 'messages', 2, { cascade: true }, (error, message) => {
  console.log('Removed a message', message);
});
```

Will call `app.service('messages').remove(2, { query: { cascade: true } })` on the server. The `id` can also be `null` to remove multiple resources:

```
socket.emit('remove', 'messages', null, { read: true });
```

Will call `app.service('messages').remove(null, { query: { read: 'true' } })` on the server to delete all read app.service('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.

```
var socket = io('http://localhost:3030/');

socket.on('messages created', function(message) {
  console.log('Got a new Todo!', message);
});
```

updated, patched

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

```
var socket = io('http://localhost:3030/');

socket.on('my/messages updated', function(message) {
  console.log('Got an updated Todo!', message);
});

socket.emit('update', 'my/messages', 1, {
  text: 'Updated text'
}, {}, function(error, callback) {
 // Do something here
});
```

removed

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

```
var socket = io('http://localhost:3030/');

socket.on('messages removed', function(message) {
  // Remove element showing the Todo from the page
  $('#message-' + message.id).remove();
});
```

Primus Client

> **Note:** We recommend using Feathers and the `@feathersjs/primus-client` module on the client if possible. To use a direct Primus connection without using Feathers on the client however see the Direct connection section.

Loading the Primus client library

In the browser the Primus client library (usually at `primus/primus.js`) always has to be loaded using a `<script>` tag:

```
<script type="text/javascript" src="primus/primus.js"></script>
```

> **Important:** This will make the `Primus` object globally available. Module loader options are currently not available.

Client use in NodeJS

In NodeJS a Primus client can be initialized as follows:

```
const Primus = require('primus');
const Emitter = require('primus-emitter');
const Socket = Primus.createSocket({
  transformer: 'websockets',
  plugin: {
    'emitter': Emitter
  }
});
const socket = new Socket('http://api.feathersjs.com');
```

@feathersjs/primus-client


```
$ npm install @feathersjs/primus-client --save
```

The `@feathersjs/primus-client` module allows to connect to services exposed through the Primus server via the configured websocket library.

> **Important:** Primus sockets are also used to *call* service methods. Using sockets for both, calling methods and receiving real-time events is generally faster than using REST and there is no need to use both, REST and websockets in the same client application at the same time.

`primus(socket)`

Initialize the Primus client using a given socket and the default options.

Modular
@feathersjs/client

```
const feathers = require('@feathersjs/feathers');
const primus = require('@feathersjs/primus-client');
```

```
const socket = new Primus('http://api.my-feathers-server.com');

const app = feathers();

app.configure(primus(socket));

// Receive real-time events through Primus
app.service('messages')
  .on('created', message => console.log('New message created', message));

// Call the `messages` service
app.service('messages').create({
  text: 'A message from a REST client'
});
```

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/core-js/2.1.4/core.min.js"></script>
<script src="//unpkg.com/@feathersjs/client@^3.0.0/dist/feathers.js"></script>
<script type="text/javascript" src="primus/primus.js"></script>
<script>
  // Socket.io is exposed as the `io` global.
  var socket = new Primus('http://api.my-feathers-server.com');
  // @feathersjs/client is exposed as the `feathers` global.
  var app = feathers();

  app.configure(feathers.primus(socket));

  // Receive real-time events through Primus
  app.service('messages')
    .on('created', message => console.log('New message created', message));

  // Call the `messages` service
  app.service('messages').create({
    text: 'A message from a REST client'
  });
</script>
```

`primus(socket, options)`

Initialize the Primus client using a given socket and the given options.

Options can be:

- `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.

```
const feathers = require('@feathersjs/feathers');
const primus = require('@feathersjs/primus-client');
const socket = new Primus('http://api.my-feathers-server.com');

const app = feathers();

app.configure(primus(socket, { timeout: 2000 }));
```

The timeout per service can be changed like this:

```
app.service('messages').timeout = 3000;
```

Direct 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.

```
<script src="primus/primus.js">
<script>
  var socket = new Primus('http://api.my-feathers-server.com');
</script>
```

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.

Authentication

Sockets can be authenticated by sending the `authenticate` event with the `strategy` and the payload. For specific examples see the "Direct Connection" section in the local and jwt authentication chapters.

```
socket.send('authenticate', {
  strategy: 'strategyname',
  ... otherData
}, function(message, data) {
  console.log(message); // message will be null
  console.log(data); // data will be {"accessToken": "your token"}
  // You can now send authenticated messages to the server
});
```

`find`

Retrieves a list of all matching resources from the service

```
primus.send('find', 'messages', { status: 'read', user: 10 }, (error, data) => {
  console.log('Found all messages', data);
});
```

Will call `app.service('messages').find({ query: { status: 'read', user: 10 } })` on the server.

get

Retrieve a single resource from the service.

```
primus.send('get', 'messages', 1, (error, message) => {
  console.log('Found message', message);
});
```

Will call `app.service('messages').get(1, {})` on the server.

```
primus.send('get', 'messages', 1, { fetch: 'all' }, (error, message) => {
```

```
  console.log('Found message', message);
});
```

Will call `app.service('messages').get(1, { query: { fetch: 'all' } })` on the server.

create

Create a new resource with `data` which may also be an array.

```
primus.send('create', 'messages', {
  text: 'I really have to iron'
}, (error, message) => {
  console.log('Message created', message);
});
```

Will call `app.service('messages').create({ "text": "I really have to iron" }, {})` on the server.

```
primus.send('create', 'messages', [
  { text: 'I really have to iron' },
  { text: 'Do laundry' }
]);
```

Will call `app.service('messages').create` on the server with the array.

update

Completely replace a single or multiple resources.

```
primus.send('update', 'messages', 2, {
  text: 'I really have to do laundry'
}, (error, message) => {
  console.log('Message updated', message);
});
```

Will call `app.service('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('update', 'messages', null, {
  complete: true
}, { complete: false });
```

Will call `app.service('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('patch', 'messages', 2, {
  read: true
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `app.service('messages').patch(2, { "read": true }, {})` on the server. The `id` can also be `null` to update multiple resources:

```
primus.send('patch', 'messages', null, {
  complete: true
}, {
  complete: false
}, (error, message) => {
  console.log('Patched message', message);
});
```

Will call `app.service('messages').patch(null, { complete: true }, { query: { complete: false } })` on the server to change the status for all read app.service('messages').

This is supported out of the box by the Feathers database adapters

remove

Remove a single or multiple resources:

```
primus.send('remove', 'messages', 2, { cascade: true }, (error, message) => {
  console.log('Removed a message', message);
});
```

Will call `app.service('messages').remove(2, { query: { cascade: true } })` on the server. The `id` can also be `null` to remove multiple resources:

```
primus.send('remove', 'messages', null, { read: true });
```

Will call `app.service('messages').remove(null, { query: { read: 'true' } })` on the server to delete all read app.service('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.

```
primus.on('messages created', function(message) {
  console.log('Got a new Message!', message);
});
```

updated, patched

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

```
primus.on('my/messages updated', function(message) {
  console.log('Got an updated Message!', message);
});

primus.send('update', 'my/messages', 1, {
```

```
  text: 'Updated text'
}, {}, function(error, callback) {
 // Do something here
});
```

removed

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

```
primus.on('messages removed', function(message) {
  // Remove element showing the Message from the page
  $('#message-' + message.id).remove();
});
```

Authentication


```
$ npm install @feathersjs/authentication --save
```

The @feathersjs/authentication module assists in using JWT for authentication. It has three primary purposes:

1. Setup an `/authentication` endpoint to create JSON Web Tokens (JWT). JWT are used as access tokens. You can learn more about JWT at jwt.io
2. Provide a consistent authentication API for all of the Feathers transports
3. Provide a framework for authentication plugins that use Passport strategies to protect endpoints.

> **Note:** If you are using a 0.x version of `feathers-authentication` please refer to the migration guide. The hooks that were once bundled with this module are now located at feathers-authentication-hooks.

Complementary Plugins

The following plugins are complementary, but entirely optional:

- Using the authencation server on the client: @feathersjs/authentication-client
- Local (username/password) authentication: @feathersjs/authentication-local
- JWT authentication: @feathersjs/authentication-jwt
- OAuth1 authentication: @feathersjs/authentication-oauth1
- OAuth2 authentication: @feathersjs/authentication-oauth2

app.configure(auth(options))

Configure the authentication plugin with the given options. For options that are not provided the default options will be used.

```
const auth = require('@feathersjs/authentication');

// Available options are listed in the "Default Options" section
app.configure(authentication(options))
```

> **Important:** The plugin has to be configured **before** any other service.

Options

The following default options will be mixed in with your global `auth` object from your config file. It will set the mixed options back on to the app so that they are available at any time by calling `app.get('authentication')`. They can all be overridden and are required by some of the authentication plugins.

```
{
 path: '/authentication', // the authentication service path
 header: 'Authorization', // the header to use when using JWT auth
 entity: 'user', // the entity that will be added to the request, socket, and context.params. (ie. req.user, so
cket.user, context.params.user)
 service: 'users', // the service to look up the entity
```

```
  passReqToCallback: true, // whether the request object should be passed to the strategies `verify` function
  session: false, // whether to use sessions
  cookie: {
   enabled: false, // whether cookie creation is enabled
   name: 'feathers-jwt', // the cookie name
   httpOnly: false, // when enabled, prevents the client from reading the cookie.
   secure: true // whether cookies should only be available over HTTPS
  },
  jwt: {
   header: { typ: 'access' }, // by default is an access token but can be any type
   audience: 'https://yourdomain.com', // The resource server where the token is processed
   subject: 'anonymous', // Typically the entity id associated with the JWT
   issuer: 'feathers', // The issuing server, application or resource
   algorithm: 'HS256', // the algorithm to use
   expiresIn: '1d' // the access token expiry
  }
 }
```

app.service('authentication')

The heart of this plugin is a service for creating JWT. It's a normal Feathers service that implements only the `create` and `remove` methods. The `/authentication` service provides all of the functionality that the `/auth/local` and `/auth/token` endpoints did. To choose a strategy, the client must pass the `strategy` name in the request body. This will be different based on the plugin used. See the documentation for the plugins listed at the top of this page for more information.

service.create(data)

The `create` method will be used in nearly every Feathers application. It creates a JWT based on the `jwt` options configured on the plugin. The API of this method utilizes the `context` object:

service.remove(data)

The `remove` method will be used less often. It mostly exists to allow for adding hooks the the "logout" process. For example, in services that require high control over security, a developer could register hooks on the `remove` method that perform token blacklisting.

service.hooks({ before })

These properties can be modified to change the behavior of the `/authentication` service:

- `context.data.payload {Object}` - determines the payload of the JWT
- `context.params.payload {Object}` - also determines the payload of the JWT. Any matching attributes in the `context.data.payload` will be overwritten by these. Persists into after hooks.
- `context.params.authenticated {Boolean}` - After successful authentication, will be set to `true`, unless it's set to `false` in a before hook. If you set it to `false` in a before hook, it will prevent the websocket from being flagged as authenticated. Persists into after hooks.

service.hooks({ after })

- `context.params[entity] {Object}` - After successful authentication, the `entity` looked up from the database will be populated here. (The default option is `user`.)

app.passport

app.passport.createJWT(payload, options)

`app.passport.createJWT(payload, options) -> Promise` is used by the authentication service to generate JSON Web Tokens.

- `payload {Object}` - becomes the JWT payload. Will also include an `exp` property denoting the expiry timestamp.
- `options {Object}` - the options passed to jsonwebtoken `sign()`
  - `secret {String | Buffer}` - either the secret for HMAC algorithms, or the PEM encoded private key for RSA and ECDSA.
  - `jwt` - See the `jsonwebtoken` package docs for other available options. The authenticate method uses the default `jwt` options. When using this package, directly, they will have to be passed in manually.

The returned `promise` resolves with the JWT or fails with an error.

app.passport.verifyJWT(token, options)

Verifies the signature and payload of the passed in JWT `token` using the `options`.

- `token {JWT}` - the JWT to be verified.
- `options {Object}` the options passed to jsonwebtoken `verify()`
  - `secret {String | Buffer}` - - either the secret for HMAC algorithms, or the PEM encoded private key for RSA and ECDSA.
  - See the `jsonwebtoken` package docs for other available options.

The returned `promise` resolves with the payload or fails with an error.

auth.hooks.authenticate(strategies)

`@feathersjs/authentication` only includes a single hook. This bundled `authenticate` hook is used to register an array of authentication strategies on a service method.

> **Note:** This should usually be used on your `/authentication` service. Without it you can hit the `authentication` service and generate a JWT `accessToken` without authentication (ie. anonymous authentication).

```
app.service('authentication').hooks({
 before: {
  create: [
   // You can chain multiple strategies
   auth.hooks.authenticate(['jwt', 'local']),
  ],
  remove: [
   auth.hooks.authenticate('jwt')
  ]
 }
});
```

Authentication Events

The `login` and `logout` events are emitted on the `app` object whenever a client successfully authenticates or "logs out". (With JWT, logging out doesn't invalidate the JWT. Read the section about how JWT work for more information.) These events are only emitted on the server.

app.on('login', callback))

app.on('logout', callback))

These two events use a `callback` function with the same signature.

- `result` {Object} - The final `context.result` from the `authentication` service. Unless you customize the `context.response` in an after hook, this will only contain the `accessToken`, which is the JWT.
- `meta` {Object} - information about the request. *The `meta` data varies per transport / provider as follows.*
- Using `@feathersjs/express/rest`
  - `provider` {String} - will always be `"rest"`
  - `req` {Object} - the Express request object.
  - `res` {Object} - the Express response object.
- Using `feathers-socketio` and `feathers-primus`:
  - `provider` {String} - the transport name: `socketio` or `primus`
  - `connection` {Object} - the same as `params` in the hook context
  - `socket` {SocketObject} - the current user's WebSocket object. It also contains the `feathers` attribute, which is the same as `params` in the hook context.

Express Middleware

There is an `authenticate` middleware. It is used the exact same way you would the regular Passport express middleware:

```
app.post('/login', auth.express.authenticate('local', { successRedirect: '/app', failureRedirect: '/login' }));
```

Additional middleware are included and exposed but typically you don't need to worry about them:

- `emitEvents` - emit `login` and `logout` events
- `exposeCookies` - expose cookies to Feathers so they are available to hooks and services. **This is NOT used by default as its use exposes your API to CSRF vulnerabilities.** Only use it if you really know what you're doing.
- `exposeHeaders` - expose headers to Feathers so they are available to hooks and services. **This is NOT used by default as its use exposes your API to CSRF vulnerabilities.** Only use it if you really know what you're doing.
- `failureRedirect` - support redirecting on auth failure. Only triggered if `hook.redirect` is set.
- `successRedirect` - support redirecting on auth success. Only triggered if `hook.redirect` is set.
- `setCookie` - support setting the JWT access token in a cookie. Only enabled if cookies are enabled. **Note: Feathers will NOT read an access token from a cookie. This would expose the API to CSRF attacks.** This `setCookie` feature is available primarily for helping with Server Side Rendering.

Complete Example

Here's an example of a Feathers server that uses `@feathersjs/authentication` for local authentication.

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');
const socketio = require('@feathersjs/socketio');
const auth = require('@feathersjs/authentication');
const local = require('@feathersjs/authentication-local');
const jwt = require('@feathersjs/authentication-jwt');
const memory = require('feathers-memory');

const app = express(feathers());
app.configure(express.rest())
 .configure(socketio())
 .use(express.json())
 .use(express.urlencoded({ extended: true }))
```

```
  .configure(auth({ secret: 'supersecret' }))
  .configure(local())
  .configure(jwt())
  .use('/users', memory())
  .use('/', feathers.static(__dirname + '/public'))
  .use(express.errorHandler());

app.service('users').hooks({
  // Make sure `password` never gets sent to the client
  after: local.hooks.protect('password')
});

app.service('authentication').hooks({
 before: {
  create: [
   // You can chain multiple strategies
   auth.hooks.authenticate(['jwt', 'local'])
  ],
  remove: [
   auth.hooks.authenticate('jwt')
  ]
 }
});

// Add a hook to the user service that automatically replaces
// the password with a hash of the password before saving it.
app.service('users').hooks({
 before: {
  find: [
   auth.hooks.authenticate('jwt')
  ],
  create: [
   local.hooks.hashPassword({ passwordField: 'password' })
  ]
 }
});

const port = 3030;
let server = app.listen(port);
server.on('listening', function() {
 console.log(`Feathers application started on localhost:${port}`);
});
```

Authentication Client


```
npm install @feathersjs/authentication-client --save
```

The @feathersjs/authentication-client module allows you to easily authenticate against a Feathers server. It is not required, but makes it easier to implement authentication in your client by automatically storing and sending the JWT access token and handling re-authenticating when a websocket disconnects.

This module contains:

- The main entry function
- Additional feathersClient methods
- Some helpful hooks

app.configure(auth(options))

Setup is done the same as all Feathers plugins, using the `configure` method:

```
const feathers = require('@feathersjs/feathers');
const auth = require('@feathersjs/authentication-client');

const app = feathers();

// Available options are listed in the "Options" section
app.configure(auth(options))
```

> The transports plugins must have been initialized previously to the authentication plugin on the client side

Options

The following default options will be mixed in with the settings you pass in when configuring authentication. It will set the mixed options back to to the app so that they are available at any time by `app.get('auth')`. They can all be overridden.

```
{
  header: 'Authorization', // the default authorization header for REST
  path: '/authentication', // the server-side authentication service path
  jwtStrategy: 'jwt', // the name of the JWT authentication strategy
  entity: 'user', // the entity you are authenticating (ie. a users)
  service: 'users', // the service to look up the entity
  cookie: 'feathers-jwt', // the name of the cookie to parse the JWT from when cookies are enabled server side
  storageKey: 'feathers-jwt', // the key to store the accessToken in localstorage or AsyncStorage on React Nati
ve
  storage: undefined // Passing a WebStorage-compatible object to enable automatic storage on the client.
}
```

To enable storing the JWT make sure to provide a `storage` when configuring the plugin. The following storage options are available:

- `window.localStorage` in the browser to use the browsers localStorage

- AsyncStorage for *React Native*
- localForage which helps deal with older browsers and browsers in Incognito / Private Browsing mode.
- cookie-storage uses cookies. It can be useful on devices that don't support `localStorage`.

app.authenticate()

`app.authenticate() -> Promise` with no arguments will to authenticate using the JWT from the `storage`. This is normally called to either show your application (when successfull) or showing a login page or redirecting to the appropriate oAuth link.

```
app.authenticate().then(() => {
  // show application page
}).catch(() => {
  // show login page
})
```

> **Important:** `app.authenticate()` **has** to be called when you want to use the token from storage and **only once** when the application initializes. Once successfull, all subsequent reuests will send their authentication information automatically.

app.authenticate(options)

`app.authenticate(options) -> Promise` will try to authenticate with a Feathers server by passing a `strategy` and other properties as credentials. It will use whichever transport has been setup on the client (@feathersjs/rest-client, @feathersjs/socketio-client, or @feathersjs/primus-client).

```
// Authenticate with the local email/password strategy
app.authenticate({
  strategy: 'local',
  email: 'my@email.com',
  password: 'my-password'
}).then(() => {
  // Loged in
}).catch(e => {
  // Show login page (potentially with `e.message`)
  console.error('Authentication error', e);
});

app.authenticate({
  strategy: 'jwt',
  accessToken: '<the.jwt.token.string>'
}).then(() => {
  // JWT authentication successful
}).catch(e => {
  console.error('Authentication error', e);
  // Show login page
});
```

- `data {Object}` - of the format `{strategy [, ...otherProps]}`
  - `strategy {String}` - the name of the strategy to be used to authenticate. Required.
  - `...otherProps {Properties}` vary depending on the chosen strategy. Above is an example of using the `jwt` strategy. Below is one for the `local` strategy.

app.logout()

Removes the JWT accessToken from storage on the client. It also calls the `remove` method of the /authentication service on the Feathers server.

app.on('reauthentication-error', errorHandler)

In the event that your server goes down or the client loses connectivity, it will automatically handle attempting to re-authenticate the socket when the client regains connectivity with the server. In order to handle an authentication failure during automatic re-authentication you need to implement the following event listener:

```
const errorHandler = error => {
  app.authenticate({
    strategy: 'local',
    email: 'admin@feathersjs.com',
    password: 'admin'
  }).then(response => {
    // You are now authenticated again
  });
};

// Handle when auth fails during a reconnect or a transport upgrade
app.on('reauthentication-error', errorHandler)
```

app.passport

`app.passport` contains helper functions to work with the JWT.

app.passport.getJWT()

Pull the JWT from `storage` or the cookie. Returns a Promise.

app.passport.verifyJWT(token)

Verify that a JWT is not expired and decode it to get the payload. Returns a Promise.

app.passport.payloadIsValid(token)

Synchronously verify that a token has not expired. Returns a Boolean.

Hooks

There are 3 hooks. They are really meant for internal use and you shouldn't need to worry about them very often.

- `populateAccessToken` - Takes the token and puts in on `hooks.params.accessToken` in case you need it in one of your client side services or hooks
- `populateHeader` - Add the accessToken to the authorization header
- `populateEntity` - Experimental. Populate an entity based on the JWT payload.

Complete Example

Here's an example of a Feathers server that uses `@feathersjs/authentication-client`.

```
const feathers = require('@feathersjs/feathers');
```

```
const rest = require('@feathersjs/rest-client');
const auth = require('@feathersjs/authentication-client');

const superagent = require('superagent');
const localStorage = require('localstorage-memory');

const feathersClient = feathers();

feathersClient.configure(rest('http://localhost:3030').superagent(superagent))
  .configure(auth({ storage: localStorage }));

feathersClient.authenticate({
  strategy: 'local',
  email: 'admin@feathersjs.com',
  password: 'admin'
})
.then(response => {
  console.log('Authenticated!', response);
  return feathersClient.passport.verifyJWT(response.accessToken);
})
.then(payload => {
  console.log('JWT Payload', payload);
  return feathersClient.service('users').get(payload.userId);
})
.then(user => {
  feathersClient.set('user', user);
  console.log('User', feathersClient.get('user'));
})
.catch(function(error){
  console.error('Error authenticating!', error);
});
```

Local Authentication


```
$ npm install @feathersjs/authentication-local --save
```

@feathersjs/authentication-local is a server side module that wraps the passport-local authentication strategy, which lets you authenticate with your Feathers application using a username and password.

This module contains 3 core pieces:

1. The main initialization function
2. The `hashPassword` hook
3. The `Verifier` class

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('@feathersjs/feathers');
const authentication = require('feathers-authentication');
const local = require('@feathersjs/authentication-local');
const app = feathers();

// Setup authentication
app.configure(authentication(settings));
app.configure(local());

// Setup a hook to only allow valid JWTs or successful
// local auth to authenticate and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['local', 'jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Options

```
{
    name: 'local', // the name to use when invoking the authentication Strategy
    entity: 'user', // the entity that you're comparing username/password against
    service: 'users', // the service to look up the entity
    usernameField: 'email', // key name of username field
    passwordField: 'password', // key name of password field
    passReqToCallback: true, // whether the request object should be passed to `verify`
    session: false // whether to use sessions,
    Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for
 details.
}
```

hooks

hashPassword

This hook is used to hash plain text passwords before they are saved to the database. It uses the bcrypt algorithm by default but can be customized by passing your own `options.hash` function.

Available options are

- `passwordField` (default: `'password'`) - key name of password field to look on context.data
- `hash` (default: bcrypt hash function) - Takes in a password and returns a hash.

```
const local = require('@feathersjs/authentication-local');

app.service('users').hooks({
  before: {
    create: [
      local.hooks.hashPassword()
    ]
  }
});
```

protect

The protect hook makes sure that protected fields don't get sent to a client.

```
const local = require('@feathersjs/authentication-local');

app.service('users').hooks({
  before: {
    create: [
      local.hooks.protect('password')
    ]
  }
});
```

Verifier

This is the verification class that does the actual username and password verification by looking up the entity (normally a `user`) on a given service by the `usernameField` and compares the hashed password using bcrypt. It has the following methods that can all be overridden. All methods return a promise except `verify`, which has the exact same signature as passport-local.

```
{
    constructor(app, options) // the class constructor
    _comparePassword(entity, password) // compares password using bcrypt
    _normalizeResult(result) // normalizes result from service to account for pagination
    verify(req, username, password, done) // queries the service and calls the other internal functions.
}
```

The `Verifier` class can be extended so that you customize it's behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want use this plugin.

An example of customizing the Verifier:

```
import local, { Verifier } from '@feathersjs/authentication-local';
```

```
class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, username, password, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}

app.configure(local({ Verifier: CustomVerifier }));
```

Client Usage

authentication-client

When this module is registered server side, using the default config values this is how you can authenticate using @feathersjs/authentication-client:

```
app.authenticate({
  strategy: 'local',
  email: 'your email',
  password: 'your password'
}).then(response => {
  // You are now authenticated
});
```

HTTP Request

If you are not using the `feathers-authentication-client` and you have registered this module server side, make a `POST` request to `/authentication` with the following payload:

```
// POST /authentication the Content-Type header set to application/json
{
  "strategy": "local",
  "email": "your email",
  "password": "your password"
}
```

Here is what that looks like with curl:

```
curl -H "Content-Type: application/json" -X POST -d '{"strategy":"local","email":"your email","password":"your password"}' http://localhost:3030/authentication
```

Sockets

Authenticating using a local strategy via sockets is done by emitting the following message:

```
const io = require('socket.io-client');
const socket = io('http://localhost:3030');

socket.emit('authenticate', {
  strategy: 'local',
  email: 'your email',
  password: 'your password'
```

```
}, function(message, data) {
  console.log(message); // message will be null
  console.log(data); // data will be {"accessToken": "your token"}
  // You can now send authenticated messages to the server
});
```

JWT Authentication


```
$ npm install @feathersjs/authentication-jwt --save
```

@feathersjs/authentication-jwt is a module for the authentication server that wraps the passport-jwt authentication strategy, which lets you authenticate with your Feathers application using a JSON Web Token access token.

This module contains 3 core pieces:

1. The main initialization function
2. The `Verifier` class
3. The `ExtractJwt` object from passport-jwt.

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('@feathersjs/feathers');
const authentication = require('feathers-authentication');
const jwt = require('@feathersjs/authentication-jwt');
const app = feathers();

// Setup authentication
app.configure(authentication(settings));
app.configure(jwt());

// Setup a hook to only allow valid JWTs to authenticate
// and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Options

```
{
    name: 'jwt', // the name to use when invoking the authentication Strategy
    entity: 'user', // the entity that you pull from if an 'id' is present in the payload
    service: 'users', // the service to look up the entity
    passReqToCallback: true, // whether the request object should be passed to `verify`
    jwtFromRequest: [ // a passport-jwt option determining where to parse the JWT
      ExtractJwt.fromHeader, // From "Authorization" header
      ExtractJwt.fromAuthHeaderWithScheme('Bearer'), // Allowing "Bearer" prefix
      ExtractJwt.fromBodyField('body') // from request body
    ],
    secretOrKey: auth.secret, // Your main secret provided to passport-jwt
    session: false // whether to use sessions,
    Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for
```

```
  details.
}
```

Additional passport-jwt options can be provided.

Verifier

This is the verification class that receives the JWT payload (if verification is successful) and either returns the payload or, if an `id` is present in the payload, populates the entity (normally a `user`) and returns both the entity and the payload. It has the following methods that can all be overridden. The `verify` function has the exact same signature as passport-jwt.

```
{
    constructor(app, options) // the class constructor
    verify(req, payload, done) // queries the configured service
}
```

Customizing the Verifier

The `Verifier` class can be extended so that you customize it's behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want use this plugin.

An example of customizing the Verifier:

```
import jwt, { Verifier } from '@feathersjs/authentication-jwt';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, payload, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}

app.configure(jwt({ Verifier: CustomVerifier }));
```

Client Usage

authentication-client

When this module is registered server side, using the default config values this is how you can authenticate using @feathersjs/authentication-client:

```
app.authenticate({
  strategy: 'jwt',
  accessToken: 'your access token'
}).then(response => {
  // You are now authenticated
});
```

HTTP

If you are not using `@feathersjs/authentication-client` and you have registered this module server side then you can include the access token in an `Authorization` header.

Here is what that looks like with curl:

```
curl -H "Content-Type: application/json" -H "Authorization: <your access token>" -X POST http://localhost:3030/
authentication
```

Sockets

Authenticating using an access token via sockets is done by emitting the following message:

```
const io = require('socket.io-client');
const socket = io('http://localhost:3030');

socket.emit('authenticate', {
  strategy: 'jwt',
  accessToken: 'your token'
}, function(message, data) {
  console.log(message); // message will be null
  console.log(data); // data will be {"accessToken": "your token"}
  // You can now send authenticated messages to the server
});
```

OAuth1 Authentication


```
$ npm install @feathersjs/authentication-oauth1 --save
```

@feathersjs/authentication-oauth1 is a server side module that allows you to use any Passport OAuth1 authentication strategy within your Feathers application, most notably Twitter.

This module contains 2 core pieces:

1. The main initialization function
2. The `Verifier` class

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('@feathersjs/feathers');
const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const oauth1 = require('@feathersjs/authentication-oauth1');
const session = require('express-session');
const TwitterStrategy = require('passport-twitter').Strategy;
const app = feathers();

// Setup in memory session
app.use(session({
  secret: 'super secret',
  resave: true,
  saveUninitialized: true
}));

// Setup authentication
app.configure(authentication(settings));
app.configure(jwt());
app.configure(oauth1({
  name: 'twitter',
  Strategy: TwitterStrategy,
  consumerKey: '<your consumer key>',
  consumerSecret: '<your consumer secret>'
}));

// Setup a hook to only allow valid JWTs to authenticate
// and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Registering the OAuth1 plugin will automatically set up routes to handle the OAuth redirects and authorization.

Options

```
{
    idField: '<provider>Id', // The field to look up the entity by when logging in with the provider. Defaults
to '<provider>Id' (ie. 'twitterId').
    path: '/auth/<provider>', // The route to register the middleware
    callbackURL: 'http(s)://hostame[:port]/auth/<provider>/callback', // The callback url. Will automatically t
ake into account your host and port and whether you are in production based on your app environment to construc
t the url. (ie. in development http://localhost:3030/auth/twitter/callback)
    entity: 'user', // the entity that you are looking up
    service: 'users', // the service to look up the entity
    passReqToCallback: true, // whether the request object should be passed to `verify`
    session: true, // whether to use sessions,
    handler: function, // Express middleware for handling the oauth callback. Defaults to the built in middlewa
re.
    formatter: function, // The response formatter. Defaults the the built in feathers-rest formatter, which re
turns JSON.
    Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for
details.
}
```

Additional passport strategy options can be provided based on the OAuth1 strategy you are configuring.

Verifier

This is the verification class that handles the OAuth1 verification by looking up the entity (normally a `user`) on a given service and either creates or updates the entity and returns them. It has the following methods that can all be overridden. All methods return a promise except `verify`, which has the exact same signature as passport-oauth1.

```
{
    constructor(app, options) // the class constructor
    _updateEntity(entity) // updates an existing entity
    _createEntity(entity) // creates an entity if they didn't exist already
    _normalizeResult(result) // normalizes result from service to account for pagination
    verify(req, accessToken, refreshToken, profile, done) // queries the service and calls the other internal f
unctions.
}
```

The `Verifier` class can be extended so that you customize it's behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want use this plugin.

An example of customizing the Verifier:

```
import oauth1, { Verifier } from '@feathersjs/authentication-oauth1';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, accessToken, refreshToken, profile, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}

app.configure(oauth1({
```

```
  name: 'twitter'
  Strategy: TwitterStrategy,
  consumerKey: '<your consumer key>',
  consumerSecret: '<your consumer secret>',
  Verifier: CustomVerifier
}));
```

Customizing The OAuth Response

Whenever you authenticate with an OAuth1 provider such as Twitter, the provider sends back an `accessToken`, `refreshToken`, and a `profile` that contains the authenticated entity's information based on the OAuth1 `scopes` you have requested and been granted.

By default the `Verifier` takes everything returned by the provider and attaches it to the `entity` (ie. the user object) under the provider name. You will likely want to customize the data that is returned. This can be done by adding a `before` hook to both the `update` and `create` service methods on your `entity`'s service.

```
app.configure(oauth1({
  name: 'twitter',
  entity: 'user',
  service: 'users',
  Strategy,
  consumerKey: '<your consumer key>',
  consumerSecret: '<your consumer secret>'
}));

function customizeTwitterProfile() {
  return function(context) {
    console.log('Customizing Twitter Profile');
    // If there is a twitter field they signed up or
    // signed in with twitter so let's pull the email. If
    if (context.data.twitter) {
      context.data.email = context.data.twitter.email;
    }

    // If you want to do something whenever any OAuth
    // provider authentication occurs you can do this.
    if (context.params.oauth) {
      // do something for all OAuth providers
    }

    if (context.params.oauth.provider === 'twitter') {
      // do something specific to the twitter provider
    }

    return Promise.resolve(context);
  };
}


app.service('users').hooks({
  before: {
    create: [customizeTwitterProfile()],
    update: [customizeTwitterProfile()]
  }
});
```

Client Usage

When this module is registered server side, whether you are using `feathers-authentication-client` or not the user has to navigate to the authentication strategy url. This could be by setting `window.location` or through a link in your app.

For example you might have a login button for Twitter:

```
<a href="/auth/twitter" class="button">Login With Twitter</a>
```

OAuth2 Authentication


```
$ npm install @feathersjs/authentication-oauth2 --save
```

@feathersjs/authentication-oauth2 is a server side module that allows you to use any Passport OAuth2 authentication strategy within your Feathers application. There are hundreds of them! Some commonly used ones are:

- Facebook
- Instagram
- Github
- Google
- Spotify

This module contains 2 core pieces:

1. The main initialization function
2. The `Verifier` class

Configuration

In most cases initializing the module is as simple as doing this:

```
const feathers = require('@feathersjs/feathers');
const authentication = require('feathers-authentication');
const jwt = require('feathers-authentication-jwt');
const oauth2 = require('@feathersjs/authentication-oauth2');
const FacebookStrategy = require('passport-facebook').Strategy;
const app = feathers();

// Setup authentication
app.configure(authentication(settings));
app.configure(jwt());
app.configure(oauth2({
  name: 'facebook',
  Strategy: FacebookStrategy,
  clientID: '<your client id>',
  clientSecret: '<your client secret>',
  scope: ['public_profile', 'email']
}));

// Setup a hook to only allow valid JWTs to authenticate
// and get new JWT access tokens
app.service('authentication').hooks({
  before: {
    create: [
      authentication.hooks.authenticate(['jwt'])
    ]
  }
});
```

This will pull from your global authentication object in your config file. It will also mix in the following defaults, which can be customized.

Registering the OAuth2 plugin will automatically set up routes to handle the OAuth redirects and authorization.

Options

```
{
    idField: '<provider>Id', // The field to look up the entity by when logging in with the provider. Defaults
to '<provider>Id' (ie. 'facebookId').
    path: '/auth/<provider>', // The route to register the middleware
    callbackURL: 'http(s)://hostname[:port]/auth/<provider>/callback', // The callback url. Will automatically
take into account your host and port and whether you are in production based on your app environment to constru
ct the url. (ie. in development http://localhost:3030/auth/facebook/callback)
    successRedirect: undefined,
    failureRedirect: undefined,
    entity: 'user', // the entity that you are looking up
    service: 'users', // the service to look up the entity
    passReqToCallback: true, // whether the request object should be passed to `verify`
    session: false, // whether to use sessions,
    handler: function, // Express middleware for handling the oauth callback. Defaults to the built in middlewa
re.
    formatter: function, // The response formatter. Defaults the the built in feathers-rest formatter, which re
turns JSON.
    Verifier: Verifier // A Verifier class. Defaults to the built-in one but can be a custom one. See below for
details.
}
```

Additional passport strategy options can be provided based on the OAuth1 strategy you are configuring.

Verifier

This is the verification class that handles the OAuth2 verification by looking up the entity (normally a `user`) on a given service and either creates or updates the entity and returns them. It has the following methods that can all be overridden. All methods return a promise except `verify`, which has the exact same signature as passport-oauth2.

```
{
    constructor(app, options) // the class constructor
    _updateEntity(entity) // updates an existing entity
    _createEntity(entity) // creates an entity if they didn't exist already
    _normalizeResult(result) // normalizes result from service to account for pagination
    verify(req, accessToken, refreshToken, profile, done) // queries the service and calls the other internal f
unctions.
}
```

The `Verifier` class can be extended so that you customize it's behavior without having to rewrite and test a totally custom local Passport implementation. Although that is always an option if you don't want use this plugin.

An example of customizing the Verifier:

```
import oauth2, { Verifier } from '@feathersjs/authentication-oauth2';

class CustomVerifier extends Verifier {
  // The verify function has the exact same inputs and
  // return values as a vanilla passport strategy
  verify(req, accessToken, refreshToken, profile, done) {
    // do your custom stuff. You can call internal Verifier methods
    // and reference this.app and this.options. This method must be implemented.

    // the 'user' variable can be any truthy value
    // the 'payload' is the payload for the JWT access token that is generated after successful authentication
    done(null, user, payload);
  }
}
```

```
app.configure(oauth2({
  name: 'facebook',
  Strategy: FacebookStrategy,
  clientID: '<your client id>',
  clientSecret: '<your client secret>',
  scope: ['public_profile', 'email'],
  Verifier: CustomVerifier
}));
```

Customizing The OAuth Response

Whenever you authenticate with an OAuth2 provider such as Facebook, the provider sends back an `accessToken`, `refreshToken`, and a `profile` that contains the authenticated entity's information based on the OAuth2 `scopes` you have requested and been granted.

By default the `Verifier` takes everything returned by the provider and attaches it to the `entity` (ie. the user object) under the provider name. You will likely want to customize the data that is returned. This can be done by adding a `before` hook to both the `update` and `create` service methods on your `entity`'s service.

```
app.configure(oauth2({
  name: 'github',
  entity: 'user',
  service: 'users',
  Strategy,
  clientID: 'your client id',
  clientSecret: 'your client secret'
}));

function customizeGithubProfile() {
  return function(context) {
    console.log('Customizing Github Profile');
    // If there is a github field they signed up or
    // signed in with github so let's pull the primary account email.
    if (context.data.github) {
      context.data.email = context.data.github.profile.emails.find(email => email.primary).value;
    }

    // If you want to do something whenever any OAuth
    // provider authentication occurs you can do this.
    if (context.params.oauth) {
      // do something for all OAuth providers
    }

    if (context.params.oauth.provider === 'github') {
      // do something specific to the github provider
    }

    return Promise.resolve(context);
  };
}


app.service('users').hooks({
  before: {
    create: [customizeGithubProfile()],
    update: [customizeGithubProfile()]
  }
});
```

Client Usage

When this module is registered server side, whether you are using `feathers-authentication-client` or not the user has to navigate to the authentication strategy url. This could be by setting `window.location` or through a link in your app.

For example you might have a login button for Facebook:

```
<a href="/auth/facebook" class="button">Login With Facebook</a>
```

Database adapters

Feathers database adapters are modules that provide services that implement standard CRUD functionality for a specific database using a common API for initialization and settings and providing a common query syntax.

> **Important:** Services allow to implement access to *any* database, the database adapters listed here are just convenience wrappers with a common API. You can still get Feathers functionality for databases that are not listed here. Also have a look at the list of community database adapters

The following databases are supported:

Database	Adapter
In memory	feathers-memory, feathers-nedb
Localstorage, AsyncStorage	feathers-localstorage
Filesystem	feathers-nedb
MongoDB	feathers-mongodb, feathers-mongoose
MySQL, PostgreSQL, MariaDB, SQLite, MSSQL	feathers-knex, feathers-sequelize
Elasticsearch	feathers-elasticsearch
RethinkDB	feathers-rethinkdb

Memory/Filesystem

- feathers-memory - An in-memory database adapter
- feathers-localstorage - An adapter for Client side Feathers that can use the browsers LocalStorage or ReactNative's AsyncStorage.
- feathers-nedb - A database adapter for NeDB an in-memory or file system based standalone database.

SQL

- feathers-knex - An adapter for KnexJS, an SQL query builder for NodeJS supporting PostgreSQL, MySQL, SQLite and MSSQL
- feathers-sequelize - An adapter for Sequelize an ORM for NodeJS supporting PostgreSQL, MySQL, SQLite and MSSQL

NoSQL

- feathers-mongoose - A database adapter for Mongoose an Object Modelling library for NodeJS and MongoDB
- feathers-mongodb - A database adapter for MongoDB using the official NodeJS database driver
- feathers-elasticsearch - A database adapter for Elasticsearch
- feathers-rethinkdb - A database adapter for RethinkDB a real-time database.

Common API

All database adapters implement a common interface for initialization, pagination, extending and querying. This chapter describes the common adapter initialization and options, how to enable and use pagination, the details on how specific service methods behave and how to extend an adapter with custom functionality.

> **Important:** Every database adapter is an implementation of the Feathers service interface. We recommend being familiar with services, service events and hooks before using a database adapter.

`service([options])`

Returns a new service instance initialized with the given options.

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

app.use('/messages', service());
app.use('/messages', service({ id, events, paginate }));
```

Options:

- `id` (*optional*) - The name of the id field property (usually set by default to `id` or `_id`).
- `events` (*optional*) - A list of custom service events sent by this service
- `paginate` (*optional*) - A pagination object containing a `default` and `max` page size

Pagination

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

- `default` - Sets the default number of items when `$limit` is not set
- `max` - Sets the maximum allowed number of items per page (even if the `$limit` query parameter is set higher)

When `paginate.default` is set, `find` will return an *page 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` for this call
app.service('todos').find({
  paginate: {
```

```
    default: 100,
    max: 200
  }
});

// disable pagination for this call
app.service('todos').find({
  paginate: false
});
```

> **Note:** Disabling or changing the default pagination is not available in the client. Only `params.query` is passed to the server (also see a workaround here)
>
> **Pro tip:** To just get the number of available records set `$limit` to `0`. This will only run a (fast) counting query against the database.

Service methods

This section describes specifics on how the service methods are implemented for all adapters.

adapter.find(params)

`adapter.find(params) -> Promise` returns a list of all records matching the query in `params.query` using the common querying mechanism. Will either return an array with the results or a page object if pagination is enabled.

> **Important**: When used via REST URLs all query values are strings. Depending on the database the values in `params.query` might have to be converted to the right type in a before hook.

```
// Find all messages for user with id 1
app.service('messages').find({
  query: {
    userId: 1
  }
}).then(messages => console.log(messages));

// Find all messages belonging to room 1 or 3
app.service('messages').find({
  query: {
    roomId: {
      $in: [ 1, 3 ]
    }
  }
}).then(messages => console.log(messages));
```

Find all messages for user with id 1

```
GET /messages?userId=1
```

Find all messages belonging to room 1 or 3

```
GET /messages?roomId[$in]=1&roomId[$in]=3
```

adapter.get(id, params)

`adapter.get(id, params) -> Promise` retrieves a single record by its unique identifier (the field set in the `id` option during initialization).

```
app.service('messages').get(1)
  .then(message => console.log(message));
```

```
GET /messages/1
```

adapter.create(data, params)

`adapter.create(data, params) -> Promise` creates a new record with `data`. `data` can also be an array to create multiple records.

```
app.service('messages').create({
    text: 'A test message'
  })
  .then(message => console.log(message));

app.service('messages').create([{
    text: 'Hi'
  }, {
    text: 'How are you'
  }])
  .then(messages => console.log(messages));
```

```
POST /messages
{
  "text": "A test message"
}
```

adapter.update(id, data, params)

`adapter.update(id, data, params) -> Promise` completely replaces a single record identified by `id` with `data`. Does not allow replacing multiple records (`id` can't be `null`). `id` can not be changed.

```
app.service('messages').update(1, {
    text: 'Updates message'
  })
  .then(message => console.log(message));
```

```
PUT /messages/1
{ "text": "Updated message" }
```

adapter.patch(id, data, params)

`adapter.patch(id, data, params) -> Promise` merges a record identified by `id` with `data`. `id` can be `null` to allow replacing multiple records (all records that match `params.query` the same as in `.find`). `id` can not be changed.

```
app.service('messages').patch(1, {
  text: 'A patched message'
}).then(message => console.log(message));

const params = {
  query: { read: false }
};

// Mark all unread messages as read
app.service('messages').patch(null, {
```

```
  read: true
}, params);
```

```
PATCH /messages/1
{ "text": "A patched message" }
```

Mark all unread messages as read

```
PATCH /messages?read=false
{ "read": true }
```

adapter.remove(id, params)

`adapter.remove(id, params) -> Promise` removes a record identified by `id`. `id` can be `null` to allow removing multiple records (all records that match `params.query` the same as in `.find`).

```
app.service('messages').remove(1)
  .then(message => console.log(message));

const params = {
  query: { read: true }
};

// Remove all read messages
app.service('messages').remove(null, params);
```

```
DELETE /messages/1
```

Remove all read messages

```
DELETE /messages?read=true
```

Extending Adapters

There are two ways to extend existing database adapters. Either by extending the ES6 base class or by adding functionality through hooks.

> **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 example, `createdAt` and `updatedAt` timestamps could be added like this:

```
const feathers = require('@feathersjs/feathers');

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

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

```
app.service('todos').hooks({
  before: {
    create: [
      (context) => context.data.createdAt = new Date()
    ],

    update: [
      (context) => context.data.updatedAt = new Date()
    ]
  }
});

app.listen(3030);
```

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>');

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
  }
}));
```

Querying

All official database adapters support a common way for querying, sorting, limiting and selecting `find` method calls as part of `params.query`. Querying also applies `update`, `patch` and `remove` method calls if the `id` is set to `null`.

> **Important**: When used via REST URLs all query values are strings. Depending on the service the values in `params.query` might have to be converted to the right type in a before hook.

Equality

All fields that do not contain special query parameters are compared directly for equality.

```
// Find all unread messages in room #2
app.service('messages').find({
  query: {
    read: false,
    roomId: 2
  }
});
```

```
GET /messages?read=false&roomId=2
```

`$limit`

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

```
// Retrieves the first two unread messages
app.service('messages').find({
  query: {
    $limit: 2,
    read: false
  }
});
```

```
GET /messages?$limit=2&read=false
```

> **Pro tip:** With pagination enabled, to just get the number of available records set `$limit` to `0`. This will only run a (fast) counting query against the database and return a page object with the `total` and an empty `data` array.

`$skip`

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

```
// Retrieves the next two unread messages
app.service('messages').find({
  query: {
    $limit: 2,
    $skip: 2,
    read: false
  }
```

```
});
```

```
GET /messages?$limit=2&$skip=2&read=false
```

$sort

`$sort` will sort based on the object you provide. It can contain a list of properties by which to sort mapped to the order (`1` ascending, `-1` descending).

```
// Find the 10 newest messages
app.service('messages').find({
  query: {
    $limit: 10,
    $sort: {
      createdAt: -1
    }
  }
});
```

```
/messages?$limit=10&$sort[createdAt]=-1
```

$select

`$select` allows to pick which fields to include in the result. This will work for any service method.

```
// Only return the `text` and `userId` field in a message
app.service('messages').find({
  query: {
    $select: [ 'text', 'userId' ]
  }
});

app.service('messages').get(1, {
  query: {
    $select: [ 'text' ]
  }
});
```

```
GET /messages?$select[]=text&$select[]=userId
GET /messages/1?$select[]=text
```

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

$in , $nin

Find all records where the property does (`$in`) or does not (`$nin`) match any of the given values.

```
// Find all messages in room 2 or 5
app.service('messages').find({
  query: {
    roomId: {
      $in: [ 2, 5 ]
    }
  }
```

```
});
```

```
GET /messages?roomId[$in]=2&roomId[$in]=5
```

`$lt` , `$lte`

Find all records where the value is less (`$lt`) or less and equal (`$lte`) to a given value.

```
// Find all messages older than a day
const DAY_MS = 24 * 60 * 60 * 1000;

app.service('messages').find({
  query: {
    createdAt: {
      $lt: new Date().getTime() - DAY_MS
    }
  }
});
```

```
GET /messages?createdAt[$lt]=1479664146607
```

`$gt` , `$gte`

Find all records where the value is more (`$gt`) or more and equal (`$gte`) to a given value.

```
// Find all messages within the last day
const DAY_MS = 24 * 60 * 60 * 1000;

app.service('messages').find({
  query: {
    createdAt: {
      $gt: new Date().getTime() - DAY_MS
    }
  }
});
```

```
GET /messages?createdAt[$gt]=1479664146607
```

`$ne`

Find all records that do not equal the given property value.

```
// Find all messages that are not marked as archived
app.service('messages').find({
  query: {
    archived: {
      $ne: true
    }
  }
});
```

```
GET /messages?archived[$ne]=true
```

$or

Find all records that match any of the given criteria.

```
// Find all messages that are not marked as archived
// or any message from room 2
app.service('messages').find({
  query: {
    $or: [
      { archived: { $ne: true } },
      { roomId: 2 }
    ]
  }
});
```

```
GET /messages?$or[0][archived][$ne]=true&$or[1][roomId]=2
```

$search

`$search` is not a public API, but may be added through hooks. For example, hooks exists for NeDB and MongoDB:

- feathers-nedb-fuzzy-search
- feathers-mongodb-fuzzy-search
- feathers-solr

Example usage:

```
// Find all messages that contains the text 'hello'
app.service('messages').find({
  query: {
    $search: 'hello'
  }
});
```

```
GET /messages?$search=hello
```

Feathers v3 (Buzzard)

Quick upgrade

To quickly upgrade any Feathers plugin or application you can use the `upgrade` command from the new CLI. First, if you have it installed, uninstall the old `feathers-cli`:

```
npm uninstall feathers-cli -g
```

Then install `@feathersjs/cli` and upgrade a project:

```
npm install @feathersjs/cli -g
cd path/to/project
feathers upgrade
```

In short (for more details see below) this will:

- Upgrade all core packages to the new scoped package names and their latest versions
- Remove all `feathers-hooks` imports and single line `app.configure(hooks());` (chained `.configure(hooks())` calls will have to be removed manually))
- Add Express compatibility to any application that uses `feathers-rest` (other Feathers apps without `feathers-rest` have to be updated manually)
- Remove all `.filter` imports and calls to `service.filter` which has been replaced by channel functionality

Adding channels

If you are using real-time (with Socket.io or Primus), add the following file as `src/channels.js`:

```
module.exports = function(app) {
  if(typeof app.channel !== 'function') {
    // If no real-time functionality has been configured just return
    return;
  }

  app.on('connection', connection => {
    // On a new real-time connection, add it to the anonymous channel
    app.channel('anonymous').join(connection);
  });

  app.on('login', (authResult, { connection }) => {
    // connection can be undefined if there is no
    // real-time connection, e.g. when logging in via REST
    if(connection) {
      // Obtain the logged in user from the connection
      // const user = connection.user;

      // The connection is no longer anonymous, remove it
      app.channel('anonymous').leave(connection);

      // Add it to the authenticated user channel
      app.channel('authenticated').join(connection);

      // Channels can be named anything and joined on any condition

      // E.g. to send real-time events only to admins use
      // if(user.isAdmin) { app.channel('admins').join(connection); }
```

```
      // If the user has joined e.g. chat rooms
      // if(Array.isArray(user.rooms)) user.rooms.forEach(room => app.channel(`rooms/${room.id}`).join(channel)
);

      // Easily organize users by email and userid for things like messaging
      // app.channel(`emails/${user.email}`).join(channel);
      // app.channel(`userIds/$(user.id}`).join(channel);
    }
  });

  app.publish((data, hook) => { // eslint-disable-line no-unused-vars
    // Here you can add event publishers to channels set up in `channels.js`
    // To publish only for a specific event use `app.publish(eventname, () => {})`

    // e.g. to publish all service events to all authenticated users use
    return app.channel('authenticated');
  });

  // Here you can also add service specific event publishers
  // e..g the publish the `users` service `created` event to the `admins` channel
  // app.service('users').publish('created', () => app.channel('admins'));

  // With the userid and email organization from above you can easily select involved users
  // app.service('messages').publish(() => {
  //   return [
  //     app.channel(`userIds/${data.createdBy}`),
  //     app.channel(`emails/${data.recipientEmail}`)
  //   ];
  // });
};
```

And require and configure it in `src/app.js` (note that it should be configured after all services so that `channels.js` can register service specific publishers):

```
const channels = require('./channels');

// After `app.configure(services)`
app.configure(channels);
```

> **Very important:** The `channels.js` file shown above will publish all real-time events to all authenticated users. This is already safer than the previous default but you should carefully review the channels documentation and implement appropriate channels so that only the right users are going to receive real-time events.

Once you migrated your application to channels you can remove all `<servicename>.filter.js` files.

Protecting fields

Feathers v3 has a new mechanism to ensure that sensitive information never gets published to any client. To protect always protect the user password, add the protect hook in `src/services/users/users.hooks.js` instead of the `remove('password')` hook:

```
const { hashPassword } = require('@feathersjs/authentication-local').hooks;

module.exports = {
  before: {
    all: [],
    find: [ authenticate('jwt') ],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },
```

```
  after: {
    all: [
      // Make sure the password field is never sent to the client
      // Always must be the last hook
      protect('password')
    ],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  },

  error: {
    all: [],
    find: [],
    get: [],
    create: [],
    update: [],
    patch: [],
    remove: []
  }
};
```

`@feathersjs` npm scope

All Feathers core modules have been moved to the `@feathersjs` npm scope. This makes it more clear which modules are considered core and which modules are community supported and also allows us to more easily manage publishing permissions. The following modules have been renamed:

Main Feathers

Old name	Scoped name
feathers	@feathersjs/feathers
feathers-cli	@feathersjs/cli
feathers-commons	@feathersjs/commons
feathers-rest	@feathersjs/express/rest
feathers-socketio	@feathersjs/socketio
feathers-primus	@feathersjs/primus
feathers-errors	@feathersjs/errors
feathers-configuration	@feathersjs/configuration
feathers-socket-commons	@feathersjs/socket-commons

Authentication

Old name	Scoped name
feathers-authentication	@feathersjs/authentication
feathers-authentication-jwt	@feathersjs/authentication-jwt
feathers-authentication-local	@feathersjs/authentication-local

feathers-authentication-oauth1	@feathersjs/authentication-oauth1
feathers-authentication-oauth2	@feathersjs/authentication-oauth2
feathers-authentication-client	@feathersjs/authentication-client

Client side Feathers

Old name	Scoped name
feathers/client	@feathersjs/feathers
feathers-client	@feathersjs/client
feathers-rest/client	@feathersjs/rest-client
feathers-socketio/client	@feathersjs/socketio-client
feathers-primus/client	@feathersjs/primus-client
feathers-authentication/client	@feathersjs/authentication-client

Documentation changes

With a better focus on Feathers core, the repositories, documentation and guides for non-core module have been moved to more appropriate locations:

- Non-core modules have been moved to the feathersjs-ecosystem and feathers-plus organizations
- Database adapter specific documentation can now be found in the respective repositories readme. Links to the repositories can be found in the database adapters chapter
- The `feathers-hooks-common` documentation can be found at feathers-plus.github.io/v1/feathers-hooks-common/
- Offline and authentication-management documentation can also be found at feathers-plus.github.io
- The Ecosystem page now points to awesome-feathersjs

Framework independent

`@feathersjs/feathers` v3 is framework independent and will work on the client and in Node out of the box. This means that it is not extending Express by default anymore.

Instead `@feathersjs/express` provides the framework bindings and the REST provider (previously `feathers-rest`) in either `require('@feathersjs/express').rest` or `@feathersjs/express/rest`. `@feathersjs/express` also brings Express built-in middleware like `express.static` and the recently included `express.json` and `express.urlencoded` body parsers. Once a Feathers application is "expressified" it can be used like the previous version:

Before

```
const feathers = require('feathers');
const bodyParser = require('body-parser');
const rest = require('feathers-rest');
const errorHandler = require('feathers-errors/handler');

const app = feathers();

app.configure(rest());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Register an Express middleware
```

```
app.get('/somewhere', function(req, res) {
  res.json({ message: 'Data from somewhere middleware' });
});
// Statically host some files
app.use('/', feathers.static(__dirname));

// Use a Feathers friendly Express error handler
app.use(errorHandler());
```

Now

```
const feathers = require('@feathersjs/feathers');
const express = require('@feathersjs/express');

// Create an Express compatible Feathers application
const app = express(feathers());

// Add body parsing middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
// Initialize REST provider (previous in `feathers-rest`)
app.configure(express.rest());

// Register an Express middleware
app.get('/somewhere', function(req, res) {
  res.json({ message: 'Data from somewhere middleware' });
});
// Statically host some files
app.use('/', express.static(__dirname));

// Use a Feathers friendly Express error handler
app.use(express.errorHandler());
```

Hooks in core

The `feathers-hooks` plugin is now a part of core and no longer has to be imported and configured. All services will have hook functionality included right away. Additionally it is now also possible to define different data that should be sent to the client in `hook.dispatch` which allows to properly secure properties that should not be shown to a client.

Before

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

const app = feathers();

app.configure(hooks());
app.use('/todos', {
  get(id) {
    return Promise.resolve({
      message: `You have to do ${id}`
    });
  }
});

app.service('todos').hooks({
  after: {
    get(hook) {
      hook.result.message = `${hook.result.message}!`;
    }
  }
});
```

Now

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

const app = feathers();

app.use('/todos', {
  get(id) {
    return Promise.resolve({
      message: `You have to do ${id}`
    });
  }
});

app.service('todos').hooks({
  after: {
    get(hook) {
      hook.result.message = `${hook.result.message}!`;
    }
  }
});
```

Event channels and publishing

Previously, filters were used to run for every event and every connection to determine if the event should be sent or not.

Event channels are a more secure and performant way to define which connections to send real-time events to. Instead of running for every event and every connection you define once which channels a connection belongs to when it is established or authenticated.

```
// On login and if it is a real-time connectionn, add the connection to the `authenticated` channel
app.on('login', (authResult, { connection }) => {
  if(connection) {
    const { user } = connection;

    app.channel('authenticated').join(connection);
  }
});

// Publish only `created` events from the `messages` service
app.service('messages').publish('created', (data, context) => app.channel('authenticated'));

// Publish all real-time events from all services to the authenticated channel
app.publish((data, context) => app.channel('authenticated'));
```

To only publish to rooms a user is in:

```
// On login and if it is a real-time connection, add the connection to the `authenticated` channel
app.on('login', (authResult, { connection }) => {
  if(connection) {
    const { user } = connection;

    // Join `authenticated` channel
    app.channel('authenticated').join(connection);

    // Join rooms channels for that user
    rooms.forEach(roomId => app.channel(`rooms/${roomId}`).join(connection));
  }
});
```

Better separation of client and server side modules

Feathers core was working on the client and the server since v2 but it wasn't always entirely clear which related modules should be used how. Now all client side connectors are located in their own repositories while the main Feathers repository can be required the same way on the client and the server.

Before

```
const io = require('socket.io-client');
const feathers = require('feathers/client');
const hooks = require('feathers-hooks');
const socketio = require('feathers-socketio/client');
const auth = require('feathers-authentication-client');

const socket = io();
const app = feathers()
  .configure(hooks())
  .configure(socketio(socket))
  .configure(auth());
```

Now

```
const io = require('socket.io-client');
const feathers = require('@feathersjs/feathers');
const socketio = require('@feathersjs/socketio-client');
const auth = require('@feathersjs/authentication-client');

const socket = io();
const app = feathers()
  .configure(socketio(socket))
  .configure(auth());
```

Node 6+

The core repositories mentioned above also have been migrated to be directly usable (e.g. when npm installing the repository as a Git/GitHub dependency) without requiring a Babel transpilation step.

Since all repositories make extensive use of ES6 that also means that Node 4 is no longer supported.

Also see /feathers/issues/608.

A new Socket message format

The websocket messaging format has been updated to support proper error messages when trying to call a non-existing service or method (instead of just timing out). Using the new `@feathersjs/socketio-client` and `@feathersjs/primus-client` will automatically use that format. You can find the details in the Socket.io client and Primus client documentation.

> **Note:** The old message format is still supported so the clients do not have to be updated at the same time.

Deprecations and other API changes

- Callbacks are no longer supported in Feathers service methods. All service methods always return a Promise. Custom services must return a Promise or use `async/await`.
- `service.before` and `service.after` have been replaced with a single `app.hooks({ before, after })`

- `app.service(path)` only returns a service and cannot be used to register a new service anymore (via `app.service(path, service)`). Use `app.use(path, service)` instead.
- Route parameters which were previously added directly to `params` are now in `params.route`
- Express middleware like `feathers.static` is now located in `const express = require('@feathersjs/express')` using `express.static`

Backwards compatibility polyfills

Besides the steps outlined above, existing hooks, database adapters, services and other plugins should be fully compatible with Feathers v3 without any additional modifications.

This section contains some quick backwards compatibility polyfills for the breaking change that can be used to make the migration easier or continue to use plugins that use deprecated syntax.

Basic service filter

This is a basic emulation of the previous event filter functionality. It does not use promises or allow modifying the data (which should now be handled by setting `hook.dispatch`).

```
app.mixins.push(service => {
  service.mixin({
    filter(eventName, callback) {
      const args = callback ? [ eventName ] : [];

      // todos.filter('created', function(data, connection, hook) {});
      if(!callback) {
        callback = eventName;
      }

      // A publisher function that sends to the `authenticated`
      // channel that we defined in the quick upgrade section above
      args.push((data, hook) => app.channel('authenticated')
        .filter(connection =>
          callback(data, connection, hook)
        )
      );

      service.publish(... args);
    }
  });
});
```

Route parameters

```
app.hooks({
  before(hook) {
    Object.assign(hook.params, hook.params.route);

    return hook;
  }
})
```

`.before` and `.after` hook registration

```
app.mixins.push(service => {
  service.mixin({
    before(before) {
```

```
      return this.hooks({ before });
    },

    after(after) {
      return this.hooks({ after });
    },
  })
});
```

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 the 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 vulnerabilites 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, or anything else your users type directly.

For more information see:

- this issue
- and this Auth0 forum thread.

Help!

There are many ways that you can get help but before you explore them please check the other parts of these docs, the FAQ, Stackoverflow, Github Issues and our Medium publication.

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 create custom methods?

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 Feathers philosophy 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 either by creating a custom service or through hooks. For example, a `userService.resetPassword` method can also be implemented as a password service that resets the password in the `create` method:

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

class PasswordService {
  create(data) {
    const userId = data.user_id;
    const userService = this.app.service('user');

    return userService.patch(userId, {
      passwordToken: crypto.randomBytes(48)
    }).then(user => sendEmail(user))
  }

  setup(app) {
    this.app = app;
  }
}
```

For more examples also see this issue comment.

How do I do nested or custom routes?

Normally we find that they actually aren't needed and that it is much better to keep your routes as flat as possible. For example something like `users/:userId/posts` is - although nice to read for humans - actually not as easy to parse and process as the equivalent `/posts?userId=<userid>` that is already supported by Feathers out of the box. Additionaly, this will also work much better when using Feathers through websocket connections which do not have a concept of routes at all.

However, nested routes for services can still be created by registering an existing service on the nested route and mapping the route parameter to a query parameter like this:

```
app.use('/posts', postService);
app.use('/users', userService);

// re-export the posts service on the /users/:userId/posts route
```

```
app.use('/users/:userId/posts', app.service('posts'));

// A hook that updates `data` with the route parameter
function mapUserIdToData(context) {
  if(context.data && context.params.route.userId) {
    context.data.userId = context.params.route.userId;
  }
}

// For the new route, map the `:userId` route parameter to the query in a hook
app.service('users/:userId/posts').hooks({
  before: {
    find(context) {
      context.params.query.userId = context.params.route.userId;
    },
    create: mapUserIdToData,
    update: mapUserIdToData,
    patch: mapUserIdToData
  }
})
```

Now going to `/users/123/posts` will call `postService.find({ query: { userId: 123 } })` and return all posts for that user.

For more information about URL routing and parameters, refer to the Express chapter.

> **Note:** URLs should never contain actions that change data (like `post/publish` or `post/delete`). This has always been an important part of the HTTP protocol and Feathers enforces this more strictly than most other frameworks. For example to publish a post you would call `.patch(id, { published: true })`.

How do I do search?

This depends on the database adapter you are using. Many databases already support their own search syntax:

- Regular expressions (converted in a hook) for Mongoose, MongoDB and NeDB, see this comment
- $like for Sequelize which can be set in params.sequelize
- Some database adapters like KnexJS, RethinkDB and Elasticsearch also support non-standard query parameters which are described in their documentation pages.

For further discussions see this issue.

Why am I not getting JSON errors?

If you get a plain text error and a 500 status code for errors that should return different status codes, make sure you have the `feathers-errors/handler` configured as described in the Express errors chapter.

Why am I not getting the correct HTTP error code

See the above answer.

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 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

In Feathers channels are the way to send real-time events to only certain clients.

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 do associations?

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.

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. We show how to associate data in a hook in the chat guide. An alternative is the populate hook in feathers-hooks-common.

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') }
    }]
  }
});
```

Or set it in a hook as described here.

Sequelize models and associations

If you are using the Sequelize adapter, understanding SQL and Sequelize first is very important. For further information see this documentation chapter and this answer on Stackoverflow.

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. Currently, Feathers is framework independent but only provides an Express integration for HTTP APIs. More frameworks may be supported in the future with direct Node HTTP being the most likely.

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. More information can be found here.

How do I do some processing after sending the response to the user?

The hooks workflow allows you to handle these situations quite gracefully. It depends on the promise that you return in your hook. Here's an example of a hook that sends an email, but doesn't wait for a success message.

```
function (context) {

  // Send an email by calling to the email service.
  context.app.service('emails').create({
    to: 'user@email.com',
    body: 'You are so great!'
  });

  // Send a message to some logging service.
  context.app.service('logging').create(context.data);

  // Return a resolved promise to immediately move to the next hook
  // and not wait for the two previous promises to resolve.
  return Promise.resolve(context);
}
```

How do I debug my app

It's really no different than debugging any other NodeJS app but you can refer to this blog post for more Feathers specific tips and tricks.

`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.

Why can't I pass `params` from the client?

When you make a call like:

```
const params = { foo: 'bar' };
client.service('users').patch(1, { admin: true }, params).then(result => {
  // handle response
});
```

on the client the `context.params` object will only be available in your client side hooks. It will not be provided to the server. The reason for this is because `context.params` on the server usually contains information that should be server-side only. This can be database options, whether a request is authenticated, etc. If we passed those directly from the client to the server this would be a big security risk. Only the client side `context.params.query` and `context.params.headers` objects are provided to the server.

If you need to pass info from the client to the server that is not part of the query you need to add it to `context.params.query` on the client side and explicitly pull it out of `context.params.query` on the server side. This can be achieved like so:

```
// client side
client.hooks({
  before: {
    all: [
      context => {
        context.params.query.$client = {
```

```
          platform: 'ios',
          version: '1.0'
        };

        return context;
      }
    ]
  }
});

// server side, inside app.hooks.js
const hooks = require('feathers-hooks-common');

module.exports = {
  before: {
    all: [
      // remove values from context.params.query.$client and move them to context.params
      // so context.params.query.$client.version -> context.params.version
      // and context.params.query.$client is removed.
      hooks.client('version', 'platform')
    ]
  }
}
```

Why are queries with arrays failing?

If you are using REST and queries with larger arrays (more than 21 items to be exact) are failing you are probably running into an issue with the querystring module which limits the size of arrays to 21 items by default. The recommended solution is to implement a custom query string parser function via `app.set('query parser', parserFunction)` with the `arrayLimit` option set to a higher value. For more information see the Express application settings @feathersjs/rest#88 and feathers-mongoose#205.

I always get a 404 for my custom middleware

Just like in Express itself, the order of middleware matters. If you registered a custom middleware outside of the generator, you have to make sure that it runs before the `notFound()` error midlleware.

How do I get OAuth working across different domains

The standard Feathers oAuth setup sets the JWT in a cookie which can only be passed between the same domain. If your frontend is running on a different domain you will have to use query string redirects as outlined in this Gist.

How do I set up HTTPS?

Check out the Feathers Express HTTPS docs.

Is Feathers production ready?

Yes! It's being used in production by a bunch of companies from startups to fortune 500s. For some more details see this answer on Quora.

Contributing

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.

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 content related to databases is located in `api/databases/`.

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

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

This is the root README.md file. It's intent is to give the reader an 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 to let others know about your changes.

MIT license