The OrbitDB Field Manual

Mark Robert Henderson Samuli Pöyhtäri Vesa-Ville Piiroinen Juuso Räsänen
Shams Methnani Richard Littauer

Abstract

Straight from the creators of OrbitDB. Contains an end-to-end tutorial, an in-depth look at OrbitDB's underlying architecture, and even some philsophical musings about decentralization and the distributed industry.

Contents

Introduction

Preamble

What is OrbitDB?

What can I use OrbitDB for?

A Warning

Part 1: The OrbitDB Tutorial

> An interactive, imperative and isometric JavaScript adventure of peer-to-peer, decentralized, and distributed proportions

Introduction

Requirements

- A computer
- A command line (unix/linux based or windows command prompt)
- A modern web browser (Firefox, Chrome, Edge, etc)
- node.js (optional)

What will I learn?

TODO: Description of tutorial chapters, after they are written

What will I build?

If you're a musician like many of us are, you probably need form of sheet music to practice. While any musician tends to spend hours practicing alone, the beauty of music is its ability to *connect* musicians to play together - duets, trios, quartets, septets, all the way up to orchestras.

These self-organizing clusters of musicians will always need better way to share common and necessary sheet music with each other. What better use case for a peer-to-peer application?

Using OrbitDB as the backbone, you will build this application. It will allow people to import and maintain a local collection of their own sheet music in the form of PDF files. More importantly, they will be able to *share* this music by letting them interface with *peers*, and search across multiple distribute databases at once for music. For fun, and for users who are just looking for something to-sight read, you will give them a magic "button" that, given an instrument, will display piece of sheet music at random from their collection.

Why a music app?

OrbitDB is already used all over the world, and this tutorial music reflect that. There are other many topics we could have chosen that touch the vast majority of humans on earth: finance, politics, climate, religion. However, those are generally contentious and complicated.

We believe that **music** is a uniquely universal cultural feature - something that we more humans than any other topic share, enjoy, or at least appreciate. Your participation in this tutorial will make it easier for musicians all over the world to find sheet music to practice with.

Conventions

- Read this tutorial in order, the learning builds on itself other over time.
- Type the examples in, don't copy and paste.
- **What just happened?** sections are interspersed to that explain in depth what happens on a technical level
- This tutorial attempts to be as *agnostic* as possible in terms of:
- Your operating system. Some command line commands are expressed as unix commands, but everything here should work on Windows as well.

- Your folder structure. All of the code here is written in a single file, `newpieceplease.js`
- Your editor. Use whatever you want.
- Your UI layer. You will see *examples* of how the code will be used on the UI layer, be it command line or browser based, but you will not be building out the UI as part of the tutorial steps.
- `async` and `await` are used prominently. Feel free to replace those with explicit `Promise` objects if you're feeling daring.
- Steps that **you** should complete are represented and hightlighted as *diffs*. Example application code is represented as Javascript

- For the sake of keeping things focused, we will exclude any HTML or CSS from this tutorial and focus only on the Javascript code.

Ready? Let's start with Chapter 1: Laying the Foundation

Chapter 1 - Laying the Foundation

> The basics of OrbitDB include *installing OrbitDB (and IPFS)*, *setting up a new isomorphic project*, *creating databases*, and how *understanding how to choose data stores*.

Table of Contents

Please see the Introduction before beginning this chapter.

Installing the requirements: IPFS and OrbitDB

You will need to get the code for OrbitDB and its dependency, IPFS, and make it available to your project. The process is different between the browser and node.js, so we cover both here.

In node.js

Choose a project directory and `cd` to there from your command line. Then run the following command.

```
$ npm init
... enter commands to create package.json ...

$ npm install orbit-db ipfs
```

This will create a `package.json`, `package.lock`, and `node_modules` folder.

In the Browser

For this tutorial, we recommend using unpkg for obtaining pre-built, minified versions of both IPFS and OrbitDB. Simply include these at the top of your `index.html` file:

```
<script src="https://unpkg.com/ipfs/dist/index.min.js"></script>
<script src="https://www.unpkg.com/orbit-db/src/OrbitDB.js"></script>
```

You will now have global `Ipfs` and `OrbitDB` objects available to you. You will see how we'll use these later.

Both OrbitDB and js-ipfs are open source, which give you the ability to build and even contribute to the code. This will be covered in detail these in Part 3.

Creating the isomorphic bookends

Since OrbitDB works in the browser and node.js, we're going to want to make our app as *isomorphic* as possoble. This means we want the same code to run in the browser as runs in JS. This is good news for the tutorial, as it means we can keep our code to **strictly** things that pertain to our app, and then apply bindings in node.js and

Luckily, you will have the luxury of using the same language, JavaScript, for both node.js and browser environments. Create a new file called `newpieceplease.js` and put this code in there:

```
+ try {
+   const Ipfs = require('ipfs')
+   const OrbitDB = require('orbit-db')
+ } catch(e) {}

+ class NewPiecePlease() {
+   constructor(IPFS, OrbitDB) { }
+ }

+ try {
+   module.exports = exports = new NewPiecePlease(IPFS, OrbitDB)
+ } catch (e) {
+   window.NPP = new NewPiecePlease(window.Ipfs, window.OrbitDB)
+ }
```

What just happened?

Using some key JavaScript features, you have created the shell for our application that runs in both node.js and the browser. It defines a new class called `NewPiecePlease`, with a constructor that takes two arguments

1. `IPFS` for the `js-ipfs` constructor
2. `OrbitDB` for the `orbit-db` constructor

In the browser, you can include this file in a script tag and have an `NPP` object at your disposal. In node.js, you can simply call something like:

```
const NPP = require('./newpieceplease')
```

From here on out, we will ignore these isometric bookends and concentrate wholly on the `NewPiecePlease` class.

Instantiating IPFS and OrbitDB

We have designed Chapters 1 and 2 of the tutorial to work work offline, not requiring any internet connectivity or connections to peers.

OrbitDB requires a running IPFS node to operate, so you will create one here and notify OrbitDB about it. by running the following code. It's a lot but it constitutes the frame for an *isomorphic* JavaScript app, that is, one that runs in both the browser and in node.js with the same code.

```
class NewPiecePlease() {
-   constructor(IPFS, OrbitDB) { }
+   constructor(IPFS, OrbitDB) {
+     let node = new IPFS({
+       preload: { enabled: false },
+       repo: "./ipfs",
+       EXPERIMENTAL: { pubsub: true },
+       config: {
+         Bootstrap: [],
+         Addresses: { Swarm: [] }
+       }
+     });
+
```

```
+     node.on("error", (e) => { throw (e) })
+     node.on("ready", this._init.bind(this))
+   }
+
+   async _init() {
+     this.orbitdb = await OrbitDB.createInstance(this.node)
+
+     this.onready()
+   }
}
```

This allows you to run something like the following in your application code:

```
NPP.onready = () => {
   console.log(NPP.orbitdb.id)
}
```

In the output you will see something called a "multihash", like `QmPSicLtjhsVifwJftnxncFs4EwYTBEjKUzWweh1nAA87B`. For now, just know that this is the identifier of your IPFS node. We explain multihashes in more detail in **Part 2: Peer-to-Peer**

What just happened?

Start with the `new Ipfs` line. This code creates a new IPFS node. Note the default settings:

- `preload: { enabled: false }` disables the use of so-called "pre-load" IPFS nodes. These nodes exist to help load balance the global network and prevent DDoS. However, these nodes can go down and cause errors. Since we are only working offline for now, we include this line to disable them.
- `repo: './ipfs'` designates the path of the repo in node.js only. In the browser, you can actually remove this line. The default setting is a folder called `.jsipfs` in your home directory. You will see why we choose this acute location for the folder later.
- `XPERIMENTAL: { pubsub: true }` enables IPFS pubsub, which is a method of communicating between nodes and is required for OrbitDB usage, despite whether or not we are connected to other peers.
- `config: { Bootstrap: [], Addresses: { Swarm: [] }}` sets both our bootstrap peers list (peers that are loaded on instantiation) and swarm peers list (peers that can connect and disconnect at any time to empty. We will populate these later.
- `node.on("error", (e) => { throw new Error(e) })` implements extremely basic error handling for if something happens during node creation
- `node.on("ready", (e) => { orbitdb = new OrbitDB(node) })` instantiates OrbitDB on top of the IPFS node, when it is ready.

By running the code above, you have created a new IPFS node that works locally and is not connected to any peers. You have also loaded a new `orbitdb` object into memory, ready to create databases and manage data.

You are now ready to use OrbitDB!

What else happened in node.js?

When you ran the code in node.js, you created two folders in your project structure: `'orbitdb/` and `ipfs/`.

```
$ # slashes added to ls output for effect
$ ls orbitdb/
QmNrPunxswb2Chmv295GeCvK9FDusWaTr1ZrYhvWV9AtGM/

$ ls ipfs/
blocks/  config  datastore/  datastore_spec  keys/  version
```

The code will always create the `orbitdb/` folder as a sibling to the location specified in the `repo` paramater of the IPFS constructor options. Looking inside of the `orbitdb/` folder, you will see that the subfolder has the same ID as orbitdb, as well as the IPFS node. This is purposeful, as this initial folder contains metadata that OrbitDB will need to operate. See Part 3 for detailed information about this.

> *Note:* The `ipfs/` folder contains all of your IPFS data. Explaining this in depth is outside of the scope of this tutorial, and the curious can find out more here.

What else happened in the browser?

In the browser IPFS content is handled inside of IndexedDB, a persistent storage mechanism for browsers

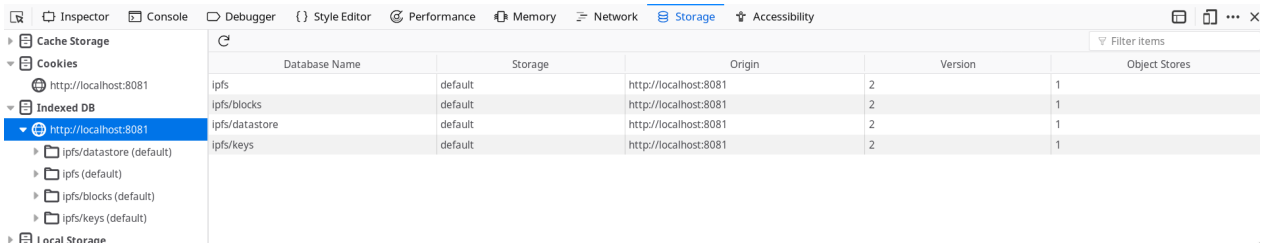

Figure 1: An image showing the IPFS IndexedDB databases in Firefox

Note since you have not explicitly defined a database in the broser, no IndexedDB databases have been created for OrbitDB yet.

> **Caution!** iOS and Android have been known to purge IndexedDB if storage space needs to be created inside of your phone. We recommend creating robust backup mechanisms at the application layer

Creating a Database

Now, you will create a local database that *only you* can read.

Expand of your `_init` function to the following:

```
  async _init {
    this.orbitdb = await OrbitDB.createInstance(node)
+   this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}
+
+   const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
+   this.pieces = await orbitdb.docstore('pieces', options)
  }
```

Then, in your application code, run this:

```
NPP.onready = () => {
   console.log(NPP.pieces.id)
}
```

You will see something like the following as an output: `/orbitdb/zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/p`
This is the id, or **address** (technically a multiaddress) of this database. It's important for you to not only *know* this, but also to understand what it is.

The first bit, `/orbitdb`, is the protocol. It tells you that this address is an OrbitDB address. The last bit, `pieces` is simply the name you provided.

It's the second, or middle, part `zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3` that is the most interesting. This value comes from the combining three pieces of data, and then hashing them:

1. The **access control list** of the database
2. The **type** of the database
3. The **name** of the database

   Note: Misunderstanding OrbitDB addressing can lead to some very unexpected - sometimes hilarious, sometimes disastrous outcomes. Read more in Part 2 to learn more.

What just happened?

Your code created a local OrbitDB database, of type "docstore", writable only by you.

- The `options` defines the paramaters for the database we are about to create.
- `accessController: { write: [orbitdb.identity.publicKey] }` defines the ACL, or "Access Control List". In this instance we are restricting `write` access to ONLY orbitdb instances identified by our particular `publicKey`
- `indexBy: "hash"` is a docstore-specific option, which specifies which field to index our database by
- `pieces = await orbitdb.docstore('pieces', options)` is the magic line that creates the database. Once this line is completed, the database is open and can be acted upon.

  Caution! A note about identity: Your public key is not your identity. We repeat, *your public key is not your identity.* That being said, it is often used as such for convenience's sake, and the lack of better alternatives. So, in the early parts of this tutorial we say "writable only to you" when we really mean "writable only by an OrbitDB instance on top of an IPFS node that has the correct id, which we are assuming is controlled by you."

See for more info: link

What else happened in node.js?

You will see some activity inside your project's `orbitdb/` folder. This is good.

```
$ ls orbitdb/
QmNrPunxswb2Chmv295GeCvK9FDusWaTr1ZrYhvWV9AtGM/  zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/

$ ls orbitdb/zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/
pieces/

$ ls orbitdb/zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/pieces/
000003.log  CURRENT  LOCK  LOG  MANIFEST-000002
```

You don't need to understand this fully for now, just know that it happened. Two subfolders, one being the original folder you saw when you instantiated OrbitDB, and now another that has the same address as your database.

What else happened in the browser?

Similarly, a new IndexedDB database was created to hold your OrbitDB-specific info, apart from the data itself which are still stored in IPFS.

This shows you one of OrbitDB's core strenths - the ability to manage a lot of complexity between its own internals and tht of IPFS, providing a clear and clean API to manage the data that matters to you.

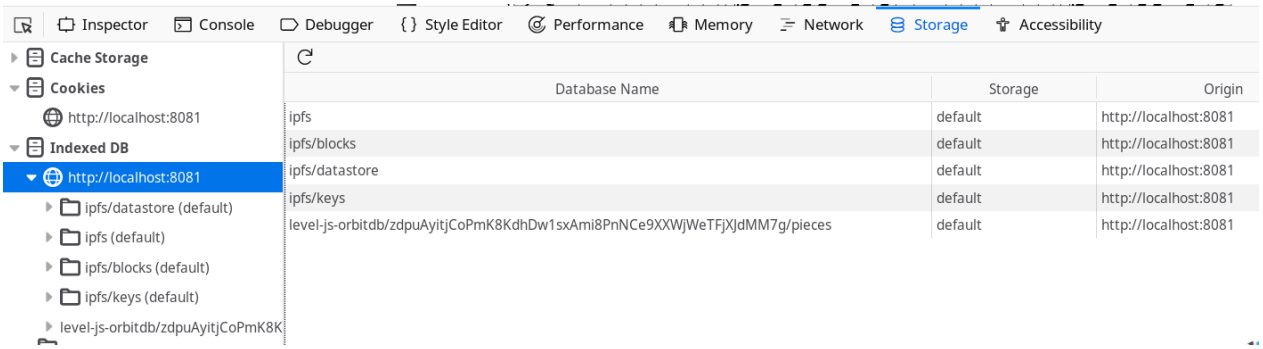

Figure 2: An image showing the IPFS and OrbitDB IndexedDB databases in Firefox

Choosing a data store

OrbitDB organizes its functionality by separating different data management concerns, schemas and APIs into **stores**. We chose a `docstore` for you in the last chapter, but after this tutorial it will be your job to determine the right store for the job.

At your disposal you have:

- **log**: an immutable (append-only) log with traversable history. Useful for *"latest N"* use cases or as a message queue.
- **feed**: a mutable log with traversable history. Entries can be added and removed. Useful for *"shopping cart"* type of use cases, or for example as a feed of blog posts or "tweets".
- **keyvalue**: a key-value database just like your favourite key-value database.
- **docs**: a document database to which JSON documents can be stored and indexed by a specified key. Useful for building search indices or version controlling documents and data.
- **counter**: Useful for counting events separate from log/feed data.

Each OrbitDB store has its own specific API methods to create, delete, retreieve and update data. In general, you can expect to always have something like a `get` and something like a `put`.

Also, users of OrbitDB can write their own stores if it suits them. This is an advanced topic and is covered in Part 3 of this book.

Key Takeaways

- OrbitDB is a distributed database layer which stores its raw data in IPFS
- Both IPFS and OrbitDB work offline and online
- OrbitDB instances have an *ID* which is the same as the underling IPFS node's ID.
- OrbitDB instances create databases, which have unique *addresses*
- Basic access rights to OrbitDB databases are managed using access control lists (or ACLs), based on the ID of the IPFS node performing the requests on the database
- OrbitDB database addresses are hashes of the database's ACL, its type, and its name.
- Since OrbitDB and IPFS are written in JavaScript, it is possible to build isomorphic applications that run in the browser and in node.js
- OrbitDB manages needed flexibility of schema and API design in functionality called **stores**.
- OrbitDB comes with a handful of stores, and you can write your own.
- Each store will have its own API, but you will generally have at least a `get` and a `put`

Now that you've laid the groudnwork, you'll learn how to work with data! Onward, then, to Chapter 2: Managing Data.

- Resolves #367

- Resolves #366
- Resolves #502

Chapter 2 - Managing Data

> Managing data in OrbitDB involves *loading databases into memory*, and then *creating*, *updating*, *reading*, and *deleting data*.

Table of Contents

Please complete Chapter 1 - Laying the Foundation first.

Loading the database

To start, you'll do a couple of things to enhance our current code and tidy up. We will also scaffold out some functions to be filled in later.

Update your `NewPiecePlease class` handler, adding **one line** at the bottom of the IPFS `ready` handler. and then run this code:

```
  _init() {
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}

    const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
    this.piecesDb = await this.orbitdb.docstore('pieces', options)
+   await this.piecesDb.load()
  }

+ async addNewPiece(hash, instrument = "Piano") { }
+
+ async deletePieceByHash() { }
+
+ getAllPieces() {}
+
+ getPiecesByInstrument(instrument) { }
+
+ getPieceByHash(hash) { }
+
+ await updatePieceByHash(hash, instrument = "Piano")
}
```

What just happened?

After you instantiated the database, you loaded its contents into memory for use. It's empty for now, but not for long! Loading the database at this point after instantiation will save you trouble later.

- `await piecesDb.load()` is a function that will need to be called whenever we want the latest and greatest snapshot of data in the database. `load()` retrieves all of the values via their *content addresses* and loads the content into memory

> **Note:** You're probably wondering about if you have a large database of millions of documents, and the implications of loading them all into memory. It's a valid concern, and you should move on to Part 4 of this book once you're done with the tutorial.

Adding data

Now that you have a database set up, adding content to it is fairly easy. Run the following code to add some sheet music to the repository.

We have uploaded and pinned a few piano scores to IPFS, and will provide the hashes. You can add these hashes to your database by fleshing out and using the `addNewPiece` function.

> **Note:** We hope you like the original Metroid NES game, or at least the music from it!

Fill in your `addNewPiece` function now:

```
- async addNewPiece(hash, instrument = "Piano") { }
+ async addNewPiece(hash, instrument = "Piano") {
+   const existingPiece = this.pieces.get(hash)
+   if(existingPiece) {
+     await this.updatePieceByHash(hash, instrument)
+     return;
+   }
+
+   const cid = await piecesDb.put({
+     hash: hash,
+     instrument: instrument
+   })
+   return cid
+ }
```

Then, in your application code, node.js or browser, you cna use this function like so, utilizing the detault value for the `instrument` argument.

```
const cid = NPP.addNewPiece("QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ")
const content = await NPP.node.dag.get(cid)
console.log(content.value.payload)
```

Running this code should give you something like the following output. Hold steady, it's overwhelming but it will make sense after we explain what happened. For more information see Part 3.

```
{
  "op":"PUT",
  "key":"QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ",
  "value": {
    "hash":"QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ",
    "instrument":"Accordion"
  }
}
```

What just happened?

- `piecesDb.put({ ... })` is the most important line here. This call takes an object to sture and returns a *mutlihash*, which is the hash of the content added to IPFS.
- `node.dag.get(hash)` is a function that takes a Content ID (CID) and returns content.

- `"op": "PUT"`is a notable part of the output. At the core of OrbitDB databases is the **OPLOG**, where all data are stored as a log of operations, which are then calculated into the appropriate schema for application use. The operation is specified here as a `PUT`, and then the `key`/`value` pair is your data.

  Note: "dag" in the code refers to the acronym DAG, which stands for Directed Acyclic Graph. This is a data structure that is, or is at least closely related to Blockchain. More on this in Part 4

You can repeat this process to add more hashes from the NES Metroid soundtrack:

```
QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ | Metroid - Ending Theme.pdf
QmRn99VSCVdC693F6H4zeS7Dz3UmaiBiSYDf6zCEYrWynq | Metroid - Escape Theme.pdf
QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPSL | Metroid - Game Start.pdf
QmcFUvG75QTMok9jrteJzBUXeoamJsuRseNuDRupDhFwA2 | Metroid - Item Found.pdf
QmTjszMGLb5gKWAhFZbo8b5LbhCGJkgS8SeeEYq3P54Vih | Metroid - Kraids Hideout.pdf
QmNfQhx3WvJRLMnKP5SucMRXEPy9YQ3V1q9dDWNC6QYMS3 | Metroid - Norfair.pdf
QmQS4QNi8DCceGzKjfmbBhLTRExNboQ8opUd988SLEtZpW | Metroid - Ridleys Hideout.pdf
QmcJPfExkBAZe8AVGfYHR7Wx4EW1Btjd5MXX8EnHCkrq54 | Metroid - Silence.pdf
Qmb1iNM1cXW6e11srUvS9iBiGX4Aw5dycGGGDPTobYfFBr | Metroid - Title Theme.pdf
QmYPpj6XVNPPYgwvN4iVaxZLHy982TPkSAxBf2rzGHDach | Metroid - Tourian.pdf
QmefKrBYeL58qyVAaJoGHXXEgYgsJrxo763gRRqzYHdL6o | Metroid - Zebetite.pdf
```

These are all stored in the global IPFS network so you can find any piece by visiting a public gateway such as `ipfs.io` and adding the IPFS multiaddress to the end of the URL like so: https://ipfs.io/ipfs/QmYPpj6XVNPPYgwvN4iVaxZLHy982TPkSAxBf2rzGHDach

Reading data

You've added data to your local database, and now you'll can query it. OrbitDB gives you a number of ways to do this, mostly based on which *store* you picked.

We gave you a `docstore` earlier, so you can flesh out the all of thet simple `get*****` functions like so. `docstore` also provides the more puwerful `query` function, which we can abstract to write a `getPiecesByInstrument` function:

Fill in the following functions now:

```
- getAllPieces() {}
+ getAllPieces() {
+   const pieces = this.piecesDb.get('')
+   return pieces
+ }
```

```
- getPieceByHash(hash) { }
+ getPieceByHash(hash) {
+   const singlePiece = this.piecesDb.get(hash)[0]
+   return singlePiece
+ }
```

```
- getByInstrument(instrument) { }
+ getByInstrument(instrument) {
+   return this.piecesDb.query((piece) => piece.instrument === instrument)
+ }
```

In your application code, you can use these functions it like so:

```
pieces = NPP.getAllPieces()
pleces.forEach((piece) => { /* do something */ })
```

```
piece = NPP.getPieceByHash('QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ')
console.log(piece)
```

Pulling a random score from the database is a great way to find random music to practice. Run this code:

```
const pieces = NPP.getPieceByInstrument("Piano")
const randomPiece = pieces[items.length * Math.random() | 0]
console.log(randomPiece)
```

Both `console.log` calls above will return something like this.

```
{
  "hash":"QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ",
  "instrument":"Accordion"
}
```

What just happened?

You queried the database of scores you created earlier in the chapter, retrieving by hash and also randomly.

- `pieces.get(hash)` is a simple function that performs a partial string search on your database indexes. It will return an array of records that match. As you can see in your `getAllPieces` function, you can pass an empty string to return all pieces.
- `return this.piecesDb.query((piece) => piece.instrument === instrument)` queries the database, returning. It's most analagous to JavaScripts `Array.filter` method.

  Note: Generally speaking, `get` functions do not return promises since the calculation of database state happens at the time of a *write*. This is a trade-off to allow for ease of use and performance based on the assumption that writes are *generally* less frequent than reads.

Updating and deleting data

You'll next want to provide your users with the ability to update and delete their pieces. For example if you realize you'd rather practice a piece on a harpsichord instead of a piano, or if they want to stop practicing a certain piece.

Again, each OrbitDB store may have slightly different methods for this. In the `docstore` you can update records by again using the `put` method and the ID of the index you want to update.

Fill in the `updatePieceByHash` and `deletePieceByHash` functions now:

```
- async updatePieceByHash(hash, instrument = "Piano") { }
+ async updatePieceByHash(hash, instrument = "Piano") {
+   var piece = await this.getPieceByHash(hash)
+   piece.instrument = instrument
+   const cid = await this.piecesDb.put(piece)
+   return cid
+ }
```

```
- async deletePieceByHash(hash) {
+ async deletePieceByHash(hash) {
+   const cid = await this.piecesDb.del(hash)
+   return cid
+ }
```

In your application code, you can run these new functions and see the opcodes that return to get a sense of what's going on.

```
const cid = await NPP.updatePiece("QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ", "Harpsichord")
// do stuff with the cid as above

const cid = await NPP.deletePieceByHash("QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ")
const content = await NPP.node.dag.get(cid)
console.log(content.value.payload)
```

While the opcode for PUT will be the same, the opcode for `deletePieceByHash` is not:

```
{
  "op":"DEL",
  "key":"QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPSL",
  "value":null
}
```

What just happened?

You may be thinking something like this: "Wait, if OrbitDB is built upon IPFS and IPFS is immutable, then how are we updating or deleting records?" Great question, and the answer lies in the opcodes Let's step through the code so we can get to that.

- `this.piecesDb.put` is nothing new, we're just using it to perform an update instead of an insert
- `this.piecesDb.del` is a simple function that takes a hash, deletes the record, and returns a CID
- `"op": "DEL"` is another opcode, `DEL` for DELETE. This log entry effectively removes this key from your records and also removes the content from your local IPFS

Storing Media Files

We are often asked if it is possible to store media files like pictures or audio directly inside OrbitDB. Our answer is that you should treat this like any other database system and store the *address* of the

Luckily, with content addressing in IPFS, this becomes rather easy, and predictable from a schema design standpoint. The overall pattern in:

1. Add the file to IPFS, which will return the *multihash* of the file
2. Store said multihash in OrbitDB
3. When it comes time to display the media, use native IPFS functionality to retrieve it from the hash

Adding content to IPFS

To see this in action, download the "Tourian" PDF to your local file system for use in the next examples

On the command line with the go-ipfs or js-ipfs daemon

After following the installation instructions to install go-ipfs or js-ipfs globally, you can run the following command:

```
$ ipfs add file.pdf
QmYPpj6XVNPPYgwvN4iVaxZLHy982TPkSAxBf2rzGHDach
```

You can then use that hash in the same manner as above to add it to the database of pieces.

In Node.js

In Node.JS, adding a file from the filesystem can be accomplished like so:

```
var IPFS = require('ipfs')
var ipfs = new IPFS(/* insert appropriate options here for your local IPFS installation */)

ipfs.addFromFs("./file.pdf").then(console.log)
```

In the browser

If you have a HTML file input with an ID of "fileUpload", you can do something like the following to add content to IPFS:

```
var fileInput = document.getElementById("fileUpload")

var file = fileInput.files[0]
if (file) {
  var reader = new FileReader();
  reader.readAsBinaryString(file)

  reader.onerror = (e) => console.error(e)
  reader.onload = async function (evt) {
    const contents = evt.target.result
    const buffer = NPP.node.types.Buffer(contents)
    const result = await NPP.node.add(buffer)
    const cid = await NPP.addNewPiece(result[0].hash, instrument)
  }
}
```

Note that there are still issues with swarming in the browser, so you may have trouble discovering content. Stay tuned for future `js-ipfs` releases to fix this.

What just happened?

You added some potentially very large media files to IPFS, and then stored the 40-byte addresses in OrbitDB for retrieval and use. You are now able to leverage the benefits of both IPFS and OrbitDB in both the browser and node.js.

> **Note:** IPFS nodes run *inside* the browser, so if you're adding lots of files via the above method, keep an eye on your IndexedDB quotas, since that's where IPFS is storing the blocks.

Key Takeaways

- Calling `load()` periodically ensures you have the latest entries from the database
- Generally speaking, a `put` or `delete` will return a Promise (or require `await`), and a `get` will return the value(s) immediately.
- Updating the database is equivalent to adding a new entry to its OPLOG.
- The OPLOG is calculated to give the current *state* of the database, which is the view you generally interact with
- OPLOGS are flexible, particularly if you're writing your own stores. `docstore` primarily utilizes the `PUT` and `DEL` opcodes
- While you technically *can* store encoded media directly in a database, media files are best stored in OrbitDB as IPFS hashes
- Keep an eye on IndexedDB size and limitations when adding content to IPFS via the browser.

Of course, in the vast majority of apps you create, you won't just be interacting with one database or one type of data. We've got you covered in Chapter 3: Structuring Data

- Resolves #365
- Resolves #438
- Resolves #381
- Resolves #242
- Resolves #430

Chapter 3 - Structuring your data

> or, "How you learned to stop worrying and love *nested databases*."

Table of Contents

Please complete Chapter 2 - Managing Data first.

Adding a practice counter to each piece

Your users may want to keep track of their practice, at minimum how many times they practiced a piece. You'll enable that functionality for them by creating a new OrbitDB `counter` store for each piece, and creating a few new functions inside the `NewPiecePlease` class to interact with the counters.

> **Note:** The nesting approach detailed here is but one of many, and you are free to organize your data as you see fit. This is a powerful feature of OrbitDB and we are excited to see how people tackle this problem in the future!

Update the `addNewPiece` function to create a `counter` store every time a new piece is added to the database. You can utilize basic access control again to ensure that only a node with your IPFS node's ID can write to it.

```
async addNewPiece(hash, instrument = "Piano") {
  const existingPiece = this.pieces.get(hash)
  if(existingPiece) {
    await this.updatePieceByHash(hash, instrument)
    return;
  }

+ const dbName = "counter." + hash.substr(20,20)
+ const counterDb = await this.orbitdb.counter(dbName, this.defaultOptions)

  const cid = await this.pieces.put({
    hash: hash,
    instrument: instrument,
+   counter: counterDb.id
  })

  return cid
}
```

In your application code this would look something like this:

```
const cid = await NPP.addNewPiece("QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPSL", "Piano")
const content = await NPP.node.dag.get(cid)
console.log(content.value.payload.value)
```

Which will then output something like:

```
{
  "hash":"QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPSL",
  "counter":"/orbitdb/zdpuAoM3yZEwsynUgeWPfizmWz5DEFPiQSvg5gUPu9VoGhxjS/counter.fzFwiEu255Nm5WiCey9n",
  "instrument":"Piano"
}
```

What just happened?

You changed your code to add a new database of type `counter` for each new entry added to the database.

- `const options = { accessController: { write: [this.orbitdb.identity.publicKey] }}` should be recognizable from Chapter 1. This sets options for the db, namely the `accessController` to give write access only to your node's ID, or public key.
- `this.orbitdb.counter` creates a new counter type with `options` that provide a write ACL for your IPFS node
- `const dbName = "counter." + hash.substr(20,20)` prepends `counter.` to the truncated database name. See the note below.
- `this.pieces.put` is then modified to store the *address* of this new database for later retrieval similar to the way you stored media addresses in a previous chapter.
- `"counter":"/orbitdb/zdpuAoM3yZEwsynUgeWPfizmWz5DEFPiQSvg5gUPu9VoGhxjS/counter.fzFwiEu255Nm5WiCey9n"` in the output now reflects this change by storing the *address* of the new DB for later retrieval and updating.

  Note: There is a limit of 40 characters on the names of the databases, and multihashes are over this limit at 46. We still need unique names for each of the databases created to generate unique addresses, so we trim down the hash and prepend it with `counter.` to get around this limitation.

Utilizing the practice counter

Now, add a few functions to `NewPiecePlease` that utilize the counters when necessary

```
+ async getPracticeCount(piece) {
+   const counter = await this.orbitdb.counter(piece.counter)
+   await counter.load()
+   return counter.value
+ }

+ async incrementPracticeCounter(piece) {
+   const counter = await this.orbitdb.counter(piece.counter)
+   const cid = await counter.inc()
+   return cid
+ }
```

These can be used in your application code like so:

```
const piece = NPP.getPieceByHash("QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPSL")
const cid = await NPP.incrementPracticeCounter(piece)
const content = await NPP.node.dag.get(cid)
console.log(content.value.payload)
```

That will `console.log` out something like:

```
{
  "op":"COUNTER",
  "key":null,
  "value": {
    "id":"042985dafe18ba45c7f1a57db.........02ae4b5e4aa3eb36bc5e67198c2d2",
    "counters": {
      "042985dafe18ba45c7f1a57db.........02ae4b5e4aa3eb36bc5e67198c2d2":3
    }
  }
}
```

What just happened?

You created and used two new functions to both read the value of, and increment a `counter`, another type of OrbitDB store.

- `await this.orbitdb.counter(piece.counter)` is a new way of using `this.orbitdb.counter`, by passing in an existing database address. This will *open* the existing database instead of creating it
- `counter.load()` is called once in `getPracticeCount`, loading the latest database entries into memory for display
- `await counter.inc()` increments the counter, like calling `counter++` would on an integer variable
- `"op":"COUNTER"` is a new operation that you havent seen yet - remember, you can create stores with any operations you want. More on this in Part 3.
- `"counters": { "042985dafe18ba45c7f1a57db.....02ae4b5e4aa3eb36bc5e67198c2d2": 3 }` is the value returned, the long value is an id based on your node's public key

Adding a higher-level database for user data

Pieces of music to practice with are great to have, but moving forward you will want to allow users to further express themselves via a username and profile. This will also help prepare you for allowing users to connect to each other in the next chapter.

You will create a new database for users, from which your database of pieces will be referenced. You can create this database in the `ready` event handler of IPFS, alongside where you declared `this.pieces`.

Update your `_init` function to look like this:

```
  async _init() {
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { write: [this.orbitdb.identity.publicKey] }

    const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
    this.pieces = await this.orbitdb.docstore('pieces', docStoreOptions)
    await this.pieces.load()

+   this.user = await this.orbitdb.kvstore("user", this.defaultOptions)
+   await this.user.load()
+   await this.user.set('pieces', this.pieces.id)
  });
```

Then add the following functions in your class:

```
+ async deleteProfileField(key) {
+   const cid = await this.user.del(key)
+   return cid
+ }

+ getAllProfileFields() {
+   return NPP.user.all();
+ }

+ getProfileField(key) {
+   return this.user.get(key)
+ }

+ async updateProfileField(key, value) {
+   const cid = await this.user.set(key, value)
```

```
+   return cid
+ }
```

In your application code, you can use them like this:

```
await NPP.updateProfile("username", "aphelionz")

var profileFields = NPP.getAllProfileFields();A
// { "username": "aphelionz", "pieces": "/orbitdb/zdpu...../pieces" }

await NPP.deleteProfileField("username")
```

We think you're getting the idea.

What just happened?

You created a database to store anything and everything that might pertain to a user, and then linked the `pieces` to that, nested inside.

- `this.orbitdb.kvstore("user", this.defaultOptions)` creates a new OrbitDB of a type that allows you to manage a simple key value store.
- `this.user.set('pieces', this.pieces.id)` is the function that the `kvstore` uses to set items. This is equivalent to something like the shorthand `user = {}; user.pieces = id`
- `NPP.user.all()` returns all keys and values from a `keystore` database
- `this.user.del(key)` deletes the specified key and corresponding value from the store
- `this.user.get(key)` retrieves the specified key and the corresponding value from the store

Dealing with fixture data

Fresh users to the app will need a strong onboarding experience, and you'll enable that for them now by giving people some data to start with, and you'll want this process to work offline.

First, create the `loadFixtureData` function inside the `NewPiecePlease` class:

```
+ async loadFixtureData(fixtureData) {
+   const fixtureKeys = Object.keys(fixtureData)
+   for (let i in fixtureKeys) {
+     let key = fixtureKeys[i]
+     if(!this.user.get(key)) await this.user.set(key, fixtureData[key])
+ }
```

Then, Update your *init* function to call `loadFixtureData` with some starter data:

```
  async _init() {
+   const nodeId = await this.node.id()
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}

    const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
    this.pieces = await this.orbitdb.docstore('pieces', docStoreOptions)
    await this.pieces.load()

    this.user = await this.orbitdb.kvstore("user", this.defaultOptions)
    await this.user.load()

+   await this.loadFixtureData({
```

```
+      "username": Math.floor(Math.rand() * 1000000),
+      "pieces": this.pieces.id,
+      "nodeId": nodeId.id
+    })

    this.onready()
  }
```

Then, if you were to clear all local data and load the app from scratch, you'd see this:

```
var profileFields = NPP.getAllProfileFields()
console.log(profileFields)
```

You would see:

```
{
  "nodeId": "QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM",
  "pieces": "/orbitdb/zdpuArXLduV6myTmAGR4WKv4T7yDDV7KvwkmBaU8faCdrKvw6/pieces",
  "peers": [],
  "username": 304532
}
```

What just happened?

You created simple fixture data and a function to load it into a fresh instantiaton of the app.

Key Takeaways

- The distributed applications of the future will be complex and require data structures to mirror and manage that complexity.
- Luckily, OrbitDB is extremely flexible when it comes to generating complex and linked data structures
- These structures can contain any combination of OrbitDB stores - you are not limited to just one.
- You can nest a database within another, and you can create new databases to next your existing databases within.
- *Nesting* databases is a powerful approach, but it is one of many. **Do not** feel limited. **Do** share novel approaches with the community.
- Fixture data can be loaded easily, and locally, by simply including a basic set of values during app initialization

And with this, you are now ready to connect to the outside world. Continue to Chapter 4: Peer to Peer Part 1 to join your app to the global IPFS network, and to other users!

Chapter 4: Peer-to-Peer, Part 1 (The IPFS Layer)

> There's a lot of ground to cover as we move from offline to fully peer-to-peer, and we need to start where it starts: *connecting to IPFS*, *connecting directly to peers*, and *communicating with them via IPFS pubsub*.

Table of Contents

Please complete Chapter 3 - Structuring Data first.

Connecting to the global IPFS network

You will now reconfigure your IPFS node to connect to the global network and begin swarming with other peers. For more information on what this means, see Part 2 - Thinking Peer to Peer. For now, just understand that it means you're getting connected.

Restoring default IPFS config values

> **Note:** Bootstrap peers are important because... TODO

We purposefully started offline, but now we're going to want to get connected -

This is because bootstrap and swarm values are persisted in your IPFS config. This is located in the filesystem in the case of node.js and in IndexedB in the case of the browser. You should not manually edit these files.

However, nothing will change yet when you run the app. What you *can* do is run the you should see some messages in your console, something like:

Restoring your default bootstrap peers

We started the tutorial offline to focus on OrbitDB's core concepts. Now you'll undo this and connect the app, properly, to the global IPFS network.

Update the `NewPiecePlease` constructor like so:

```
class NewPiecePlease {
  constructor (IPFS, OrbitDB) {
    this.node = new IPFS({
-     preload: { enabled: false }
+     relay: { enabled: true, hop: { enabled: true, active: true } },
      EXPERIMENTAL: { pubsub: true },
      repo: "./ipfs",
-     config: {  Bootstrap: [], Addresses: { Swarm: [] } }
    });
```

Now, you can either delete your data, by deleting the `ipfs` and `orbitdb` folders in node.js, or by clearing your local data in the browser, or you can restore locally.

If you don't want to blow away your data, then you can manually restore your bootstrap peers.

```
await NPP.node.bootstrap.list()
// outputs []
```

To restore the default peers, like the one generated in the previous chapters, run this command *once* to restore your default bootstrap peers.

```
this.node.bootstrap.add(undefined, { default: true })
```

Re-running `bootstrap.list` now gives you a colorful array of bootstrap peers, ready to be connected to.

```
await NPP.node.bootstrap.list()
/* outputs:
'/ip4/104.236.176.52/tcp/4001/ipfs/QmSoLnSGccFuZQJzRadHn95W2CrSFmZuTdDWP8HXaHca9z',
'/ip4/104.131.131.82/tcp/4001/ipfs/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ',
'/ip4/104.236.179.241/tcp/4001/ipfs/QmSoLPppuBtQSGwKDZT2M73ULpjvfd3aZ6ha4oFGL1KrGM',
'/ip4/162.243.248.213/tcp/4001/ipfs/QmSoLueR4xBeUbY9WZ9xGUUxunbKWcrNFTDAadQJmocnWm',
'/ip4/128.199.219.111/tcp/4001/ipfs/QmSoLSafTMBsPKadTEgaXctDQVcqN88CNLHXMkTNwMKPnu',
'/ip4/104.236.76.40/tcp/4001/ipfs/QmSoLV4Bbm51jM9C4gDYZQ9Cy3U6aXMJDAbzgu2fzaDs64',
'/ip4/178.62.158.247/tcp/4001/ipfs/QmSoLer265NRgSp2LA3dPaeykiS1J6DifTC88f5uVQKNAd',
'/ip4/178.62.61.185/tcp/4001/ipfs/QmSoLMeWqB7YGVLJN3pNLQpmmEk35v6wYtsMGLzSr5QBU3',
'/ip4/104.236.151.122/tcp/4001/ipfs/QmSoLju6m7xTh3DuokvT3886QRYqxAzb1kShaanJgW36yx',
'/ip6/2604:a880:1:20::1f9:9001/tcp/4001/ipfs/QmSoLnSGccFuZQJzRadHn95W2CrSFmZuTdDWP8HXaHca9z',
'/ip6/2604:a880:1:20::203:d001/tcp/4001/ipfs/QmSoLPppuBtQSGwKDZT2M73ULpjvfd3aZ6ha4oFGL1KrGM',
'/ip6/2604:a880:0:1010::23:d001/tcp/4001/ipfs/QmSoLueR4xBeUbY9WZ9xGUUxunbKWcrNFTDAadQJmocnWm',
'/ip6/2400:6180:0:d0::151:6001/tcp/4001/ipfs/QmSoLSafTMBsPKadTEgaXctDQVcqN88CNLHXMkTNwMKPnu',
'/ip6/2604:a880:800:10::4a:5001/tcp/4001/ipfs/QmSoLV4Bbm51jM9C4gDYZQ9Cy3U6aXMJDAbzgu2fzaDs64',
'/ip6/2a03:b0c0:0:1010::23:1001/tcp/4001/ipfs/QmSoLer265NRgSp2LA3dPaeykiS1J6DifTC88f5uVQKNAd',
'/ip6/2a03:b0c0:1:d0::e7:1/tcp/4001/ipfs/QmSoLMeWqB7YGVLJN3pNLQpmmEk35v6wYtsMGLzSr5QBU3',
'/ip6/2604:a880:1:20::1d9:6001/tcp/4001/ipfs/QmSoLju6m7xTh3DuokvT3886QRYqxAzb1kShaanJgW36yx',
'/dns4/node0.preload.ipfs.io/tcp/443/wss/ipfs/QmZMxNdpMkewiVZLMRxaNxUeZpDUb34pWjZ1kZvsd16Zic',
'/dns4/node1.preload.ipfs.io/tcp/443/wss/ipfs/Qmbut9Ywz9YEDrz8ySBSgWyJk41Uvm2QJPhwDJzJyGFsD6'
*/
```

Enabling the swarm

Next, you will restore your default swarm addresses. These are adresses that your node announces itself to the world on.

Luckily, in the browser you don't have to do anything, the default is an empty array. You should already see something like this in your console:

```
Swarm listening on /p2p-circuit/ipfs/QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX
Swarm listening on /p2p-circuit/p2p-websocket-star/ipfs/QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX
```

In node.js, run this command:

```
NPP.node.config.set("Addresses.Swarm", ['/ip4/0.0.0.0/tcp/4002', '/ip4/127.0.0.1/tcp/4003/ws'], console.
```

Again, you won't have to do either of these restorations if you're starting with a fresh IPFS repo. These instructions are just included to deepen your understanding of what's going on in the stack.

Restart your app you'll see the console output confirming you're swarming. In node.js you'll see something like:

```
Swarm listening on /p2p-websocket-star/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/127.0.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/172.16.100.191/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/172.17.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/127.0.0.1/tcp/4003/ws/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /p2p-circuit/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /p2p-circuit/p2p-websocket-star/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
```

```
Swarm listening on /p2p-circuit/ip4/127.0.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnE
Swarm listening on /p2p-circuit/ip4/172.16.100.191/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUE
Swarm listening on /p2p-circuit/ip4/172.17.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVr
Swarm listening on /p2p-circuit/ip4/127.0.0.1/tcp/4003/ws/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqA
```

You can get the addresses that your node is publishing on via the following command:

```
const id = await NPP.node.id()
console.log(id.addresses)
```

You'll see a list of addresses your node is publishing on. Expect the browser to have only 2, and node.js to have more. Since we're dealing with both node.js and the browser, we will use the addresses starting with `p2p-circut`.

What just happened?

Before this, you were working offline. Now you're not. You've been connected to the global IPFS network and are ready for peer to-peer connections.

- Removing `preload: { enabled: false }` enables connection to the bottom two nodes from the above bootstrap list.
- Removing `config: {  Bootstrap: [], Addresses: { Swarm: [] } }` will prevent thie storing of empty arrays in your config files for the `Bootstrap` and `Addresses.Swarm` config keys
- `this.node.bootstrap.add(undefined, { default: true })` restores the default list of bootstrap peers, as seen above
- `NPP.node.config.set("Addresses.Swarm", ...` restores the default swarm addresses. You should have run this in node.js only
- `relay: { enabled: true, hop: { enabled: true, active: true } }` sets up a your node as a "circut relay", which means that others will be able to "hop" through your node to connect to your peers, and your node will hop over others to do the same.

We realize we've been spending a lot of time in IPFS config and IPFS commands - it's understandable, since the IPFS features form the backbone of what we're doing with OrbitDB.

> **Note:** If you experience 529 errors from the `preload.ipfs.io` servers in your console, rest assured that there is nothing wrong with your app. Those servers exist to strengthen the network and increase application performance but they are *not* necessary. You can re-insert `preload: { enabled: false }` any time and still remain connected to the global IPFS network

Getting a list of connected peers

First, create the `getIpfsPeers` function inside of the `NewPiecePlease` class.

```
+ async getIpfsPeers() {
+   const peers = await this.node.swarm.peers()
+   return peers
+ }
```

Then, in your application code:

```
const peers = await NPP.getPeers()
console.log(peers.length)
// 8
```

Note that this number will increase over time as your swarm automatically grows.

Connecting to peers

Next, you'll allow your users to connect to other peers via their *multiaddresses*.

> **Note:** There's a number of ways to model and test this during development - you could open up two browsers, or a public and private window in the same browser. Similarly, you could run one instance of the app in node.js and the other in the browser. You should be able to connect to all.

You can now enable peer connection by adding this function to the `NewPiecePlease` class:

```
+ async connectToPeer(multiaddr, protocol = "/p2p-circuit/ipfs") {
+   try {
+     await this.node.swarm.connect(multiaddr)
+   } catch(e) {
+     throw (e)
+   }
+ }
```

Then, update the `_init_` function to include an event handler for when a peer is connected:

```
  async _init() {
    const nodeInfo = await this.node.id()
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}

    const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
    this.pieces = await this.orbitdb.docstore('pieces', docStoreOptions)
    await this.pieces.load()

    this.user = await this.orbitdb.kvstore("user", this.defaultOptions)
    await this.user.load()

    await this.loadFixtureData({
      "username": Math.floor(Math.rand() * 1000000),
      "pieces": this.pieces.id,
      "nodeId": nodeInfo.id
    })

+   this.node.libp2p.on("peer:connect", this.handlePeerConnected.bind(this))

    if(this.onready) this.onready()
  }
```

Finally, create the a simple, yet extensible, `handlePeerConnected` function.

```
+ handlePeerConnected(ipfsPeer) {
+   const ipfsId = ipfsPeer.id._idB58String;
+   if(this.onpeerconnect) this.onpeerconnect(ipfsPeer)
+ }
```

In your application code, implement these functions like so:

```
NPP.onpeerconnect = console.log
await NPP.connectToPeer("/p2p-circuit/ipfs/QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX")
// some time later, outputs "QmZg8h94DAmWozqA8nqKttCFrFeaxRDSazHGCgc8fzkiJS"
```

What just happened?

You created 2 functions: one that shows a list of peers and another that lets you connect to peers via their multiaddress.

- blahblah

Peer to peer communication via IPFS pubsub

The term "pubsub" derived from "publish" and "subscribe", and is a common messaging model in distributed systems. The idea here is that peers will "broadcast" messages to the entire network via a "topic", and other peers can subscribe to those topics and receive all the messages. You can leverage the underlying IPFS pubsub infrastructure to create a simple communication mechanism between the user nodes.

Subscribing to "your" channel

First, you'll add some code to allow you to subscribe to a channel. Channels have names, and in this case you'll just use the node ID.

Update the `ready` handler in the `NewPiecePlease` constructor to look like the following:

```
  async _init() {
    const nodeInfo = await this.node.id()
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}

    const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
    this.pieces = await this.orbitdb.docstore('pieces', docStoreOptions)
    await this.pieces.load()

    this.user = await this.orbitdb.kvstore("user", this.defaultOptions)
    await this.user.load()

    await this.loadFixtureData({
      "username": Math.floor(Math.rand() * 1000000),
      "pieces": this.pieces.id,
      "nodeId": nodeInfo.id
    })

    this.node.libp2p.on("peer:connect", this.handlePeerConnected.bind(this))
+   await this.node.pubsub.subscribe(nodeInfo.id, this.handleMessageReceived.bind(this))

    if(this.onready) this.onready()
  }
```

Then, add the `onmessage` and `sendMessage` function to `NewPiecePlease`

```
handleMessageReceived(msg) {
  if(this.onmessage) this.onmessage(msg)
}
```

Use this in your application:

```
NPP.onmessage = console.log
/* When receiving a message, it will output something like:
{
  "from": "QmVQYfz7Ksimx8a4kqWJinX9BqoiYM5BQVyoCvotVDjj6P",
  "data": "<Buffer 64 61 74 61>",
```

```
  "seqno": "<Buffer 78 3e 6b 8c fd de 5d 7b 27 ab e4 e0 c9 72 4e c0 aa ee 94 20>",
  "topicIDs": [ "QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM" ]
}
*/
```

Sending messages to peers

Next you'll give your users the ability to send messages to each other via those pubsub topics.

```
+ async sendMessage(topic, message, callback) {
+   try {
+     const msgString = JSON.stringify(message)
+     const messageBuffer = this.node.types.Buffer(msgString)
+     await this.node.pubsub.publish(topic, messageBuffer)
+   } catch (e) {
+     throw (e)
+   }
+ }
```

You can then utilize this function in your application code, and your user will see the output as defined above.

```
let data // can be any JSON-serializable value
var hash = "QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM";
var callback = console.error
await NPP.sendMessage(hash, data, callback)
```

What just happened?

> **Note:** These techniques presented for *educational purposes only*, with no consideration as to security or privacy. You should be encrypting and signing messages at the application level. More on this in Chapter 6 of the tutorial.

Key Takeaways

- Resolves #463
- Resolves #468
- Resolves #471
- Resolves #498
- Resolves #519
- Resolves #296
- Resolves #264
- Resolves #460
- Resolves #484
- Resolves #474
- Resolves #505
- Resolves #496

Now, move on to Chapter 05 - Peer to Peer Part 2

Chapter 5: Peer-to-Peer Part 2 (OrbitDB)

TODO: Description

Table of Contents

Please complete Chapter 4 - Peer to Peer first.

Enabling debug logging

There's a lot of moving parts in connecting to a peer's OrbitDB database, and you'll want a deeper look into what's going on as you start to work with connections.

Throughout the OrbitDB / IPFS stack, logging is controlled via a global variable called `LOG` which uses string pattern matching to filter and display logs, e.g. `LOG="*"` will show all logs and be very noisy.

In node.js, you can enable this by passing an environment variable before the invocation of the `node` command:

```
$ LOG="orbit*" node
```

In the browser, you can set this as a global variable on `window`:

```
window.LOG="orbit*"
```

Then, once you re-run the app, you should see a great deal of console info, abridged here:

```
[DEBUG] orbit-db: open()
[DEBUG] orbit-db: Open database '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/pieces'
[DEBUG] orbit-db: Look from './orbitdb'
[DEBUG] cache: load, database: /orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/pieces
[DEBUG] orbit-db: Found database '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/pieces'
[DEBUG] orbit-db: Loading Manifest for '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/piece
[DEBUG] orbit-db: Manifest for '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/pieces':
{
  "name": "pieces",
  "type": "docstore",
  "accessController": "/ipfs/zdpuB1XW983eHNiCcUFEiApGFt1UEbsfqTBQ7YAYnkVNpLiPF"
}
[DEBUG] cache: load, database: /orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/pieces
[DEBUG] orbit-db: Saved manifest to IPFS as 'zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz'
[DEBUG] cache: load, database: /orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz/pieces
```

What just happened?

You enabled debug logging in the app for orbitdb so you can get detailed information about what's going on when you run certain commands.

- `Open database` corresponds to your `this.orbitdb.keyvalue`, `this.orbitdb.docs` calls which are wrappers around `this.orbitdb.open({ type: "keyvalue|docs" })`
- The database `manifest` is a JSON document stored via `ipfs.dag.put` at the address in the database location, `zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y4LCnjvBz` in the above examples. Try using `NPP.node.dag.get()` to explore that content!

- `load` calls then read the database contents into memory and correspond with your `db.load` calls.

Much more information about what's going on internally is provided in Part 3 of this book, OrbitDB Architecture.

Discovering Peer's Databases

To share data between peers, you will need to know their OrbitDB address. Unforutately, simply connecting to a peer is not enough, since there's not a simple way to obtain databases address from a simple IPFS peer-to-peer connection. To remedy this, you'll create a simple flow that exchanges user information via IPFS pubsub, and then use OrbitDB's loading and event system to load and display the data.

In order to provide a proper user experience, you'll want to hide as much of the peer and database discovery as possible by using OrbitDB and IPFS internals to exchange database addresses and load data upon peer connection.

The flow you'll create will be: 1. User manually requests a connection to a user 2. On a successful connection, both peers send messages containing their user information via a database address 3. Peer user databases are loaded, replicated, and inspected for a `userDb` key 4. On a successful discovery, user information is added to our local `companions` database

First, update your `handlePeerConnected` function to call `sendMessage` we introduce a timeout here to give the peers a second or two to breathe once they are connected. You can later tune this, or remove it as you see fit and as future IPFS features provide greater network reliability and performance.

```
  handlePeerConnected(ipfsPeer) {
    const ipfsId = ipfsPeer.id._idB58String;
+   setTimeout(async () => {$
+     await this.sendMessage(ipfsId, { userDb: this.user.id })
+   }, 2000)$
    if(this.onpeerconnect) this.onpeerconnect(ipfsPeer)
  }
```

Then, register and add the `handleMessageReceived` function to the `NewPiecePlease` class

```
async _init() {
  const nodeInfo = await this.node.id()
  this.orbitdb = await OrbitDB.createInstance(this.node)
  this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}

  const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
  this.pieces = await this.orbitdb.docs('pieces', docStoreOptions)
  await this.pieces.load()

  this.user = await this.orbitdb.keyvalue("user", this.defaultOptions)
  await this.user.load()

  await this.loadFixtureData({
    "username": Math.floor(Math.random() * 1000000),
    "pieces": this.pieces.id,
    "nodeId": nodeInfo.id,
  })

  this.node.libp2p.on("peer:connect", this.handlePeerConnected.bind(this))
+ await this.node.pubsub.subscribe(nodeInfo.id, this.handleMessageReceived.bind(this))
```

```
  if(this.onready) this.onready()
}
```

```
+ async handleMessageReceived(msg) {
+   const parsedMsg = JSON.parse(msg.data.toString())
+   const msgKeys = Object.keys(parsedMsg)
+
+   switch(msgKeys[0]) {
+     case "userDb":
+       var peerDb = await this.orbitdb.open(parsedMsg.userDb)
+       peerDb.events.on("replicated", async () => {
+         if(peerDb.get("pieces")) {
+           this.ondbdiscovered && this.ondbdiscovered(peerDb)
+         }
+       })
+       break;
+     default:
+       break;
+   }
+
+   if(this.onmessage) this.onmessage(msg)
+ }
```

In your application code you can use this functionality like so:

```
// Connect to a peer that you know has a New Piece, Please! user database
await NPP.connectToPeer("Qm.....")

NPP.ondbdiscovered = (db) => console.log(db.all())
/* outputs:
{
  "nodeId": "QmNdQgScpUFV19PxvUQ7mtibtmce8MYQkmN7PZ37HApprS",
  "pieces": "/orbitdb/zdpuAppq7gD2XwmfxWZ3MzeucEKiMYonRUXVwSE76CLQ1LDxn/pieces",
  "username": 875271
}
*/
```

What just happened?

You updated your code to send a message to connected peers after 2 seconds, and then registered a handler function for this message that connects to and replicates another user's database.

- `this.sendMessage(ipfsId, { userDb: this.user.id })` utilizes the function you created previously to send a message to a peer via a topic named from their IPFS id
- `this.node.pubsub.subscribe` registers an event handler that calls `this.handleMessageReceived`
- `peerDb.events.on("replicated" ...` fires when the database has been loaded and the data has been retrieved from IPFS and is stored locally. It means, simply, that you have the data and it is ready to be used.

  Note: If you're a security-minded person, this is probably giving you a panic attack. That's ok, these methods are for educational purposes only and are meant to enhance your understanding of how a system like this works. We will cover authorization and authentication in the next chapter.

Connecting automatically to peers with discovered databases

Peer discovery is great, but your users are going to want those peers to stick around so you can continue to use their data and receive new data as those peers add pieces. You'll make a couple minor modifications the above functions to enable that now. Also, peers is so technical sounding! Musicians might prefer something like "companions" instead.

First, update your `_init` function to make a new "companions" database:

```
async _init() {
  const nodeInfo = await this.node.id()
  this.orbitdb = await OrbitDB.createInstance(this.node)
  this.defaultOptions = { accessController: { write: [this.orbitdb.identity.publicKey] }}

  const docStoreOptions = Object.assign(this.defaultOptions, { indexBy: 'hash' })
  this.pieces = await this.orbitdb.docs('pieces', docStoreOptions)
  await this.pieces.load()

  this.user = await this.orbitdb.keyvalue("user", this.defaultOptions)
  await this.user.load()

+ this.companions = await this.orbitdb.keyvalue("companions", this.defaultOptions)
+ await this.companions.load()

  await this.loadFixtureData({
    "username": Math.floor(Math.random() * 1000000),
    "pieces": this.pieces.id,
    "nodeId": nodeInfo.id,
  })

  this.node.libp2p.on("peer:connect", this.handlePeerConnected.bind(this))
  await this.node.pubsub.subscribe(nodeInfo.id, this.handleMessageReceived.bind(this))

+ this.companionConnectionInterval = setInterval(this.connectToCompanions.bind(this), 10000)
+ this.connectToCompanions()

  if(this.onready) this.onready()
}
```

Next, create a `getCompanions()` abstraction for your application layer

```
+ getCompanions() {
+   return this.companions.all()
+ }
```

Then, update your `handleMessageReceived` function to add a discovered peer's user database to the `companions` register:

```
  async handleMessageReceived(msg) {
    const parsedMsg = JSON.parse(msg.data.toString())
    const msgKeys = Object.keys(parsedMsg)

    switch(msgKeys[0]) {
      case "userDb":
        var peerDb = await this.orbitdb.open(parsedMsg.userDb)
        peerDb.events.on("replicated", async () => {
          if(peerDb.get("pieces")) {
```

```
+           await this.companions.set(peerDb.id, peerDb.all())
            this.ondbdiscovered && this.ondbdiscovered(peerDb)
          }
        })
        break;
      default:
        break;
    }

    if(this.onmessage) this.onmessage(msg)
  }
```

Finally, create the `connectToCompanions` function:

```
+ async connectToCompanions() {
+   var companionIds = Object.values(this.companions.all()).map(companion => companion.nodeId)
+   var connectedPeerIds = await this.getIpfsPeers()$
+   companionIds.map(async (companionId) => {
+     if (connectedPeerIds.indexOf(companionId) !== -1) return
+     try {
+       await this.connectToPeer(companionId)
+       this.oncompaniononline && this.oncompaniononline()
+     } catch (e) {
+       this.oncompanionnotfound && this.oncompanionnotfound()
+     }
+   })
+ }
```

In your application layer, you can test this functionality like so:

```
NPP.oncompaniononline = console.log
NPP.oncompanionnotfound = () => { throw(e) }
```

What just happened?

You created yet another database for your user's musical companions, and updated this database upon database discovery. You can use this to create "online indicators" for all companions in your UI layer.

- `await this.orbitdb.keyvalue("companions", this.defaultOptions)` creates a new keyvalue store called "companions"
- `this.companions.all()` retrieves the full list of key/value pairs from the database
- `this.companions.set(peerDb.id, peerDb.all())` adds a record to the companions database, with the database ID as the key, and the data as the value stored. Note that you can do nested keys and values inside a `keyvalue` store
- The `companionIds.map` call will then call `this.connectToPeer(companionId)` in parallel for all registered companions in your database. If they are found `oncompaniononline` will fire. If not, `oncompanionnotfound` will fire next.

Simple distributed queries

This may be the moment you've been waiting for - now you'll perform a simple parallel distributed query on across multiple peers, pooling all pleces together into one result.

Create the following function, which combines much of the code you've written and knowledge you've obtained so far:

```
+ async queryCatalog() {
+   const peerIndex = NPP.companions.all()
+   const dbAddrs = Object.keys(peerIndex).map(key => peerIndex[key].pieces)
+
+   const allPieces = await Promise.all(dbAddrs.map(async (addr) => {
+     const db = await this.orbitdb.open(addr)
+     await db.load()
+
+     return db.get('')
+   }))
+
+   return allPieces.reduce((flatPieces, pieces) => {
+     pieces.forEach(p => flatPieces.push(p))
+     return flatPieces
+   }, this.pieces.get(''))
+ }
```

You can now test this by creating a few different instances of the app (try both browser and node.js instances), connecting them via their peer IDs, discovering their databases, and running `NPP.queryCatalog()`.

What just happened?

You performed your first distributed query using OrbitDB. We hope that by now the power of such a simple system, under 200 lines of code so far, can be used to create distributed applications.

- `NPP.companions.all()` will return the current list of discovered companions
- `this.orbitdb.open(addr)` will open the peer's database and `db.load` will load it into memory
- `allPieces.reduce` will take an array of arrays and squash it into a flat array

For now it will return *all* pieces, but for bonus points you can try incorporating the `docstore.query` function instaed of `docstore.get('')`.

Key takeaways

TODO

You're not done yet! Chapter 6 to learn about how you can vastly extend the identity and access control capabilities of OrbitDB

Chapter 6: Identity and Permissions

> **Note:** Please complete Chapter 5 - Replication first.

Access Control

Identity Management

Security Disclosures

- Resolves: #397
- Resolves: #222
- Resolves: #327
- Resolves: #357
- Resolves: #475
- Resolves: #380
- Resolves: #458
- Resolves: #467

Conclusion

The end!

Conclusion

Part 2: Thinking Peer to Peer

Introduction

Part 3: The Architecture of OrbitDB

Introduction

Part 4: What Next?

Introduction

Appendix 1: CRDTs