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.

This work is work is copyrighted by Haja Networks Oy, under a CC-BY-NC 4.0 Unported International License.

Contents

Introduction

Centralized Power: The Present

The last few years have been a watershed for how we collectively understand the internet, data, and our increasingly interconnected world. For instance, here are a few major events that have changed our perceptions of security, privacy, and access:

- Data stores in centralized locations has been breached time and time again in the form of government and criminal hacking, exposing the data of millions of people per attack. TODO Link all of these with real-world examples
- The Internet-of-Things has linked your behavioral and usage data directly to the headquarters of large corporations; corporations, like Amazon, Apple, and Microsoft, that are so large that they are able to directly influence politics on the global scale, and are unlike anything that our society has before. This linking has in turn lead to personalization, manipulation, and exploitation to an alarming degree.
- Artists, musicians, performers, and writers alike are now able to use platforms like Spotify and Amazon to publish their work to a larger audience, but the devaluation of art and music is dire, leaving struggling artisans to compete with each other for fractions of what they made even 10 years ago.
- Historical data is now being mined and repurposed for algorithms, leading to practices such as real estate pricing and predictive policing. Only very recently has any attention been paid to the bias inherent in these data sets.
- The trend of tracking user hits has lead to many large corporations monetizing **attention** by way of harvesting and processing user data. This, when done both by major media companies (Fox News, MSNBC) and social web sites (Facebook) has wreaked its fair share of havoc in terms of society and politics.

These individual points are connected. The central problem in all these cases comes from a simple asymmetry: the people that *generate* the data do not end up *owning* the data. This can (and has, and will) be used against you: in a court of law, in your bank account balance, in your social structure, and in your digital and physical life.

How did we get here? A Historical Context

Distributed systems are nothing new, yet somehow advances in this space tend to "disrupt" more thoroughly than any VC-funded startup can ever hope to, with results that are often as tragic as they are profound.

The Birth of the Internet

In the late 1960s, The Internet itself was developed as a decentralized system of computers capable of sustaining a nuclear attack. Individual computers became linked together via underground wire and a process known as "packet switching" was implemented to send data in disparate chunks which were expected to be re-assembled in order on the receiving end.

By the late 1980s, academic works involving distributed consensus algorithms such as Paxos were introduced. TODO: flesh out

in 1991, the first World Wide Website went live, and what is now the most popular content and user interface layer for the internet was born. The web allowed anybody with the means to publish a website and link to other sites using their location-based addresses.

Already you can see we have a distributed infrastructure, the seeds of distributed computation, and a hyperlinked data structure. This alone

Early Peer-to-Peer Applications

TODO: Kleppman, etc

TODO: 1999-2004 Napster, Kazaa, etc.

The Gig Economy

Companies now use decentralized thiBy providing a centralized platform to manage a decentralized set of assets

Enter the Blockchain

The publication of the Bitcoin whitepaper in October 2008, started a new renaissance in our industry. Not only did it prove out a new technology in the blockchain, but it began a new way of thinking about distributed systems in general. In addition to giving rise to a new generation of financial applications with a high degree of speculative activity, Bitcoin's underlying (and beyond computationally expensive) *proof-of-work* algorithm gave us distributed consensus.

Thegave the industry a new focus and a new way of thinking about a critical finance. The fact that it was published by a mysterious figure, the so-called Satoshi Nakamoto, only served to mystify technologists and

The Way Out: Our Distributed Future

TODO: IPFS

We would argue that cryptocurrency and blockchain technologies only laid the foundation of our distributed future. Projects like IPFS, which harness the power of directed acyclic graphs. TODO: Filecoin

What is OrbitDB?

If *IPFS* is the distributed "hard drive" where all data are content-addressed and retrievable from a peer-to-peer swarm, then *OrbitDB* is the database engine. OrbitDB creates and manages mutable databases and provides an extremely simple interface, typically centered around simple `get` and `set` functions, to manage a large deal of complexity to store these databases in a distributed fashion on IPFS.

OrbitDB achieves this by building structures called *Conflict-free Replicated Data Types*, or CRDTs. CRDTs are essentially logs with specific "clock" values that allow multiple users to perform independent and asychonous operations on the same database. When the peers share these logs with each other, the clock values ensure that there is no ambiguity about how their disparate entries will be put back together.

What does this mean in practical terms, and what should you as the developer understand about OrbitDB?

OrbitDB is written in JavaScript

OrbitDB is packaged as a Node.js library, available here. JavaScript was chosen because of it's popularity and ubiquity in the programming community, and for its interpoerability with the JavaScript implementation of IPFS, called *js-ipfs*.

Work is currently underway to allow interoperability between multiple programming languages via a common HTTP API.

OrbitDB is free (as in freedom) software

OrbitDB is released under the MIT software license, which is an exceedingly permissible license. Organizations and individual develoers are free to fork, embed, modify, and contribute the code with no obligations. Of course, this also means there are no warrantys as well.

OrbitDB is a tool for you, the developer, to build distributed applications

Outside of a rereference implementation called Orbit Chat, the OrbitDB team primiarily focuses on making OrbitDB more robust, reliable, and performant. Our mission is to enable you to build distributed applications that will break into the mainstream and allows your users true soverignty over their data.

What can I use OrbitDB for?

We want to believe that any application that can be built using traditional models can be buiilt with a peer-to-peer architecture. Perhaps view this as a challenge.

OrbitDB excels in flexibly building complex distributed appplications.

OrbitDB is NOT a Blockchain

OrbitDB operates on the model of *eventual consistency* meaning that operations can be taking place at places and times that you are unaware of, with the assumption that you'll eventually sync up. Each time you interact with the database, you are interactive with a snapshot in time.

This contrasts with Blockchain's idea of *strong consistency* where entries are added to the database only after they have been verified by a distributed consensus algorithm.

Who is already using OrbitDB?

Perhaps the best way to answer your question of "What can I build with OrbitDB?" is by example. There are already several organizations using OrbitDB in the wild.

If you're looking for ideas for your own app, maybe start here!

Project Name	Description
Colony	Decentralized organizations
uPort / ConsenSys 3Box	A distributed database built using OrbitDB
Origin Protocol	Origin Messaging A messaging protocol built using OrbitDB
ChluNetwork	Chlu
Cluster Labs	IpfsCloud File storage / sharing

Project Name	Description
TallyLab	Data capture / recording app, "tallying"
Computes (Magic Leap)	Computes Decentralized Naming Service
DASH	Dash Evolution
Blockstack	Graphite Docs decentralized and encrypted replacement for Google G-Suite and Microsoft Office
Protocol Labs	Discussify Discussion platform

A Warning

OrbitDB, and the underlying IPFS layer, are currently *alpha* software. There have already been high-severity bugs, API-breaking changes, and security disclosures ahead. This is transparently communicated, yet these warnings have done nothing at all to stop large organizations from using these tools in production. It's already out in the wild.

If that sounds bad, don't worry - it gets worse.

Not only are these tools alpha, but the **entire decentralized industry is in an alpha state**. There are unsolved problems not just technical in nature but legal, political, economic, and societal. The wave of the personal computer revolution crashed, followed by the information age. The wave of the mobile revolution left us in the social age. Now we're here in the midst of a new decentralized revolution.

You may hear claims of what type of age might follow this, but it's imposisble to tell. Whatever it may be, it's probably safe to say that the world will not be ready for it.So, brace yourself. It is our hope that you find this challenging and exciting, but don't be surprised by feeling a healthy dose of trepidation, and even downright horror at times. This is how it's going to be for a while.

That all being said: welcome to the distributed future, and we're excited to have you with us.

Part 1: The OrbitDB Tutorial

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

Introduction

In order to maximize accessibility this tutorial does not favor either Node.js or the browser, since the same OrbitDB code runs on both. By following this tutorial step-by-step, your goal will be to build a *library* in the form of a JavaScript class that empowers a UI developer (CLI or Web) to build out a fully-realized application.

Requirements

- A computer with 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?

In the following chapters you will learn the following topics.

1. Laying the Foundation covers the installation of IPFS and OrbitDB and the basics of database creation.
2. Managing Data introduces OrbitDB data stores and walks you through basic create, update, and delete methods of OrbitDB.
3. Structuring Data suggests some ways to structure multiple OrbitDBs into a more robust schema.
4. Peer-to-peer, Part 1 (IPFS) begins a large discussion of peer-to-peer networking and messaging, starting with the IPFS layer.
5. Peer-to-peer, Part 2 (OrbitDB) adds OrbitDB to the mix through database discovery, connection, and replication.
6. Identity and Permissions hardens the library via encryption and distributed identity.

This tutorial is a work in progress, and individuals should feel encouraged to submit pull requests and provide feedback.

What will I build?

If you are a musician, you probably need some 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 - forming duets, trios, quartets, septets, all the way up to orchestras.

These self-organizing clusters of musicians will always need better ways 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 a JavaScript class that enables such an 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 distributed 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 we believe that **music** is a uniquely universal cultural feature - something that all humans 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. This isn't a naïve overstatement; it is really possible that what you make here will functionally be usable immediately, as a solid MVP, by actual musicians - and we encourage you to let us know if you know anyone who does end up using it! Some MVPs are make-and-forget; this one will stand on its own legs.

Admittedly, a music app is somewhat arbitrary. But if you're here to learn more about OrbitDB, you'll get what you came for.

Conventions

- This tutorial is:
    - *Imperative*, meaning that it contains instructions you will follow
    - *Interactive*, meaning you are meant to code as you read
    - *Isomorphic*, meaning that it will work in the browser or command line, whichever you choose
- Read this tutorial in order, the learning builds on itself over time.
- The UI Layer for this library is *suggested*, instead of built directly. The tutorial focuses on the building of a single JavaScript class which encapsulates all the functionality needed to build the UI layer.

- Type the examples in, do not copy and paste.
- **What just happened?** sections are interspersed to 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 like.
- Steps that **you** should complete are represented and highlighted 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. • *Italicized* words are words which we think you should learn to become familiar with. • **Bolded** words are merely for textual emphasis. We want to make sure you don't miss something! • Some typographical errors and misspellings - Javascript for JavaScript, decentralised instead of decentralized, and so on - will sneak in. If you see something, open a PR with a change! Check out our GitHub repository that this document is hosted in. All contributors will be credited.

Ready? Start with Chapter 1: Laying the Foundation

Chapter 1 - Laying the Foundation

> The basics of OrbitDB include installing OrbitDB (and IPFS), setting up an isomorphic project that runs in both Node.js and the browser, creating databases, and understanding how to choose datastores.

Table of Contents

Please see the Introduction before beginning this chapter.

Installing the requirements: IPFS and OrbitDB

Before you start working on the core functionality behind the peer-to-peer sharing of sheet music, you'll need to get the foundations of our distributed database set up.

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.

> **Note**: 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 in Part 3.

In Node.js

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

```
$ npm init --yes
$ npm install orbit-db ipfs
```

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

> **Note:** If you're running on a Windows prompt, or if you don't have certain build tools like `g++` and `python` installed, you may see a noisy console output with lots of warnings and errors. Keep going, your code should still run.

> **Note:** Adding the `--yes` flag will automatically use your npm defaults. You can go through and edit the package.json later, but it's not entirely necessary for this part of the tutorial.

In the Browser

If you're using the browser for this tutorial, we recommend using unpkg for obtaining pre-built, minified versions of both IPFS and OrbitDB. Simply include these in your HTML:

```html
<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.

Creating the isomorphic bookends

Since OrbitDB works in the browser and Node.js, you're going to want to make the library as *isomorphic* as possible. This means we want the same code to run in the browser as runs in REPL or local environment. 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)
+ }
```

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:

```
$ node

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

Not much should happen either way, since there's not much code there yet. For now just make sure you can create the NPP constant.

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

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

Instantiating IPFS and OrbitDB

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.

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

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

```
+          }
+      });
+
+      this.node.on("error", (e) => { throw (e) })
+      this.node.on("ready", this._init.bind(this))
+   }
+
+   async _init() {
+      this.orbitdb = await this.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 QmPSicLtjhsVifwJft-
nxncFs4EwYTBEjKUzWweh1nAA87B. This is the identifier of your IPFS node.

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.
- `EXPERIMENTAL: { pubsub: true }` enables IPFS pubsub, which is a method of com-
  municating 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 if something happens during the creation of the IPFS node.
- `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
```

Looking inside 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 needs to operate. See Part 3 for detailed information about this.

The `ipfs/` folder contains all of your IPFS data. Explaining this in depth is outside of the scope of this tutorial, but 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
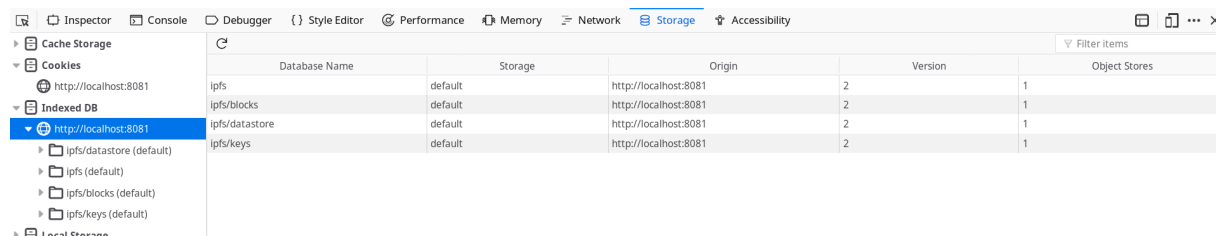
Figure 1: An image showing the IPFS IndexedDB databases in Firefox

Note since you have not explicitly defined a database in the browser, 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

Your users will want to create a catalog of musical pieces to practice. You will now create this database, and ensure that *only that user* can write to it.

Expand of your `_init` function to the following:

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

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/zdpuB3VvBJHqYCocN4utQrpBseHou88mq2D This is the id, or **address** (technically a multiaddress) of this database. It is important for you to not only *know* this, but also to understand what it is. This string is composed of 3 parts, separated by /s:

1. The first bit, `/orbitdb.`, is the protocol. It tells you that this address is an OrbitDB address.
2. The second, or middle, part `zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3` that is the most interesting. This is the Content ID (CID) of the database manifest, which contains:

   - The **access control list** of the database
   - The **type** of the database
   - The **name** of the database

3. The final part is the name you provided, in this case `pieces`, which becomes the final part of the multiaddress

Note: Addresses that start with Qm… are typically CIDv0 content addresses, while addresses that start with zdpu…. are CIDv1. Misunderstanding OrbitDB addresses can lead to some very unexpected - sometimes hilarious, sometimes disastrous outcomes.

What just happened?

Your code created a local OrbitDB database, of type "docstore", writable only by the user who created it.

- `defaultOptions` and `docStoreOptions` define the parameters 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 the 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."

What else happened in Node.js?

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

```
$ ls orbitdb/
QmNrPunxswb2Chmv295GeCvK9FDusWaTr1ZrYhvWV9AtGM/   zdpuB3VvBJHqYCocN4utQrpBseHou88
```

```
$ ls orbitdb/zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/
pieces/
```

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

You do not 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.

Database Name	Storage	Origin
ipfs	default	http://localhost:8081
ipfs/blocks	default	http://localhost:8081
ipfs/datastore	default	http://localhost:8081
ipfs/keys	default	http://localhost:8081
level-js-orbitdb/zdpuAyitjCoPmK8KdhDw1sxAmi8PnNCe9XXWjWeTFjXJdMM7g/pieces	default	http://localhost:8081
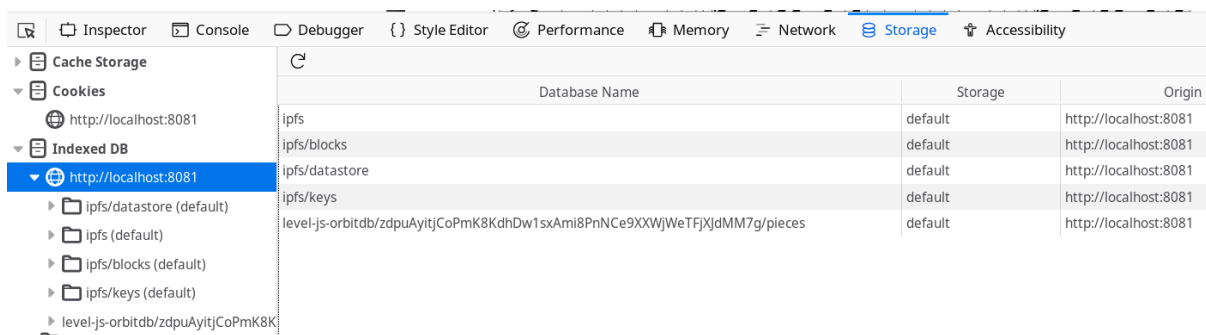
Figure 2: An image showing the IPFS and OrbitDB IndexedDB databases in Firefox

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

Choosing a datastore

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.

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

You have the following choices:

Name	Description
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".
keyvalu	A simple key-value database that supports any JSON-serializable data, even nested objects.

Name	Description
docs	A document database that stores JSON documents which can be indexed by a specified key. Useful for building search indices or version controlling documents and data.
counter	An increment-only integer counter useful for counting events separate from log/feed data.

Also, OrbitDB developers 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 underlying 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 have laid the groundwork, you will 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

The first thing your users will want is to make sure that when they load the app, their data is available. You will do so easily by loading the database contents into memory.

Update your `NewPiecePlease class` handler, adding **one line** at the bottom of the IPFS `ready` handler:

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

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

What just happened?

After you instantiated the database you loaded its contents into memory for use. It is 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. The `load` function retrieves all of the values via their *content addresses* and loads the content into memory.

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

Adding data

Next, your users will want to be able to add sheet music to their catalog. You will use functions exposed from OrbitDB's `keyvalue` store now.

Add a function called `addNewPiece` function now:

```
+ 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
+ }
```

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.

In your application code, Node.js or browser, you can use this function like so, utilizing the default 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'is overwhelming but it will make sense after we explain what happened. For more information see Part 3, The Architecture of OrbitDB.

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

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

What just happened?

You wrote and tested a function that allows users to add new sheet music to the database.

- `piecesDb.put({ ... })` is the most important line here. This call takes an object to store and returns a *multihash*, which is the hash of the content added to IPFS.
- `node.dag.get(hash)` is a function that takes a 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 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 closely related to blockchain.

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

Of course, your users will want to read their data after creating it, so you will enable that functionality now. 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 write some simple `get*` functions like so. `docstore` also provides the more powerful `query` function, which we can abstract to write a `getPieces-ByInstrument` function:

Fill in the following functions now:

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

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

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

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

```
pieces = NPP.getAllPieces()
pieces.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 === in-strument)` queries the database, returning. It's most analogous 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

Next, you will want to provide your users with the ability to update and delete their pieces. For example, if you realize you would 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") {
+   const piece = await this.getPieceByHash(hash)
+   piece.instrument = instrument
+   const cid = await this.piecesDb.put(piece)
+   return cid
+ }
```

```
+ 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 is going on.

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

const cid = await NPP.deletePieceByHash("QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPM
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 us step through the code so we can get to that.

- `this.piecesDb.put` is nothing new, we are 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

Your users will probably not want to mess with content hashes, so you will want to provide them the ability to add files directly to IPFS. This section shows how you will be able to do this, and then store the *address* of the file in OrbitDB.

The overall pattern is:

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 insta

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 a local database available in memory
- 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 will 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("QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPS
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.f
  "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.publicKe }}` should be recognizable from Chapter 1. This sets options for the db, namely the `access-Controller` 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` 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 haven't seen yet - remember, you can create stores with any operations you want. More on this in Part 3.
- `"counters": { "042985dafe18ba45c7f1a57db.........02ae4b5e4aa3eb36bc5e67198c2 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 a profile. You will create a new database for users, from which your database of pieces will be referenced. This will also help prepare you for allowing users to connect to each other in the next chapter.

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 = {
    ...defaultOptions,
    indexBy: 'hash',
  }
  this.piecesDb = 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 will enable that for them now by giving people some data to start with, and you will 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 peerInfo = await this.node.id()
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.pu

    const docStoreOptions = {
      ...defaultOptions,
      indexBy: 'hash',
    }
    this.piecesDb = 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": peerInfo.id
+    })

    this.onready()
  }
```

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

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

You would see:

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

What just happened?

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

- `for (let i in fixtureKeys)` - this type of for loop is used to ensure that the writes happen serially, one after another.
- `await this.user.set(key, fixtureData[key])` sets the user profile key to the fixture value, if the key does not exist
- `await this.node.id()` is a slight misnomer, as it provides a more generalized `peerInfo` object.
- `peerInfo.id` contains the ID string you want, the base58 hash of the IPFS id.

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 nest 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

Your users can manage their local sheet music library, but music is a social, connected venture. The app should reflect that! You will now reconfigure your IPFS node to connect to the global network and begin swarming with other peers. This tutorial started offline to focus on OrbitDB's core concepts, and now you will undo this and connect the app, properly, to the global IPFS network.

To do so, 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 mind doing this, you can skip ahead to Getting a list of connected peers.

Restoring default IPFS config values

If you don't want to blow away your data, then you can manually restore the default values by running a few commands on the application layer.

Restoring your default bootstrap peers

Bootstrap peers are peers that your node connects to automatically when it starts. IPFS supplies this list by default and is comprised of a decentralized set of public IPFS servers.

However, since we purposefully started with an empty list of bootstrap peers and they won't be restored by simply removing the config values. 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 IndexedDB in the case of the browser. You should not manually edit these files.

However, nothing will change yet when you run the app. To see this, run this command in a running application:

```
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/QmSoLnSGccFuZQJzRadHn95W2CrSFmZuTdDWP8HXaHca9
'/ip4/104.131.131.82/tcp/4001/ipfs/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvu
'/ip4/104.236.179.241/tcp/4001/ipfs/QmSoLPppuBtQSGwKDZT2M73ULpjvfd3aZ6ha4oFGL1Kr
'/ip4/162.243.248.213/tcp/4001/ipfs/QmSoLueR4xBeUbY9WZ9xGUUxunbKWcrNFTDAadQJmocn
'/ip4/128.199.219.111/tcp/4001/ipfs/QmSoLSafTMBsPKadTEgaXctDQVcqN88CNLHXMkTNwMKP
'/ip4/104.236.76.40/tcp/4001/ipfs/QmSoLV4Bbm51jM9C4gDYZQ9Cy3U6aXMJDAbzgu2fzaDs64
'/ip4/178.62.158.247/tcp/4001/ipfs/QmSoLer265NRgSp2LA3dPaeykiS1J6DifTC88f5uVQKNA
'/ip4/178.62.61.185/tcp/4001/ipfs/QmSoLMeWqB7YGVLJN3pNLQpmmEk35v6wYtsMGLzSr5QBU3
'/ip4/104.236.151.122/tcp/4001/ipfs/QmSoLju6m7xTh3DuokvT3886QRYqxAzb1kShaanJgW36
'/ip6/2604:a880:1:20::1f9:9001/tcp/4001/ipfs/QmSoLnSGccFuZQJzRadHn95W2CrSFmZuTdD
'/ip6/2604:a880:1:20::203:d001/tcp/4001/ipfs/QmSoLPppuBtQSGwKDZT2M73ULpjvfd3aZ6h
'/ip6/2604:a880:0:1010::23:d001/tcp/4001/ipfs/QmSoLueR4xBeUbY9WZ9xGUUxunbKWcrNFT
'/ip6/2400:6180:0:d0::151:6001/tcp/4001/ipfs/QmSoLSafTMBsPKadTEgaXctDQVcqN88CNLH
```

```
'/ip6/2604:a880:800:10::4a:5001/tcp/4001/ipfs/QmSoLV4Bbm51jM9C4gDYZQ9Cy3U6aXMJDA
'/ip6/2a03:b0c0:0:1010::23:1001/tcp/4001/ipfs/QmSoLer265NRgSp2LA3dPaeykiS1J6DifT
'/ip6/2a03:b0c0:1:d0::e7:1/tcp/4001/ipfs/QmSoLMeWqB7YGVLJN3pNLQpmmEk35v6wYtsMGLz
'/ip6/2604:a880:1:20::1d9:6001/tcp/4001/ipfs/QmSoLju6m7xTh3DuokvT3886QRYqxAzb1kS
'/dns4/node0.preload.ipfs.io/tcp/443/wss/ipfs/QmZMxNdpMkewiVZLMRxaNxUeZpDUb34pWj
'/dns4/node1.preload.ipfs.io/tcp/443/wss/ipfs/Qmbut9Ywz9YEDrz8ySBSgWyJk41Uvm2QJP
*/
```

Enabling the swarm

Next, you will restore your default swarm addresses. These are addresses 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/QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz
Swarm listening on /p2p-circuit/p2p-websocket-star/ipfs/QmWxWkrCcgNBG2uf1HSVAwb9
```

In Node.js, run this command:

```
NPP.node.config.set("Addresses.Swarm", ['/ip4/0.0.0.0/tcp/4002', '/ip4/127.0.0.1/t
```

After restarting your app you will see the console output confirming you're swarming. In Node.js you will see something like:

```
Swarm listening on /p2p-websocket-star/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWL
Swarm listening on /ip4/127.0.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYR
Swarm listening on /ip4/172.16.100.191/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z
Swarm listening on /ip4/172.17.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSY
Swarm listening on /ip4/127.0.0.1/tcp/4003/ws/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5j
Swarm listening on /p2p-circuit/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVn
Swarm listening on /p2p-circuit/p2p-websocket-star/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3
Swarm listening on /p2p-circuit/ip4/127.0.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zS
Swarm listening on /p2p-circuit/ip4/172.16.100.191/tcp/4002/ipfs/QmXG8yk8UJjMT6q
Swarm listening on /p2p-circuit/ip4/172.17.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2z
Swarm listening on /p2p-circuit/ip4/127.0.0.1/tcp/4003/ws/ipfs/QmXG8yk8UJjMT6qtE
```

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 will 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'll focus on the addresses starting with `p2p-circuit`.

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 the 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 "circuit 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.

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. 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

Your users will want to know which users they are connected to or at the very least how many peers. You will enable that using a simple IPFS function.

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, so check and update periodically.

Manually connecting to peers

All users will be running their own IPFS nodes either in the browser or on Node.js. They'll want to connect together, so you will now allow your users to connect to other peers via their IPFS ids.

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 any of them.

Create the connectToPeer function inside the NewPiecePlease class:

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

    const docStoreOptions = {
      ...defaultOptions,
```

```
    indexBy: 'hash',
  }
  this.piecesDb = 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(ipfsId)
+ }
```

In your application code, implement these functions like so:

```
NPP.onpeerconnect = console.log
await NPP.connectToPeer("QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX")
// some time later, outputs "QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX"
```

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.

- `this.node.swarm.peers()` returns a an array of connected peers
- `protocol = "/p2p-circuit/ipfs/"` is used as a default since this is a common protocol between browser and Node.js

- `this.node.swarm.connect(protocol + multiaddr)` connects to a peer via their a multiaddress that combines `protocol` and `multiaddr` into an address like `p2p-circuit/ipfs/Qm....`
- `this.node.libp2p.on("peer:connect", this.handlePeerConnected.bind(this))` registers the `handlePeerConnected` function to be called whenever a peer connects to the application node.
- `ipfsPeer.id._idB58String` is a nice, synchronous way to get the peer id.

Peer to peer communication via IPFS pubsub

The term "pubsub" derived from "**pub**lish and **sub**scribe, 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 this mechanism to create a simple communication mechanism between the user nodes.

Subscribing to "your" channel

Your users will need to be able to receive messages that are broadcast to them. You will enable this by having them subscribe to a topic named after their IPFS id.

Update the `_init` function 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.pu

  const docStoreOptions = {
    ...defaultOptions,
    indexBy: 'hash',
  }
  this.piecesDb = 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.bin

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

Then, add the `handleMessageReceived` 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" ]
}
*/
```

Messages are sent with the data as type `Buffer` and can contain any JSON-serializable data, and can be decoded using the `msg.data.toString()` function.

Sending messages to peers

Now that the nodes are listening, they'll want to send messages to each other. You will enable this to give your users the ability to send messages to each other via the pubsub topics of their IPFS ids.

Create the `sendMessage` function inside the `NewPiecePlease` class:

```
+ 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 in the previous subsection.

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

What just happened?

You enabled a simple message sending and receiving system which allows peer-to-peer communication.

- `this.node.types.Buffer(msgString)` encoded the message, after JSON stringification, into a Buffer object.
- `this.node.pubsub.publish(topic, messageBuffer)` sends the encoded Buffer to the topic specified. In our case, this will be the IPFS id

Note: These techniques are 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

- In order to do anything peer-to-peer, you will need to be connected to the global IPFS network
- If you started offline, the default configuration is restorable
- Your node announces itself to the network via its *swarm* addresses
- Your node will connect to *bootstrap* peers automatically, and you can manually connect to peers on command
- Peer-to-peer communication is achieved through a "pubsub" model
- In pubsub, users subscribe to, and broadcast messages to, "topics"

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

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

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

> OrbitDB utilizes IPFS's underlying peer-to-peer layer to share data between peers. In this chapter you will learn methods for *discovering peers*, *connecting automatically to known peers*, and making *distributed queries*.

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 will 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/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH
[DEBUG] orbit-db: Look from './orbitdb'
[DEBUG] cache: load, database: /orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y
[DEBUG] orbit-db: Found database '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipT
[DEBUG] orbit-db: Loading Manifest for '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKF
[DEBUG] orbit-db: Manifest for '/orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7
```

```
{
  "name": "pieces",
  "type": "docstore",
  "accessController": "/ipfs/zdpuB1XW983eHNiCcUFEiApGFt1UEbsfqTBQ7YAYnkVNpLiPF"
}
[DEBUG] cache: load, database: /orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y
[DEBUG] orbit-db: Saved manifest to IPFS as 'zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfi
[DEBUG] cache: load, database: /orbitdb/zdpuAz77YioswjyfnnqVDhjycEn4BKFhvxfipTH7y
```

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. Unfortunately, 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 will 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 will 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 will 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.publ

  const docStoreOptions = {
    ...defaultOptions,
    indexBy: 'hash',
  }
  this.piecesDb = await this.orbitdb.docstore('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(

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

Finally, create the `handleMessageReceived` function:

```
+ 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 anxiety. 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 will 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.publ

  const docStoreOptions = {
    ...defaultOptions,
    indexBy: 'hash',
  }
  this.piecesDb = await this.orbitdb.docstore('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.defaultOption
+ 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.companionConnectionInterval = setInterval(this.connectToCompanions.bind(t
+ 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":
        const 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() {
+   const companionIds = Object.values(this.companions.all()).map(companion => c
+   const connectedPeerIds = await this.getIpfsPeers()
+   companionIds.forEach(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
- `companionIds.forEach` 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 will perform a simple parallel distributed query on across multiple peers, pooling all pieces 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 instead of `docstore.get('')`.

Key takeaways

- Debug logging can be enabled through a global `LOG` variable
- You cannot discover a user's database address through their IPFS id
- Database discovery, however, can be achieved by utilizing the IPFS pubsub
- When a database is `replicated`, you reliably have access to the data you requested.
- Automatic peer connection can be achieved programmatically based on the data in your database
- Once you have a registry of databases with the same schema, you can write JavaScript functions to perform distributed, parallel queries

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

> TODO: Description

Table of Contents

Please complete Chapter 5 - Peer to Peer Part 2 (OrbitDB) first.

On security

As we've stated before, security-minded people reading this tutorial, or even approaching the distributed industry as a whole, tend to think that this is a wild west full of cowboys who throw caution to the wind while they create systems with security holes you can run a stampede of cattle through.

They're right. OrbitDB and IPFS are *alpha software in an alpha industry*, there have been security flaws discovered, reported, and disclosed and there. Even so far in our app, the databases you have created in this tutorial are only write-protected via a static ACL.

This has the following implications:

1. Everybody will still be able to read the data given a CID or content hash via a simple `ipfs.dag.get` or `ipfs.get` call.
2. The ACL can never change, which means you can't add/revoke other write permissions, and if you lose access to your IPFS node, you can never write to it either.

Security is a vast topic. People can, and do, spend decades of their lives thinking about and working on security in all it's numerous aspects. For the purposes of this tutorial, we will approach security from three perspectives: Encryption, Identity, and Authorization.

Encrypting and decrypting your data

The first thing you'll do to mitigate implication #1 above is to encrypt the data locally, and store it in OrbitDB (and therefore IPFS) in its encrypted form. Your users might want to create some private profile fields, so you'll enable this functionality.

OrbitDB is agnostic in terms of encryption to ensure maximum flexibility with your application layer. However, you can learn a simple method of reading and writing encrypted data to the stores, by creating a simple (and useless in terms of real security) pair of functions.

```
+ encrypt(data) {
+   const stringified = JSON.stringify(data)
+   const reversed = stringified.split("").reverse().join("")
+   return
+ }


+ decrypt(data) {
+   const unreversed = data.split("").reverse().join("")
+   const jsonEncoded = JSON.parse(jsonEncoded)
+   return jsonEncoded
+ }
```

Then, for example, you could update the getProfileFields and updateProfile functions:

```
- getProfileField(key) {
+ getProfileField(key, encrypted=false) {
-   return this.user.get(key)
+   let data
+
+   if(encrypted) {
+     key = this.decrypt(key)
+     data = this.decrypt(data)
+   } else {
+     data = this.user.get(key)
+   }
+
+   return data
+ }

- async updateProfileField(key, value) {
- async updateProfileField(key, value, encrypted=true) {
+   if(encrypted) {
+     key = this.encrypt(key)
+     value = this.encrypt(value)
+   }
    const cid = await this.user.set(key, value)
    return cid
}
```

Of course, this ia a toy example and only reverses strings, but you can see how encryption can be used to protect your data and "hide it in plain sight," so to speak.

What just happened?

You learned a simple but effective method of encrypting and decrypting data for transferring data in and out of OrbitDB.

- `this.encrypt` will encrypt the data locally in memory before storage
- `this.decrypt` will take encrypted data from storage, and decrypt it locally, in memory

You have many options to choose from in terms of, and while we are reticent to make an "official" recommendation, here's a few places you can start to look:

- Node.js crypto module
- Web Crypto Libraries

Encrypting data will work great given a strong enough encryption method. If you picked correctly, you could wait at the restaurant at the end of the universe until somebody guesses your encryption keys and brute forces your data open. The problem, then, becomes **key management**.

Smart Contract Authentication

Distributed identity is still a hotly contested topic, and frankly an unsolved problem, in our industry. However, many organizations have recently lept ahead with solutions and there will be more to come. [Metamask] and its ties to the ethereum blockchain, are one.

What just happened?

Creating your own authentication middleware

What just happened?

Key takeaways

- Resolves: #397
- Resolves: #222
- Resolves: #327
- Resolves: #357

- Resolves: #475
- Resolves: #380
- Resolves: #458
- Resolves: #467

Conclusion

Congratulations, and thank you for completing the tutorial!

By this point, you probably know more about OrbitDB than 99% of the general public, and should be able to start creating distributed applications. You should feel empowered by this knowledge, and the knowledge that this is a frontier as-yet unexplored. So, go forth and *create*! Just make sure you circle back with us and show us all the cool work you're doing.

If you have issues with the tutorial, if you find errors or simply have a suggestion as to better approaches, please create a pull request and provide feedback.

The next part of the book, Thinking Peer to Peer, expands on the deeper concepts that you've been exposed to here at a higher level. Read on.

Part 2: Thinking Peer to Peer

Introduction

Knowing the mechanics of how to create distrbuted applications is only the first step. In order to create truly successful applications that can transcend mere curiousities that only our industry appreciates, you will need a deep understanding of how peer-to-peer applications truly work, and how they interact as a swarm rather than as a client or server.

This won't be easy - models like client-server and relational databases are pervasive in our industry. While this part of the book is not meant to disparage these technologies, learning to *think peer-to-peer* will require some re-education and re-programming of your mentality away from these traditional models.

Also, while the resources that you interact with are the same resources you're used to: computation, memory, and storage, different approaches will be required in order to understand resource limitations and mitigation.

This part concludes with a detailed example of how OrbitDB replicates data, which encompasses many of the topics discussed at a high-level in Part 2.

Peer-to-Peer vs Client-Server

> The Internet's most ubiquitous model of content delivery requires that users run clients that connect to centralized servers. Peer-to-peer is a drastically contrasting approach, and this chapter transitions client-server thinking to peer-to-peer, favoring *swarms over servers*, *peers over clients*, *encryption over authorization*, and *pubsub messaging over API calls*.

Table of Contents

Client-server architecture: a review

In the traditional model, users run *clients* that connect to a *servers*, which store and manage their data. Using the easiest example, users can run multiple Facebook clients - the website, mobile apps, SMS interfaces - that then make requests to Facebook servers, which queries and updates their data.
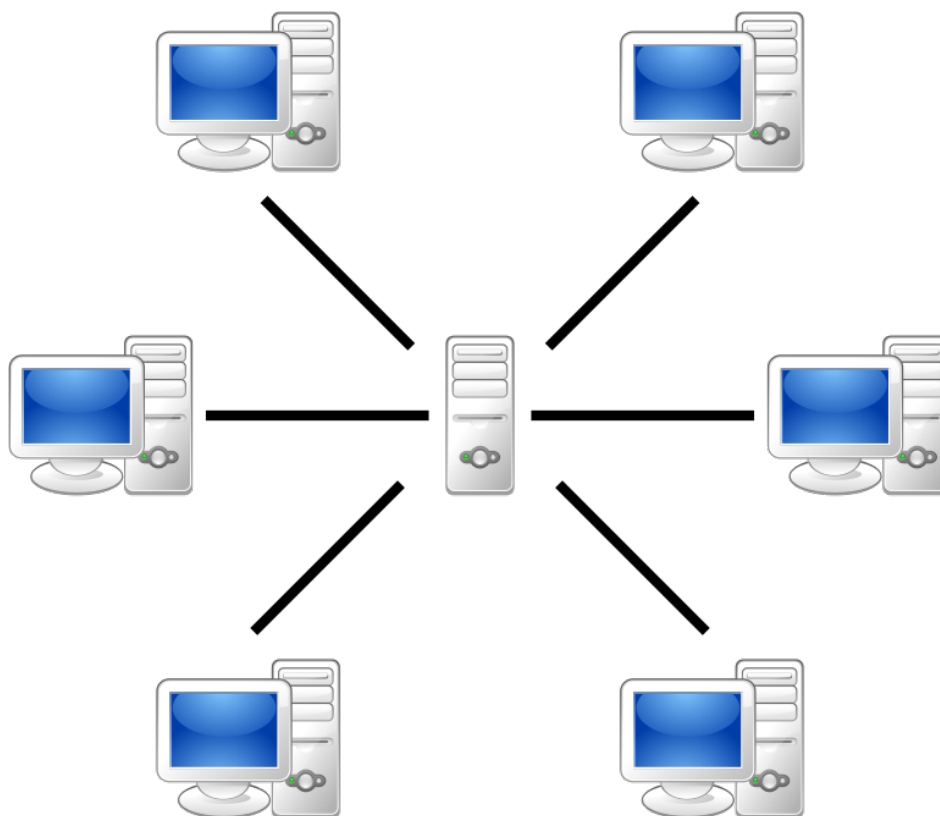
Figure 3: Client-server Model

Over recent years, this method has been proven problematic in terms of data management, data ownership, and data security. These three together can be singularly referred to as *data sovereignty*, meaning the user who generated the data has control over who the data are sent to, what the data are used for, and how.

Now, component by component, you can unlearn the traditional way of thinking, and learn how to conceptualize how a peer-to-peer system might be architected and how it might behave.

Swarms vs Servers

The first notion you should try to dismiss is that of the server, as they are no longer necessary to run peer-to-peer applications. Yes, applications like IPFS nodes run on servers, but it is designed to run in browsers or on regular desktop computers as well.

In the peer-to-peer model, users start out alone and unconnected with their data only stored locally, but very quickly connect to other peers in the same network in a relatively indiscriminate fashion. There is no real "center" of such a network, and thus it forms a **swarm** of connected peers.
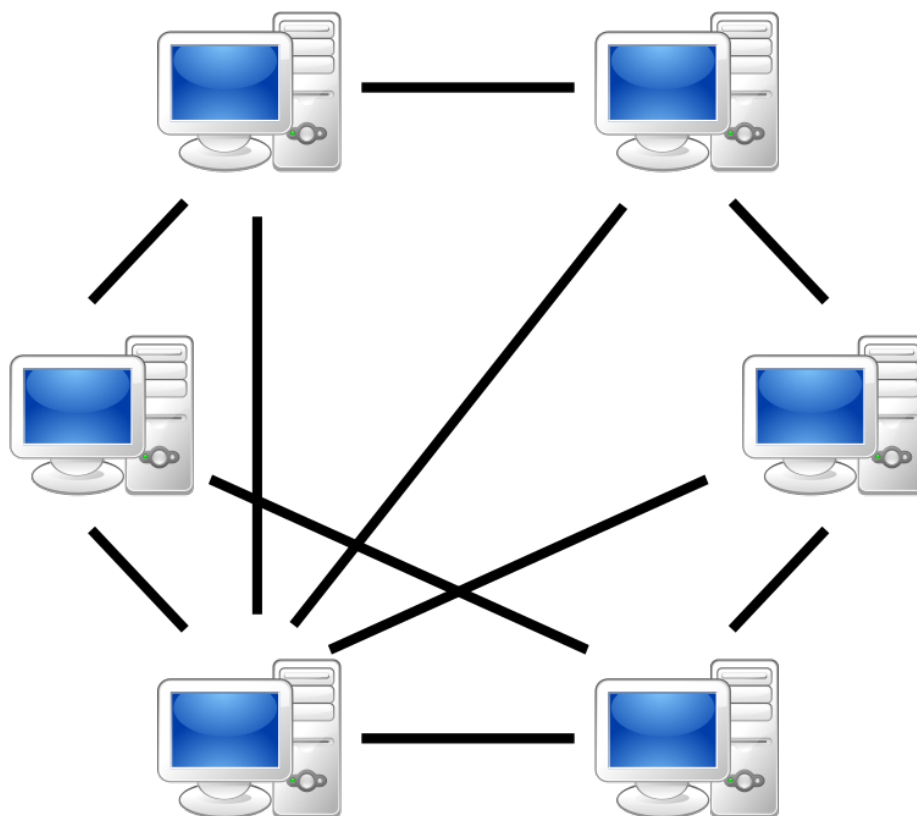
Figure 4: Peer to peer swarm

This approach has obvious pros and cons:

- **Pro:** By default, applications will work offline and keep your data local and safe.
- **Con:** The peers in the network are "untrusted" and will be able to request your data if they know the content address
- **Pro:** The network will be robust and self-healing

- **Con:** If an offline peer that has data you need, you may not be able to access it unless it's replicated elsewhere
- **Pro:** If there *is* a data breach, the so-called blast radius will be contained since it will only affect singular or small groups of devices, not necessarily affecting every user of the application.
- **Con:** There will be new, as-of-yet unimagined, vectors of attack to deceive and compromise data in these networks.

On "Serverless"

It's often claimed, and rightly so, that the popular term "serverless" is at best a misnomer, and at worst completely disingenuous. This term was coined to popularize the idea of writing code that can then be run, at a cost, on a "Function-as-a-Service" platform that all of the major cloud providers run. However, the common criticism is that this is just "somebody else's server," and it's true.

Peer-to-peer architectures *can be* the true "serverless." It is possible, today, to create applications that share user data in a peer-to-peer fashion, avoiding middlemen and third parties, and relying on the surplus computational power. The reason the previous sentence is qualified is because there are some major challenges we will all be facing together in our distributed future:

1. Data Persistence
2. Payment Gateways
3. API Relaying

For now, you're going to see a lot more applications that run on `localhost` by default, that connect to other instances of the application running locally on other computers. There are currently traditional clients that already do this: Gitter, Slack, Discord, to name a few. These desktop apps run locally, but still connect to a server.

Therefore, they are not true *peers.*

Peers vs Clients

If there are no servers, then it would follow that there are no clients. In the traditional model, the functionality of an application is split up between it's server and its client. Perhaps you've heard the terms *thin client*, *business layer*, *front-end*, *back-end*. They all mean specific things but all hint at a core feature of client server - that there is some functionality that a user has access to, and some functionality they don't.

The concept of a **peer** dismisses this notion, and when users run peer-to-peer applications, they are running the entire application code of the entire system locally. This is true of any true peer-to-peer application you are used to: Bittorrent, a Bitcoin wallet, even a program from previous generations like

Kazaa. When you run any of those, you're running the entire system on your computer, and that's that. No additional code is required.

While the functionality of the peer is the same across all instances of the application, the true power of a peer-to-peer system comes from its connection to, and interaction with, other peers. A single bittorrent client is useless without other peers to connect to, just as Bitcoin wallet is useless without generating transactions and consensus with other peers and the underlying blockchain.

Encryption vs Authorization

In a system where anybody can connect to anybody else, security becomes paramount. However, without a centralized server, traditional forms of security become impossible. If there are no central chokepoints to marshall data through, there is no place to authorize every user that wants access. Additionally, users often want as much privacy as possible - even demanding full anonymity (or at least pseudonymity) in many cases.

A prime solution to this problem of distributed security is the use of strong encryption: both to encrypt the data *at rest*, meaning when it is stored on any device, and *in transit*, meaning when it is being transmitted from one device to another over an internet connection.

- **Pro:** There are many types of encryption that are effectively impossible to crack via brute-force methods, meaning that it will take at least millions of years to guess the encryption key
- **Con:** If the encryption keys are lost and cannot be recovered, so too is the encrypted data.
- **Pro:** Systems that pass a large amount of encrypted data, particularly data encrypted via multiple different sets of keypairs, can create a large amount of "noise" making it very difficult for attackers to make heads or tails of what's going on in the system
- **Con:** UX can be tricky, particularly for users new to the distributed space who don't have the same understanding of things like keys and encryption as the developers
- **Pro:** Any encryption key leaks will only affect data encrypted with those keys, not the entire data set of the system

One final trade-off to using keypair encryption as a form of application security is that it introduces the problems in key management, storage, and transfer.

Pubsub vs API Calls

As we move away from the traditional client-server model and towards a swarm of connected peers, we must think differently about the communication model. Traditionally, you would send messages via some sort of API - SOAP, REST, etc. The server would then process those requests, and disseminate information back out to the necessary clients.

But now, we have a peer-to-peer swarm where certain clients may not be connected at the time of messaging, and different types of messages need to be sent. Using the pubsub model - short for "publish and subscribe" - is one way of solving this issue. Peers will be able to subscribe to "topics" and other peers will be able to broadcast on those same topics. Users must be connected to a swarm to be able to pubsub with other peers.

Some examples:

1. One peer might want to query the database of all connected peers. They could do so by broadcasting on a "query" topic and then listening for messages broadcast back on a "result" topic.
2. One peer might be new to the swarm, and would want to announce themselves as online and available to interact with. They could broadcast on an "announce" topic, which would cause other peers to recognize them and connect
3. Pubsub can also be used as a rudimentary chat protocol, allowing peers to broadcast messages to each other via their Node IDs or some other uniquely identifying value.

The names of topics can be defined via unique IDs such as you did in the tutorial to minimize the amount of eavesdropping. These messages should be encrypted in transit, regardless.

OrbitDB vs Traditional Databases

Intro

The Limitations of Peer to Peer

> The limtations of peer-to-peer architecture are similar to those of traditional systems: RAM, CPU, Storage. However, you will be interacting with these resources in different ways.

Storage

Data persistence

Data validation

Computation

Memory

What this is NOT

This is not Bittorrent

This is not Git

This is not Kazaa

Comparison to Traditional Database Systems

What this is

- Resolves #536

On Performance and Scalability

- Resolves #241
- Resolves #479
- Resolves #402

Persistence and Validation

- Resolves #36
- Resolves #310

Interactions with the IPFS Ecosystem

- Resolves #165
- Resolves #540

Part 3: The Architecture of OrbitDB

Introduction

You may still find yourself wondering "but exactly *how* does OrbitDB use IPFS to store, manage, and edit data?" This part is for you. You'll now start from the very bottom of the stack and work your way up to the OrbitDB JavaScript API layer, allowing you to understand, in depth, exactly what's going on internally

This part of the book can be read start-to-finish, or you can utilize it like a reference section, obtaining knowledge from here when you need it. This part culminates in a workshop section that explains how to create your own pluggable stores, allowing for massive amounts of flexibility when designing your own OrbitDB-based applications.

Firmament: The Interplanetary File System

> In order to understand how OrbitDB works, you need to understand a few things about how the Interplanetary File System (IPFS) works. IPFS is unique in a number of ways, but the TODO most relevant to you are how it assigns addresses to data, and

Table of Contents

Content-Addressed vs Location-Addressed

Most content on the internet is *location-addressed*. You type in a familiar name, such as https://github.com/orbitdb and that request is sent to the Domain Name System (DNS), which queries, cross-references, and determines which servers out of the millions out there are the ones with your data on it. Then, that server would understand how to process your query, and send the data back.
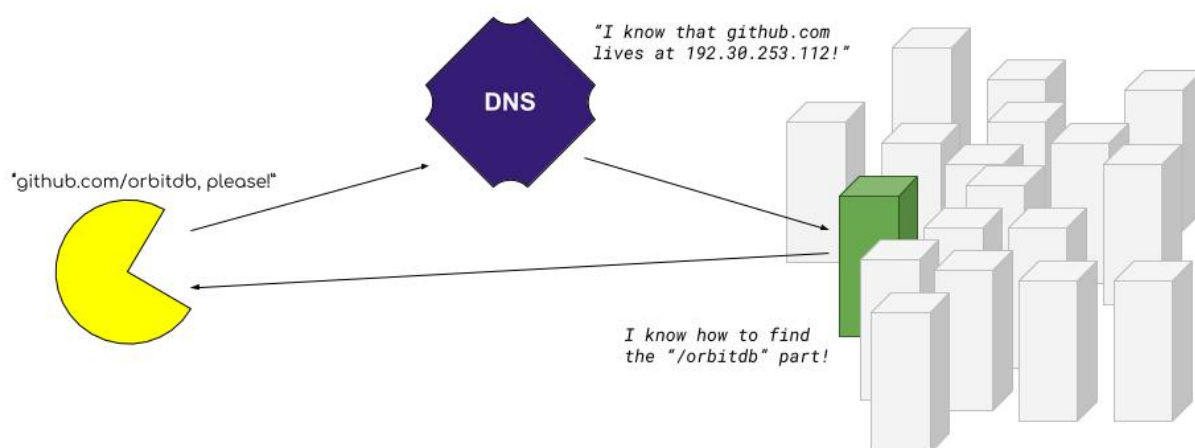

Figure 5: Location-Addressed Illustration

In IPFS, your files are instead *content-addressed*. When you add content to IPFS, that content is given an address based on *what* it is, freeing it from the constraints of its location. You simply ask for what you want, by its *hash*, and multiple servers can respond at the same time if they have the data.
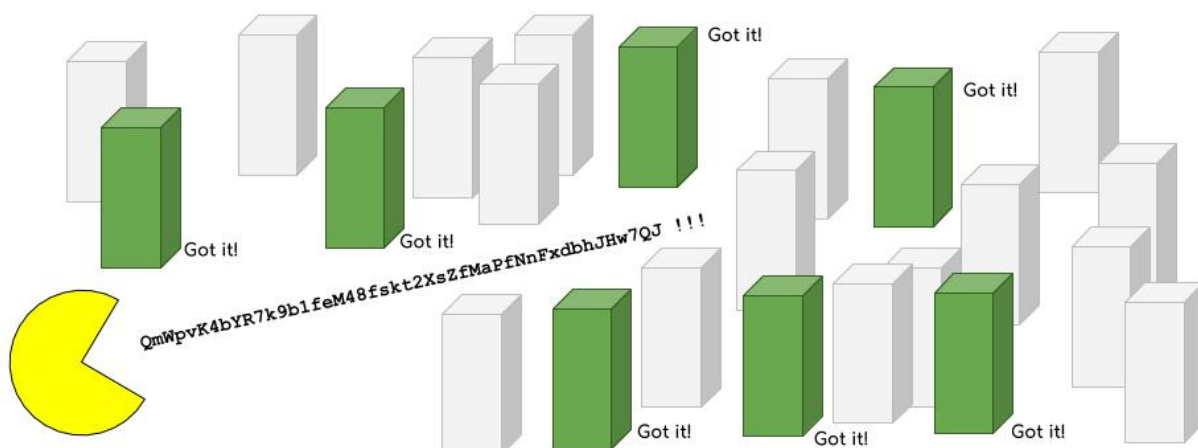
Figure 6: Content-Addressed Hashing

Content addressing is achieved by a technique called *hashing*, which is a very oblique way of saying "chops up your data into blocks, sum them together repeatedly, and reduce the filw down to a unique, consistently-sized alphanumeric string." This is a process identical to generating a "checksum," if you're familiar with that.

For hashing algorithms, there are currently two standards in play: Content ID version 0 (CIDv0) and Content ID version 1 (CIDv1).

- CIDv0 hashes look like this: `QmWpvK4bYR7k9b1feM48fskt2XsZfMaPfNnFxdbhJHw7QJ`
- CIDv1 hashes look like this: `zdpuAmRtbL62Yt5w3H6rpm8PoMZFoQuqLgxoMsDJR5frJGxKJ`

Note: These hashes are a special type called a multihash. In practice, this means they have self-describing prefixes. If you see something starting with zdpu, you know it's a CIDv1.

The two main reasons to switch to content addressing are *performance* and *verifiability*. The performance boost comes from the fact that you can download files simultaneously from multiple peers, similar to Bittorrent. The hashes are also verifiable, meaning that you only download data you request. No other data can have that same hash.

Example

Let's take a very famous file, and add it to IPFS. Here's the plain-text MIT license:

```
Permission is hereby granted, free of charge, to any person obtaining a copy of
associated documentation files (the "Software"), to deal in the Software without
including without limitation the rights to use, copy, modify, merge, publish, di
```

```
and/or sell copies of the Software, and to permit persons to whom the Software i
subject to the following conditions:

The above copyright notice and this permission notice shall be included in all c
portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLI
NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPO
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMA
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

If you copy and paste that , newlines and all, into a file named MIT and then add that text to IPFS, it will return QmWpvK4bYR7k9b1feM48fskt2XsZfMaPfNnFxdbhJHw7QJ every time. That is now, and will be in the future, the *content address* of that file.

Directed Acyclic Graphs

This is a *graph* of connected *nodes*. Think of a network of computers connected via Ethernet cable. Ethernet is directionless; the computers are simply "connected."
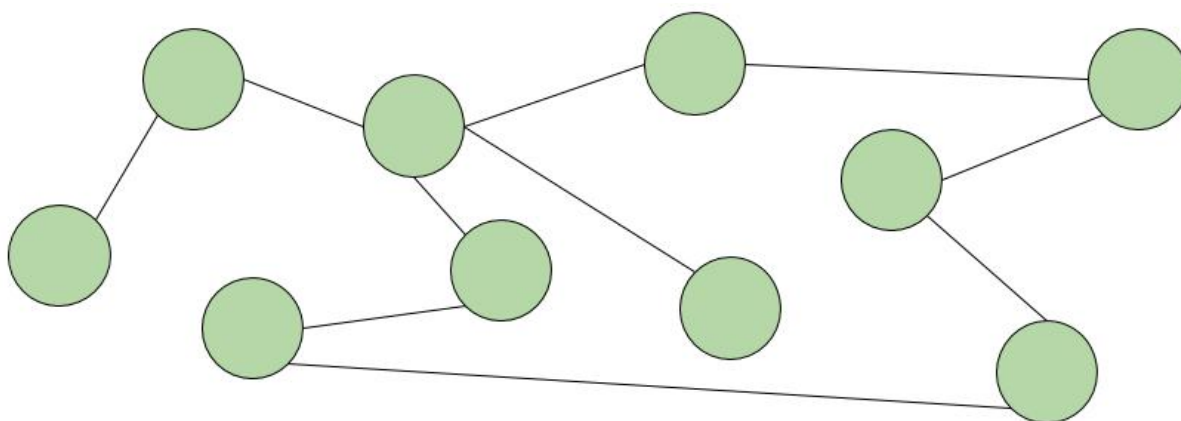

Figure 7: Simple Graph

This is a *directed graph*. Connections flow in a certain direction, but can be reciprocal. Think of users on a social media network that allows "following." You following another user is not the same as them following you, but following can be mutual.
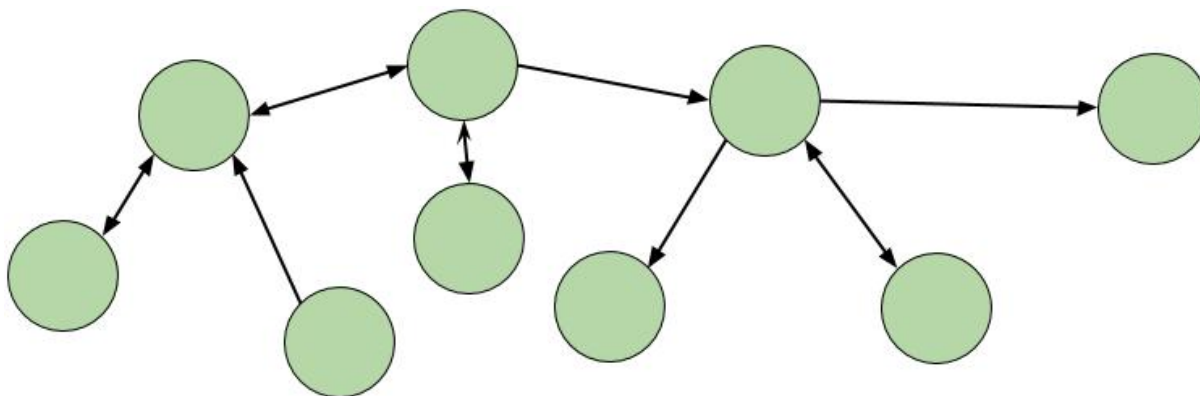
Figure 8: Directed Graph

This is a *directed acyclic graph* or a "DAG". Connections only flow in one direction and never "cycle." or loop back. This
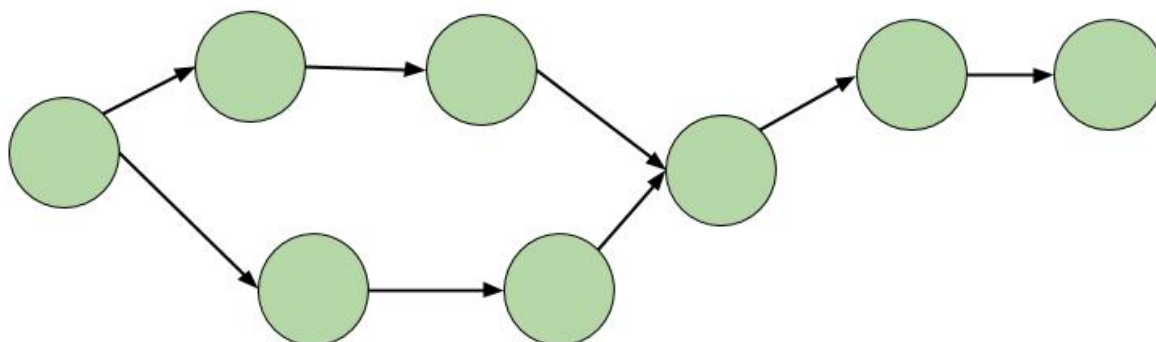

Figure 9: Directed Acyclic Graph

Node connections are generally represented in data by storing a pointer to another node id. For example, modeling a twitter follow in JSON might look sometihng like { id: "@your_username", follows: "@aphelionz" }. In a DAG, a common and very effective way is to point directly their CIDs - the unique cryptographic hash of the content you're looking for. This gives you benefits of using CIDS in general: verifiability and performance, and also the added benefit of being able to *enforce* the acyclic property of the graph - it is effectively impossible for any past nodes to predict the hashes of future nodes in order to store a pointer to them ahead of time.

> **Note:** This technique of using cryptographic hashes to link data is named after Ralph Merkle, so this data structure is called a *Merkle DAG*.

Example

TODO: `ipfs.dag` example

The `ipfs-log` package

Basic `ipfs-log` commands

Dissecting a log entry

The OrbitDB Stores

Keyvalue

Docstore

Counter

Log

Feed

OrbitDB Replication

> OrbitDB replication is something that happens automatically, under the hood. Understanding how to use it is one thing, but many times you'll want a deeper understanding of what happens, step-by-step when a database is created and data is transferred betwen peers.

Table of Contents

Replication, step-by-step

In the tutorial, you learned how to enable debug logging and were able to see the steps of replication take place in your console output. Here, you will be able to dissect these logs and learn how OrbitDB shares data between peers reliably by sharing a minimal amount of data to start.

To review,

On "Sharding"

There have been questions about how sharding is handled in such a peer-to-peer database system.

Workshop: Creating Your Own Store

- Resolves #342 Data persistence on IPFS

Workshop: Creating Custom Authentication Middleware

Introduction

Conclusion

TODO: Call for contributions

Part 4: What's Next?

Introduction

- Resolves #364
- Resolves #377
- Resolves #442
- Resolves #386
- Resolves #434
- Resolves #501
- Resolves #504

Distributed Identity

Encryption Breakthroughs

Why the Web is still critical in peer-to-peer

Overview

Progressive Web Apps

Using OrbitDB in a Web Worker

Appendix I: Glossary

We use a lot of new terms in this tutorial that you may not have run across. In order to make it easier to look up what these terms are, we've added this short glossary. It also helps us to formalize what spelling or definitions we're going with - "datastore" over "data store", isomorphic meaning platform agnostic as opposed to just frontend vs backend, etc.

Terms to use

- **database**:
- **datastore**:
- **isomorphic**: