The OrbitDB Field Manual

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

Abstract

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

Contents

Introduction **5**
- A different perspective on centralized power . . . 5
- What is OrbitDB? . . . 6
  - OrbitDB is written in JavaScript . . . 6
  - OrbitDB is NOT a Blockchain . . . 6
  - OrbitDB is free (as in freedom) software . . . 6
  - OrbitDB is a tool for you, the developer, to build distributed applications . . . 6
- What can I use OrbitDB for? . . . 7
  - Who is already using OrbitDB? . . . 7
- A Warning . . . 8
  - Up next in the book . . . 8
- Who wrote this . . . 9

Part 1: The OrbitDB Tutorial **10**
- Introduction . . . 10
  - Requirements . . . 10
  - What will I learn? . . . 10
  - What will I build? . . . 10
  - Conventions . . . 11
- Chapter 1 - Laying the Foundation . . . 12
  - Installing the requirements: IPFS and OrbitDB . . . 12
  - Creating the isomorphic bookends . . . 13
  - Instantiating IPFS and OrbitDB . . . 13
  - Creating a database . . . 15
  - Choosing a datastore . . . 17
  - Key takeaways . . . 17
- Chapter 2 - Managing Data . . . 19
  - Loading the database . . . 19
  - Adding data . . . 19
  - Reading data . . . 21
  - Updating and deleting data . . . 22
  - Storing Media Files . . . 23
  - Key takeaways . . . 24
- Chapter 3 - Structuring your data . . . 25
  - Adding a practice counter to each piece . . . 25
  - Utilizing the practice counter . . . 26
  - Adding a higher-level database for user data . . . 27
  - Dealing with fixture data . . . 28
  - Key takeaways . . . 29
- Chapter 4: Peer-to-Peer, Part 1 (The IPFS Layer) . . . 30
  - Connecting to the global IPFS network . . . 30
  - Getting a list of connected peers . . . 32
  - Manually connecting to peers . . . 32
  - Peer to peer communication via IPFS pubsub . . . 34
  - Key Takeaways . . . 35
- Chapter 5: Peer-to-Peer Part 2 (OrbitDB) . . . 37
  - Enabling debug logging . . . 37
  - Discovering Peer's Databases . . . 38
  - Connecting automatically to peers with discovered databases . . . 39
  - Simple distributed queries . . . 41
  - Key takeaways . . . 41
- Chapter 6: Identity and Permissions . . . 43

On security . . . 43
Encrypting and decrypting your data . . . 43
Creating your own authentication middleware . . . 44
Key takeaways . . . 45
Conclusion . . . 46

Part 2: Thinking Peer to Peer **47**
Introduction . . . 47
Peer-to-Peer vs Client-Server . . . 48
Client-server architecture: a review . . . 48
Swarms vs Servers . . . 48
Peers vs Clients . . . 51
Encryption vs Authorization . . . 51
Message Passing vs API Calls . . . 52

Part 3: The Architecture of OrbitDB **53**
Introduction . . . 53
Overview of OrbitDB Architecture . . . 53
Firmament: The Interplanetary File System . . . 54
Content-Addressed vs Location-Addressed . . . 54
Directed Acyclic Graphs . . . 55
The `ipfs-log` package . . . 58
The Conflict-Free Replicated Data Type (CRDT) . . . 58
Lamport Clocks . . . 58
Heads and Tails . . . 58
Anatomy of a Log Entry . . . 60
The OrbitDB Stores . . . 63
Keyvalue . . . 63
Docstore . . . 63
Counter . . . 63
Log . . . 63
Feed . . . 63
OrbitDB Replication . . . 64
Replication, step-by-step . . . 64
On "Sharding" . . . 64
Chapter 5 - The RESTful API . . . 65
Setting up . . . 65
Obtaining the SSL certificates . . . 65
Interacting with OrbitDB over HTTP . . . 67
Consuming your first request . . . 67
Replicating . . . 68
Key takeaways . . . 69
Workshop: Creating Your Own Store . . . 70

Part 4: What's Next? **71**
Introduction . . . 71
Distributed Identity . . . 72
Encryption Breakthroughs . . . 73
Why the Web is still critical in peer-to-peer . . . 74
Overview . . . 74
Progressive Web Apps . . . 74
Using OrbitDB in a Web Worker . . . 74

Part 5: Customizing OrbitDB **75**
What will we implement? . . . 75

What do you need to read to understand this chapter? . . . 75
What are Stores, AccessControllers and Indicies. . . . 76
The Store . . . 76
The Index . . . 76
The Access Controllers . . . 76
Choosing a data structure . . . 77
Requirements . . . 77
Generating the Trees from the `ipfs-log` . . . 77
What happens when we delete notes or comments, with all those comments referring to them? 77
Implementing the Index. . . . 77
Isomorphic Bookends. . . . 77
Defining the `NotesIndex` class. . . . 78
Defining the `TreeNode` helper class. . . . 78
Defining the Index . . . 79
A few utility functions. . . . 79
Conclusion . . . 82
Defining the Store . . . 83
The `NotesStore` class . . . 83
Defining the `NotesStore` class . . . 83
Defining a `getNotes` and `getComments` . . . 84
Adding Data to the Store . . . 84
`addNotesBinary` . . . 85
Other Stores . . . 86
Key Takeaways . . . 86
On Customizing OrbitDB . . . 87
Where to go from here? . . . 87
Moderating your Comment Threads. . . . 88
Implementing a custom Access controllers . . . 89
Implementation . . . 89

Appendix I: Glossary **90**
Terms to use . . . 90

Introduction

Welcome to the OrbitDB Field Manual. This book serves as a technical manual to the *distributed*, *decentralized*, *peer-to-peer* database engine known as *OrbitDB*, and also provides higher-level context for peer-to-peer technologies and ways of thinking.

Please read the introduction first and then feel free to skip ahead to the relevant part or parts that interest you. Parts 1 and 3 are more technical in focus, and Parts 2 and 4 are more prosaic.

A different perspective on centralized power

We think there's enough writing out there about the evils of centralization.

Instead, we want to offer a more positive perspective: Every third-party company that has come before us, every technology that they have created, and every piece of data they have stored in their silos has collectively provided a vast set of opportunities for us to build off of inside the decentralized space. We believe that the ziggurat of the centralized is not only a pedestal for us to stand on, but also a springboard for us to leap off of into a bright future - filled with new products and services as yet unimagined.

We now have the capabilities to create distributed, decentralized applications where:

- The user is in full control of their own data.
- Third-party applications ask permission to run their code locally, instead of on their servers.
- Peers connect directly to each other, without third parties.
- Applications start as GDPR and HIPAA compliant, never revealing sensitive information.
- Every layer of the application can be swapped out and configured: the data layer, the middleware, and the UI layer, creating hyper-customizable experiences.
- Industries once thought intractable can be served: finance, healthcare, manufacturing, and more.

We truly believe that everything past, for better or worse, is a prologue to this, and we're so glad you're with us.

What is OrbitDB?

OrbitDB is a database engine that is built on top of the *Interplanetary File System*, or *IPFS*.

You can consider *IPFS* to be the distributed "hard drive" where all data are content-addressed and retrievable from a peer-to-peer swarm. It then follows that *OrbitDB* is the distributed database that lives on that hard drive.

OrbitDB creates and manages mutable databases and provides an extremely simple interface, centered around the IPFS `get` and `set` functions. From these simple underpinnings, OrbitDB manages a great 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 specifically-formatted "clock" values that allow multiple users to perform independent and asynchronous operations on the same distributed 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.

Beyond that, 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 its popularity in the programming community, its ubiquity in web browsers, and its interoperability with the JavaScript implementation of IPFS, called *js-ipfs*.

Work is currently underway to allow support for other programming languages via a common HTTP API.

OrbitDB is NOT a Blockchain

OrbitDB operates on the model of *strong 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 connect with peers, share your logs, and sync your data. This contrasts with Blockchain's idea of *strong consistency* where entries are added to the database only after they have been verified by some distributed consensus algorithm.

There is no built-in "double spend" protection in OrbitDB - that is on you, the developer, to implement.

OrbitDB is free (as in freedom) software

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

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

Outside of a reference implementation called Orbit Chat, the OrbitDB team primarily focuses on making OrbitDB more robust, reliable, and performant. Our mission is to enable you, the developer, to build distributed applications that will break into the mainstream and allow your users true sovereignty and control 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 built with a peer-to-peer architecture, but overall OrbitDB excels in flexibly building complex distributed applications.

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. We maintain a registry called Awesome OrbitDB with several applications and libraries built with OrbitDB. If you're looking for ideas for your own app, maybe start here.

Also, if you have a project you want to show off, Awesome OrbitDB accepts issues and pull requests.

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, and there are more ahead. This is generally transparently communicated, yet these warnings have not stoped large organizations from using these tools in production.

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 also legal, political, economic, and societal. The wave of the personal computer revolution crashed on the shores of the information age, the wave of the mobile revolution crashed on the shores of the social age, and now we're here, riding the crest of the wave of decentralization. We don't know where we're going to land yet, but we know it's gonna be a bumpy ride. Get your life jackets on.

You may hear claims of what type of age might follow this, but it's truly impossible to tell. Whatever it may be, it's probably safe to say that the world will not be ready for it. 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.

That all being said: welcome to the first days of our distributed future.

Up next in the book

Next up is Part 1: The OrbitDB Tutorial. This is a long form tutorial that will introduce a new coder to all the important aspects of OrbitDB. If you are not a coder and simply wish to understand the nature of decentralization and peer-to-peer technologies, check out Part 2: Thinking Peer-to-Peer or Part 4: What's Next?.

Who wrote this

This book has been written largely by the amazing Mark Henderson (don't worry, someone else included the word amazing – he's actually quite humble.). However, it is also a collaborative effort.

If you're interested in writing any section of this book, head to the GitHub repository at orbitdb/field-manual where you can suggest new sections, point out bugs in some code, or just edit some spelling errors.

Anyone who contributes will have their name put into future versions of this book. We'll add you if you contribute something. Here's a list of contributors, so far, in alphabetical order.

- Haad [@haadcode](https://github.com/haadcode)
- Juuso Räsänen [@sirfumblestone](https://github.com/sirfumblestone)
- Mark Henderson ([@aphelionz](https://github.com/aphelionz))
- Richard Littauer ([@RichardLitt](https://github.com/RichardLitt))
- [@shamb0t](https://github.com/shamb0t)
- Teemu
- Vesa-Ville [@vvp](https://github.com/vvp)

If you would prefer not to be in this list, or to provide some different metadata, let us know!

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 a 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
- Instantiating IPFS and OrbitDB
- Creating a database
- Choosing a datastore
- Key takeaways

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 # use npm defaults, you can edit this later
$ npm install --save 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.

If you want to use Git to track your progress, we also suggest the following:

```
$ git init
$ echo node_modules > .gitignore
```

Of course, be careful before copying and pasting any commands anyone ever tells you into your terminal. If you don't understand a command, figure out what it is supposed to do, before copying it over. Copy and paste at your own risk.

> **Note:** This code was tested on Node v11.14.0. Your mileage for other versions may vary.

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:

```
<script src="https://unpkg.com/ipfs@0.55.1/dist/index.min.js"></script>
<script src="https://unpkg.com/orbit-db@0.26.1/dist/orbitdb.min.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:

```
class NewPiecePlease {
  constructor (Ipfs, OrbitDB) {
    this.Ipfs = Ipfs
    this.OrbitDB = OrbitDB
  }
}

try {
    const Ipfs = require('ipfs')
    const OrbitDB = require('orbit-db')

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

Source: GitHub, or on IPFS at `QmRZycUKy3MnRKRxkLu8jTzBEVHZovsYcbhdiwLQ221eBP`.

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 `NewPiecePlease` class.

Instantiating IPFS and OrbitDB

OrbitDB requires a running IPFS node to operate, so you will create one here and notify OrbitDB about it.

> **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) {
    this.OrbitDB = OrbitDB
    this.Ipfs = Ipfs
  }
```

```
  async create() {
    this.node = await this.Ipfs.create({
      preload: { enabled: false },
      repo: './ipfs',
      EXPERIMENTAL: { pubsub: true },
      config: {
        Bootstrap: [],
        Addresses: { Swarm: [] }
      }
    })

    this._init()
  }

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

Source: GitHub.

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

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

NPP.create()
```

In the output you will see something called a "multihash", like `QmPSicLtjhsVifwJftnxncFs4EwYTBEjKUzWweh1nAA87B`. This is the identifier of your IPFS node. (You may have noticed we referenced multihashes above for the code examples: these are the multihashes you can use to download the example code files, if GitHub is down.)

What just happened? Once calling `create`, the `Ipfs.create` line 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 specific location for the folder later.
- `EXPERIMENTAL: { pubsub: true }` enables IPFS pubsub, which is a method of communicating between nodes and **is required for OrbitDB usage**, despite whether or not we are connected to other peers.
- `config: { Bootstrap: [], Addresses: { Swarm: [] }}` sets both our bootstrap peers list (peers that are loaded on instantiation) and swarm peers list (peers that can connect and disconnect at any time to empty. We will populate these later.
- `node.on("error", (e) => { throw new Error(e) })` implements extremely basic error handling 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

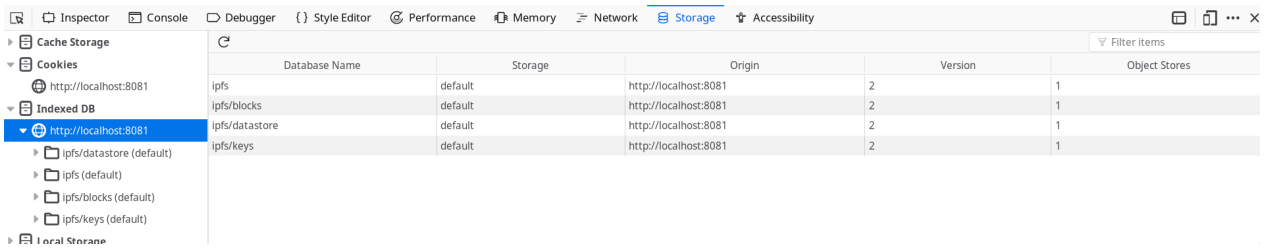

Figure 1: An image showing the IPFS IndexedDB databases in Firefox

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.id] }}
+
+   const docStoreOptions = {
+     ...this.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/zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/p` 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.id] }` defines the ACL, or "Access Control List". In this instance we are restricting `write` access to ONLY the OrbitDB instances identified by our particular `id`
  - `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/  zdpuB3VvBJHqYCocN4utQrpBseHou88mq2DLh7bUkWviBQSE3/

$ 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.

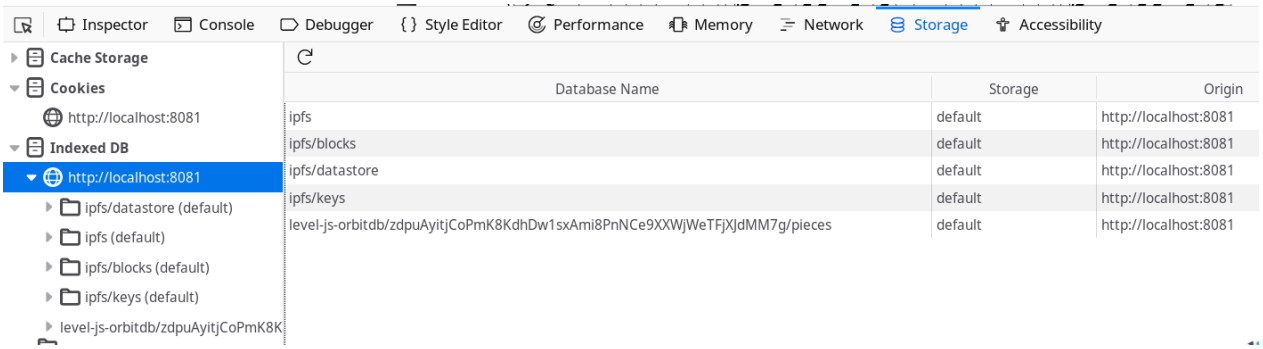

Figure 2: An image showing the IPFS and OrbitDB IndexedDB databases in Firefox

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".
keyvalue	A simple key-value database that supports any JSON-serializable data, even nested objects.
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
- Adding data
- Reading data
- Updating and deleting data
- Storing media files
- Key takeaways

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:

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

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

    this.onready()
  }
}
```

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 pieces.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, with some commented out functions we'll use later.

```
+ async addNewPiece(hash, instrument = 'Piano') {
+   // const existingPiece = this.getPieceByHash(hash)
```

```
+   if (existingPiece) {
+     // await this.updatePieceByHash(hash, instrument)
+     return
+   }
+
+   const cid = await this.pieces.put({ hash, 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 = await 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! All music Copyright © Nintendo.

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

- `pieces.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 `getPiecesByInstrument` function:

Fill in the following functions now:

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

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

```
+ getPieceByInstrument(instrument) {
+   return this.pieces.query((piece) => piece.instrument === instrument)
+ }
```

and uncomment them from `addNewPiece`:

```
+ async addNewPiece(hash, instrument = 'Piano') {
-   // const existingPiece = this.getPieceByHash(hash)
+   const existingPiece = this.getPieceByHash(hash)
+   if (existingPiece) {
-     // await this.updatePieceByHash(hash, instrument)
+     await this.updatePieceByHash(hash, instrument)
+     return
+   }
+
+   const cid = await this.pieces.put({ hash, instrument })
+   return cid
+ }
```

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

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

piece = NPP.getPieceByHash('QmNR2n4zywCV61MeMLB6JwPueAPqtheqpfiA4fLPMxouEmQ')
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[pieces.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.pieces.query((piece) => piece.instrument === instrument)` 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.pieces.put(piece)
+   return cid
+ }
```

```
+ async deletePieceByHash (hash) {
+   const cid = await this.pieces.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", "Harpsichord")
// do stuff with the cid as above

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

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

```
{
  "op":"DEL",
```

```
  "key":"QmdzDacgJ9EQF9Z8G3L1fzFwiEu255Nm5WiCey9ntrDPSL",
  "value":null
}
```

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

- `this.pieces.put` is nothing new, we are just using it to perform an update instead of an insert
- `this.pieces.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 installation */)

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

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

```
document.getElementById("fileUpload").addEventListener('change', async (event) => {
  const file = event.target.files[0]
  if (file) {
    const result = await NPP.node.add(file)
    const cid = await NPP.addNewPiece(result[0].hash)
  }
})
```

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
- Utilizing your practice counter
- Adding a higher-level user database
- Dealing with fixture data
- Key takeaways

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 counter = await this.orbitdb.counter(dbName, this.defaultOptions)

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

  return cid
}
```

In your application code this would look something like this:

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

Which will then output something like:

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

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

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

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

Utilizing the practice counter

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

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

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

These can be used in your application code like so:

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

That will `console.log` out something like:

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

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

- `await this.orbitdb.counter(piece.counter)` is a new way of using `this.orbitdb.counter`, by passing in an existing database address. This will *open* the existing database instead of creating it
- `counter.load()` is called once in `getPracticeCount`, loading the latest database entries into memory for display
- `await counter.inc()` increments the counter, like calling `counter++` would on an integer variable
- `"op":"COUNTER"` is a new operation that you haven't seen yet - remember, you can create stores with any operations you want. More on this in Part 3.
- `"counters": { "042985dafe18ba45c7f1a57db.....02ae4b5e4aa3eb36bc5e67198c2d2": 3 }` is the value returned, the long value is an id based on your node's public key

Adding a higher-level database for user data

Pieces of music to practice with are great to have, but moving forward you will want to allow users to further express themselves via a username and 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.id] }

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

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

    this.onready()
  });
```

Then add the following functions in your class:

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

+ getAllProfileFields () {
+   return this.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.updateProfileField("username", "aphelionz")

const profileFields = NPP.getAllProfileFields()
// { "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`
- `this.user.all` contains all keys and values from a `keystore` database **This is a property, not a function!**
- `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.id] }}

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

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

```
+   await this.loadFixtureData({
+     'username': Math.floor(Math.random() * 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
- Getting a list of connected peers
- Manually connecting to peers
- Peer to peer communication via IPFS pubsub
- Key takeaways

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 connect globally, the `NewPiecePlease` constructor like so:

```
class NewPiecePlease {
  constructor (IPFS, OrbitDB) {
+   this.IPFS = IPFS
    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.reset()
```

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

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

Enabling the swarm Next, you will restore your default swarm addresses. These are 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/QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX
Swarm listening on /p2p-circuit/p2p-websocket-star/ipfs/QmWxWkrCcgNBG2uf1HSVAwb9RzcSYYC2d6CRsfJcqrz2FX
```

In Node.js, run this command:

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

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/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/127.0.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/172.16.100.191/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/172.17.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /ip4/127.0.0.1/tcp/4003/ws/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /p2p-circuit/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /p2p-circuit/p2p-websocket-star/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnByM
Swarm listening on /p2p-circuit/ip4/127.0.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVnB
Swarm listening on /p2p-circuit/ip4/172.16.100.191/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUF
Swarm listening on /p2p-circuit/ip4/172.17.0.1/tcp/4002/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqAVn
Swarm listening on /p2p-circuit/ip4/127.0.0.1/tcp/4003/ws/ipfs/QmXG8yk8UJjMT6qtE2zSxzz3U7z5jSYRgVWLCUFqA
```

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

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

You 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.getIpfsPeers()
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 peerInfo = await this.node.id()
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.id] }}

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

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

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

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

    this.onready()
  }
```

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

```
+ handlePeerConnected (ipfsPeer) {
+   const ipfsId = ipfsPeer.id.toB58String()
+   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.toB58String()` 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 peerInfo = await this.node.id()
    this.orbitdb = await OrbitDB.createInstance(this.node)
    this.defaultOptions = { accessController: { write: [this.orbitdb.identity.id] }}

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

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

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

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

    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) {
+   try {
+     const msgString = JSON.stringify(message)
+     const messageBuffer = this.IPFS.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";
await NPP.sendMessage(hash, data)
```

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

- `this.IPFS.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
- Discovering Peer's Databases
- Connecting automatically to peers via IPFS bootstrap
- Simple distributed queries
- Key takeaways

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

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

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

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

Discovering Peer's Databases

To share data between peers, you will need to know their OrbitDB address. 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 `user` 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.toB58String()
+   setTimeout(async () => {
+     await this.sendMessage(ipfsId, { userDb: this.user.id })
+   }, 2000)
    if(this.onpeerconnect) this.onpeerconnect(ipfsPeer)
  }
```

Now, update your `handleMessageReceived` function to replicate the user database:

```
+ 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, { user: 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`
- `peer.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.id] }}

  const docStoreOptions = {
    ...this.defaultOptions,
    indexBy: 'hash',
  }
  this.pieces = 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.defaultOptions)
+ await this.companions.load()

  await this.loadFixtureData({
    'username': Math.floor(Math.random() * 1000000),
    'pieces': this.pieces.id,
```

```
    'nodeId': nodeInfo.id,
  })

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

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

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

```
+ }
```

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

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

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

- `await this.orbitdb.keyvalue('companions', this.defaultOptions)` creates a new keyvalue store called "companions"
- `this.companions.all` retrieves the full list of key/value pairs from the database
- `this.companions.set(peer.id, peer.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.map` 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 (queryFn) {
+   const dbAddrs = Object.values(this.companions.all).map(peer => peer.pieces)
+
+   const allPieces = await Promise.all(dbAddrs.map(async (addr) => {
+     const db = await this.orbitdb.open(addr)
+     await db.load()
+
+     return db.query(queryFn)
+   }))
+
+   return allPieces.reduce((flatPieces, pieces) => flatPieces.concat(pieces), this.pieces.query(queryFn))
+ }
```

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(x => true)`.

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.

- `this.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
- `db.query(queryFn)` will filter the pieces in the peer's database using the `queryFn` as a filter
- `allPieces.reduce` will take an array of arrays and squash it into a flat array

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

> OrbitDB is extremely flexible and extensible, allowing for security via strong *encryption*, and *custom access control* based on your application's specific code and requirements

Table of Contents

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

- On security
- Encryping and decrypting your data
- Creating a custom access controller

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 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 its numerous aspects. For the purposes of this tutorial, we will approach security from two perspectives: Encryption and Access control

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 reversed
+ }

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

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

```
- getProfileField (key) {
+ getProfileField (key, encrypted = false) {
```

```
-     return this.user.get(key)
+     if (encrypted) key = this.decrypt(key)
+
+     let data = this.user.get(key)
+     if (encrypted) data = this.decrypt(data)
+
+     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 is 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 when it comes to encryption, 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*.

Creating your own authentication middleware

Write access to databases is managed using OrbitDB's access controllers. Access controllers are flexible and extensible; you can either use the default, OrbitDBAccessController, or you can create your own. This means that you can use a third-party application to verify access, such as "Log in with Metamask", or use custom application code to control access.

If no access controller is specified, OrbitDB will use the default OrbitDBAccessController to grant write access. OrbitDBAccessController allows you to list which peers have access. If no peers are specified, only the creator of the database will be granted write access. The peer's identity.id property is used to grant write access to a database when using OrbitDBAccessController.

Alternatively, you can define your own rules to manage write access. You will now implement an access controller using a simple example that can be built upon in the future.

Add a simple function to the `NewPiecePlease` class:

```
+ authenticated() { return true }
```

It's not much to look at, but remember that you can put whatever code you want there to return true or false: You can check a cookie, contact an API, or verify identity against a third party service like Keybase or Twitter. You just want this function to return a boolean.

Then, create a brand new JavaScript class called "NPPAccessController"

```
+ class NPPAccessController extends AccessController {
+   async canAppend (entry, identityProvider) {
+     const authenticated = NPP.authenticated()
+     return Promise.resolve(authenticated)
+   }
+
+   async grant () { }
+ }
```

Finally, add this to your options object to include this access controller, and pass in the options when creating databases.

```
+ const customAccessOptions = {
+    ...this.defaultOptions,
+    accessController: NPPAccessController
+ }
+ const counterDb = await this.orbitdb.counter(dbName, customAccessOptions)
```

Your access controller will then utilize the `NPP.authenticated` function to verify. Right now this will always return `true`, but you are free to modify that function as you see fit based on your custom needs.

What just happened? You created a simple access controller, using code from within the `NewPiecePlease` class to verify.

- `authenticated` is your playground - check cookies, ask your servers, do whatever you need to do
- `AccessController` is the class you want to extend to create your custom access controllers
- `canAppend` is a permission function that returns a resolved promise based on your custom code - a rejected promise means a rejected database append
- `grant`, if implemented, would allow you to grant access to new users of the system over time

Key takeaways

- Security is hard and it is on you, the developer, to keep it in mind as you develop your applications
- OrbitDB is unopinionated in terms of security topics, which provides maximal flexibility
- Strong encryption is an effective way to ensure your data remains private inside a peer-to-peer swarm
- OrbitDB Access Control is fully plugabble and customizable

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 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 distributed applications is only the first step. In order to create truly successful applications that can transcend mere curiosities 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
- Peers vs Clients
- Swarms vs Servers
- Encryption vs Authorization
- Pubsub vs API Calls

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.

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.

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 a peer, that has data you need, is offline, 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 on a "Function-as-a-Service" platform, without any a-priori server set up or configuration, and without on-going costs beyond the immediate execution. However, the *serverless* name refers only to the developer no longer needing to configure or maintain a server themselves, but servers are still needed.

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

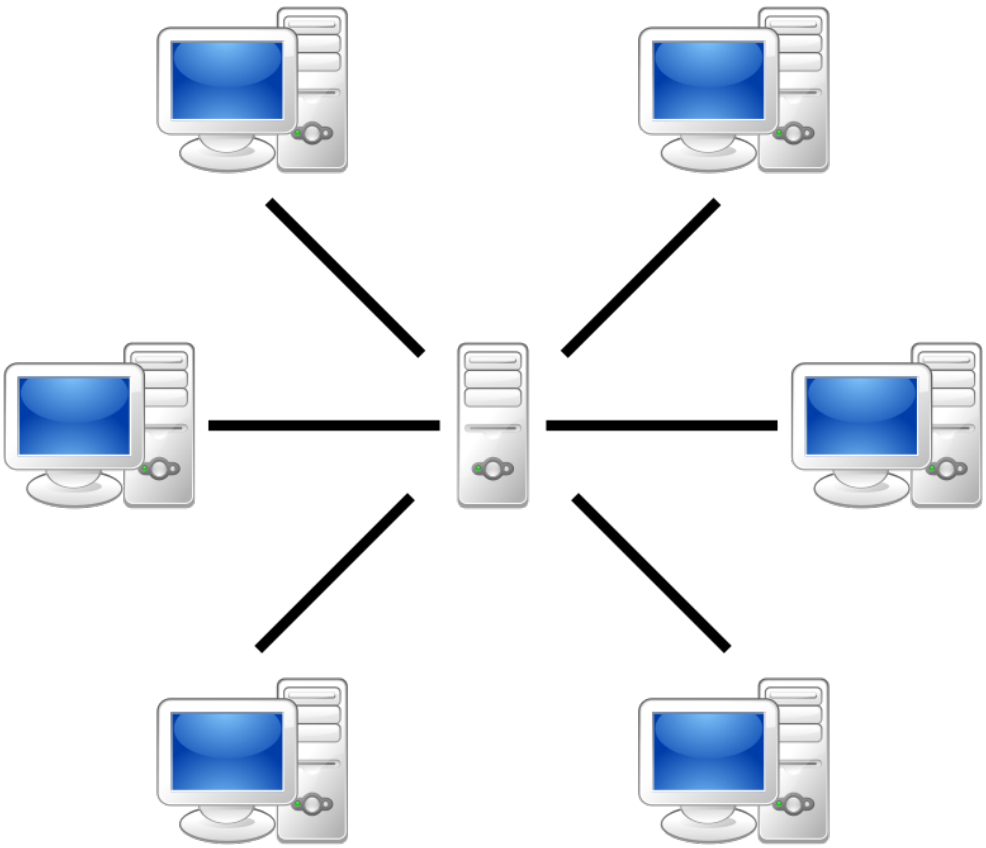

Figure 3: Client-server Model

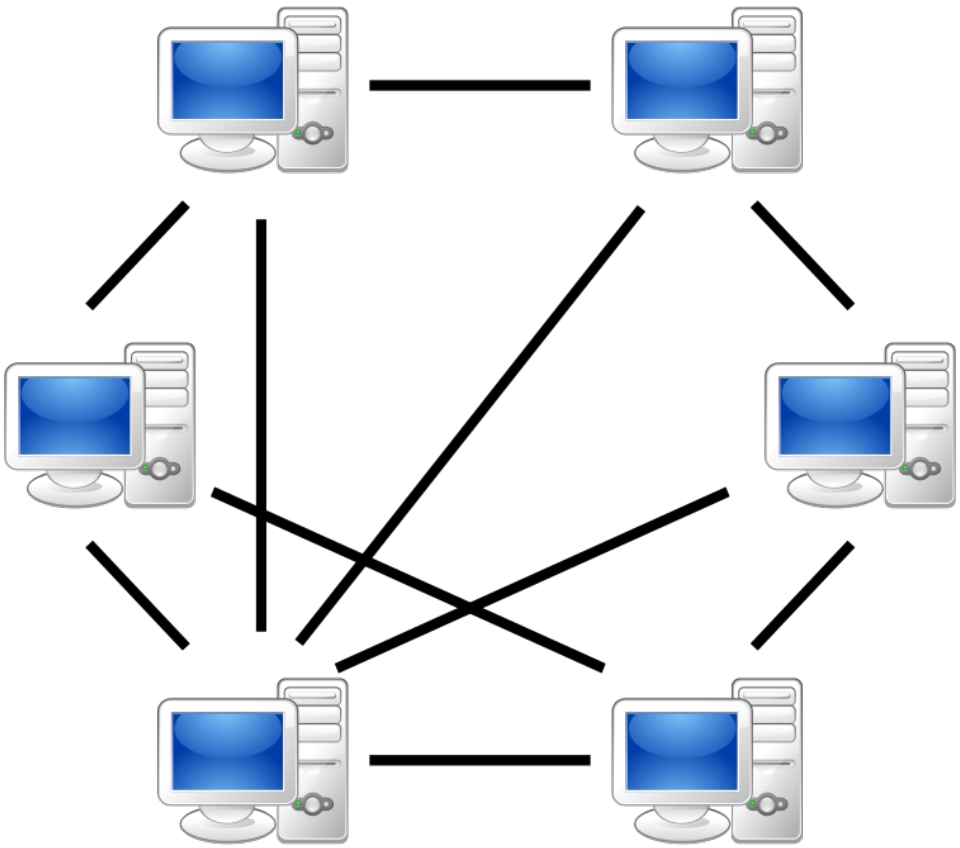

Figure 4: Peer to peer swarm

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 its 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 marshal 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.

Message Passing 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 message passing model, 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. IPFS implements message passing in the form of *pubsub* - short for "publish and subscribe."

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.

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.

Overview of OrbitDB Architecture

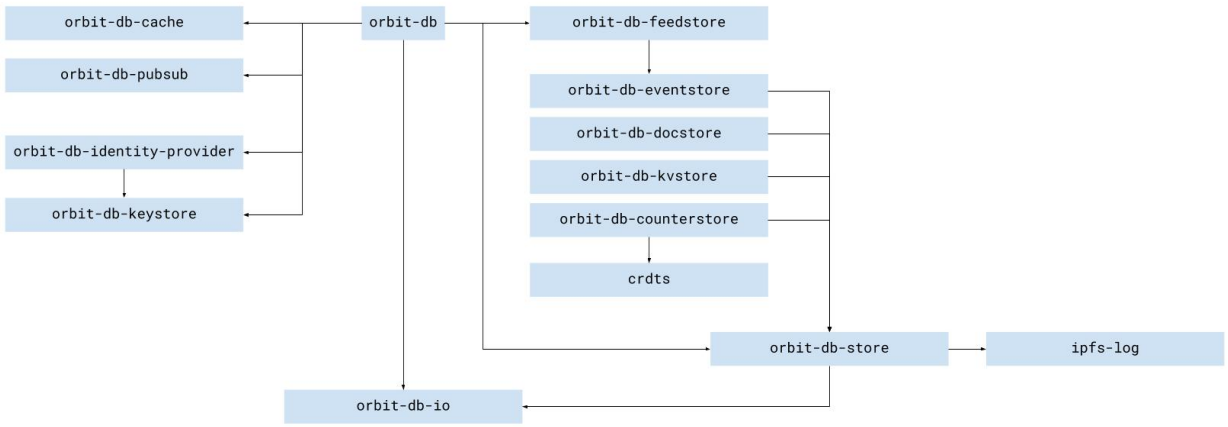

Figure 5: OrbitDB Dependency Graph

Next: Firmament: The Interplanetary File System

Firmament: The Interplanetary File System

> In order to understand OrbitDB, 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 how *linked data* structures are created.

Table of Contents

- Content-Addressed vs Location-Addressed
- Directed Acyclic Graphs

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.

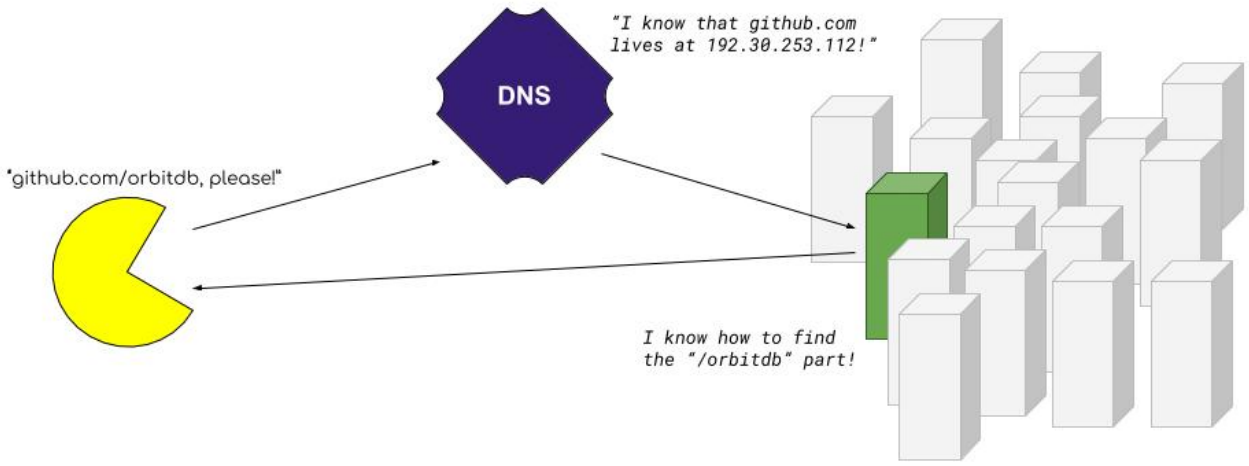

Figure 6: Location-Addressed Illustration

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.

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 file 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.

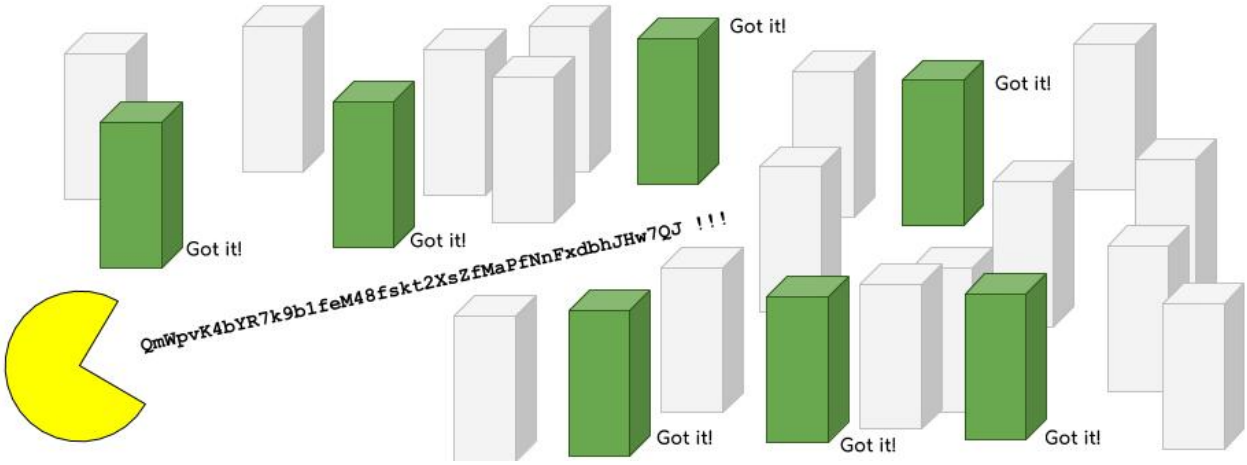

Figure 7: Content-Addressed Hashing

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 this software and
associated documentation files (the "Software"), to deal in the Software without restriction,
including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT
NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
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."

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.

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

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 something 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

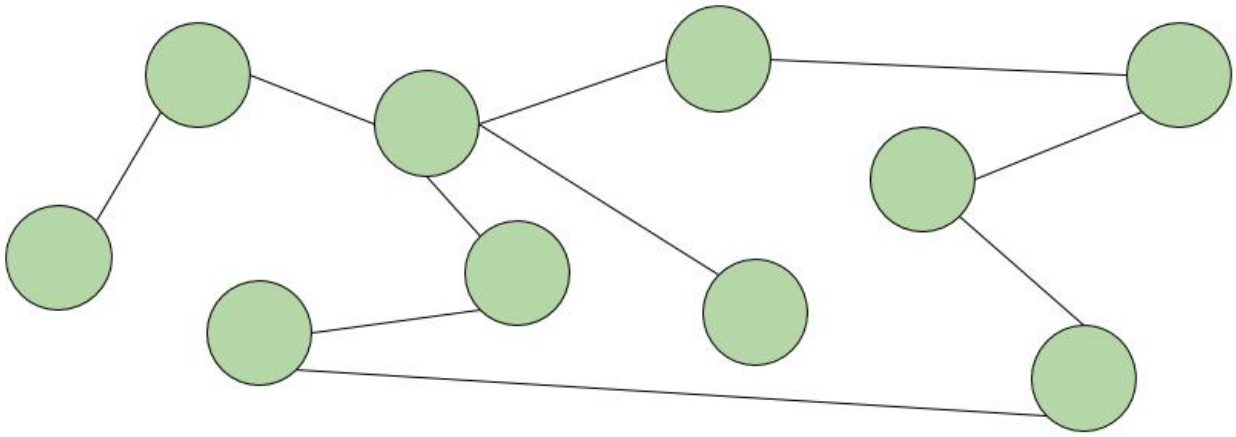

Figure 8: Simple Graph

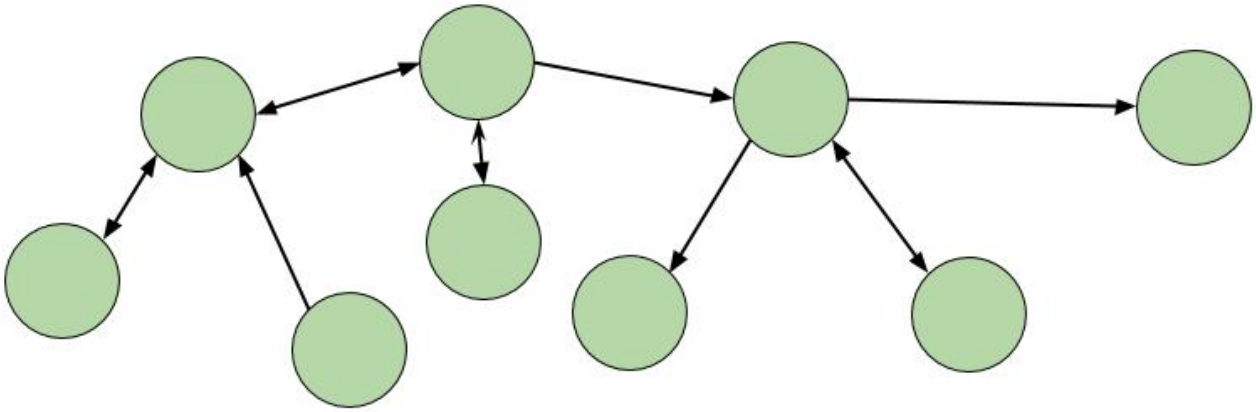

Figure 9: Directed Graph

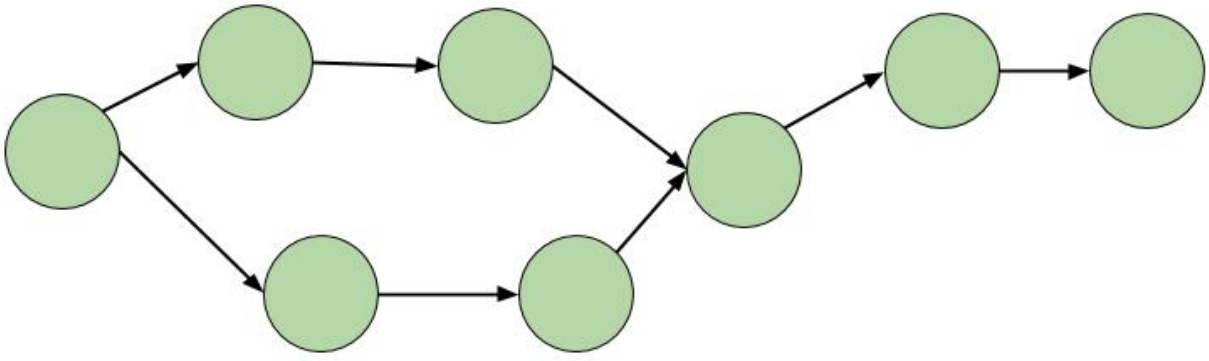

Figure 10: Directed Acyclic Graph

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

> The funcionality provided by the `ipfs-log` package is an implementation of a *Conflict-Free Replicated Data Type* (CRDT) that utilizes IPFS's built in *directed acyclic graph* (DAG) functionality to link data in a specific way. The functionality in this package forms the backbone of orbit-db.

Table of Contents

- The Conflict-Free Replicated Data Type (CRDT)
- Lamport Clocks
- Heads and Tails

The Conflict-Free Replicated Data Type (CRDT)

In the previous chapter we discussed how we can use IPFS's *directted acyclic graph* (DAG) functionality to create linked data structures. OrbitDB utilizes this by building logs wherein each entry is linked to the previous one. To share state reliably between users, and to prevent the system from being confused as to how to parse these logs deterministically, a specific type of data structure called a *Conflict-Free Replicated Data Type*, or CRDT is used.

A CRDT is a type of log that solves the problem of locally storing and ultimately merging distrubuted data sets to other distributed data sets1. CRDTs allows users to perform operations on local databases with the intent of merging or joining those data with the data stored on the devices of other peers in the network.

The `ipfs-log` package specifically uses a G-Set CRDT, which in practice means append-only with no deletetion.

```
class GSet {
  constuctor (values) {}
  append (value) {}
  merge (set) {}
  get (value) {}
  has (value) {}
  get values () {}
  get length () {}
}
```

Lamport Clocks

To achieve successful merging - merging that is properly associative and deterministic - entries are timestamped with something called a Lamport Clock2. The timestamp of each entry is a pair of values: a logical clock counter of the entry (as opposed to wall clock), and an identifier of the user or device that generated the entry.

In the case of /, the identifier is the public key of the IPFS node where the entries are initially generated.

```
// Lamport Clock Object
{
  "id": "042750228c5d81653e5142e6a56d55...e5216b9a6612dbfc56e906bdbf34ea373c92b30d7",
  "time": 0
}
```

> **Note:** The "time" field is a monotonically increasing integer that increments each time a new entry is added to the log. It is not "wall time", e.g. a unix timestamp.

Heads and Tails

Heads and Tails are important concepts in terms of CRDTs, and many people require a bit of explanation before fully understanding the concept and its implications.

Heads The head of a log is an entry that is not referenced by any other entry. Practically speaking, these are the latest entries being appended to a log or logs.

This is best understood by example, observing how the heads change over time. In the following examples, circles are entries, green circles are heads, and arrows denote the pointers contained in the entry, to the previous record.

Let's start with the simplest example - a single user writing entries to a single log.

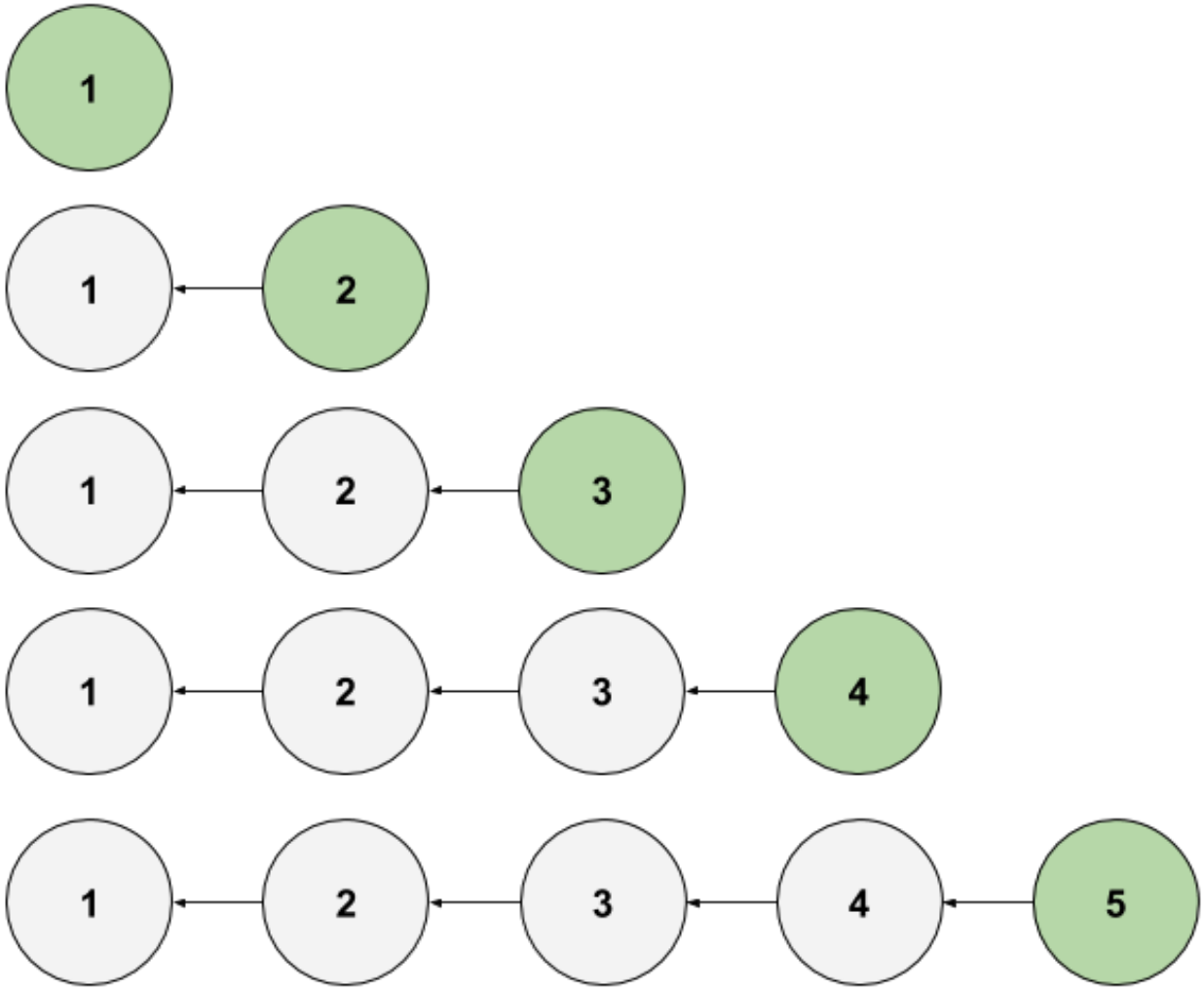

Figure 11: Single-Node CRDT over time, Simplest Example

However, we are not in the business of single-device / single-user logs, so let's imagine the following scenario in an attempt to find the least-complex, but still complete example of how the heads would work over time.

First, in plain words and some pseudocode:

1. User 1 starts a Log (`log1 = new Log`)
2. User 2 starts a Log (`log2 = new Log`)
3. User 1 adds two entries to the log (`log1.append`)
4. User 2 merges that log with their own (`log2.join(log1)`)
5. User 1 adds two more entries (`log1.append`)
6. User 2 adds an entry (`log2.append`)

7. User 2 merges the log again (`log2.join`)
8. User 2 adds one more entry (`log2.append`)

Now in a diagram:

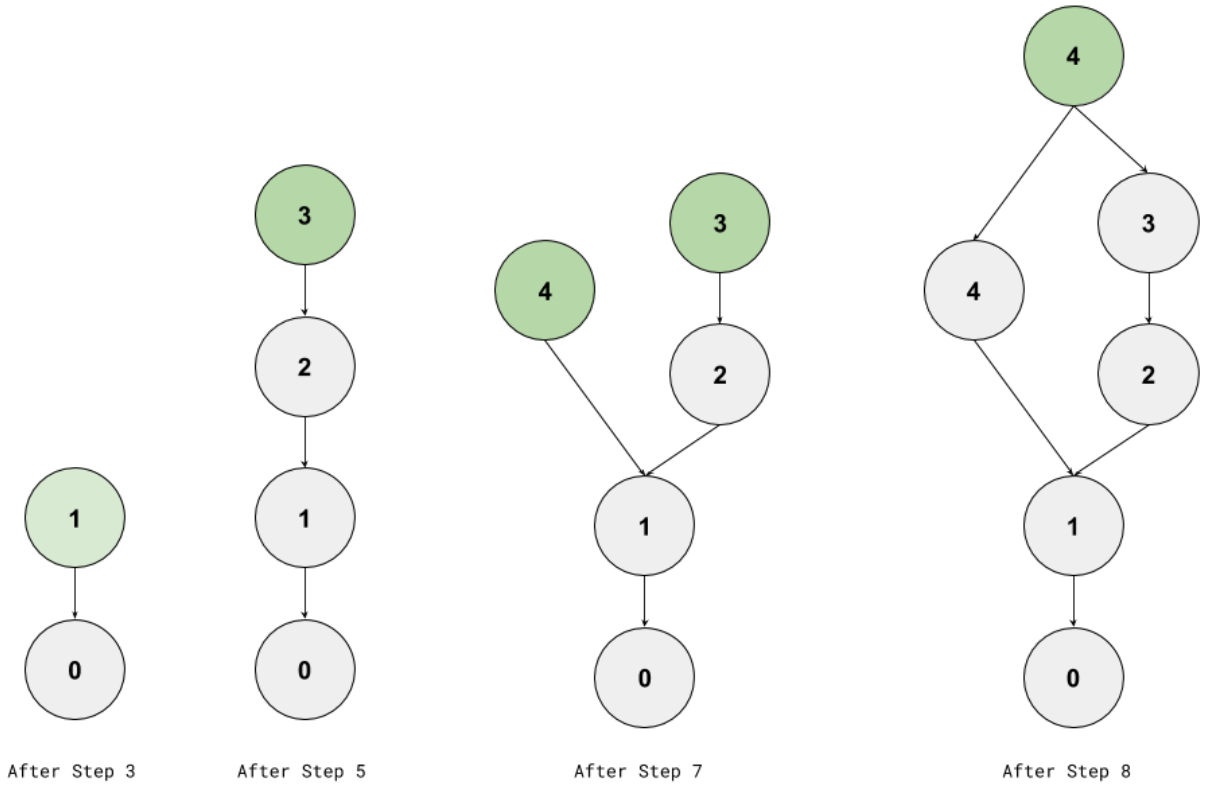

Figure 12: Multiple Nodes Over Time

We can then see how it's possible that a CRDT may have more than one head entry (maybe hundreds) and how those entries change over time with multiple users.

Tails A CRDT of any size can be stored in IPFS. However, When performing computations on the data, it needs to be loaded into an "input array" (i.e. subset of the log) that exists in a finite memory space. The tails of such a log point to entries that are not in the input array.

For our example, let's imagine a log with hundreds of millions of entries. You don't have access to a supercomputing center so tt's not feasible to load the log into memory. Thus, we use a partial traversal of the log, the tails of which contain the pointers to the next records to be traversed, if we so choose.

This concept is visualized below, with the dim entries signifying non-traversed, and the orange entries signifying tails.

Anatomy of a Log Entry

`ipfs-log` entries are JSON objects that follow a specific schema to form linked lists, or "chains" in a (Directed Acyclic Graph) DAG. In doing so, IPFS can be utilized as an append-only operation log.

What follows is sort of a "minimum viable example" of such a log. Below the example is a field-by-field explanation of what's going on.

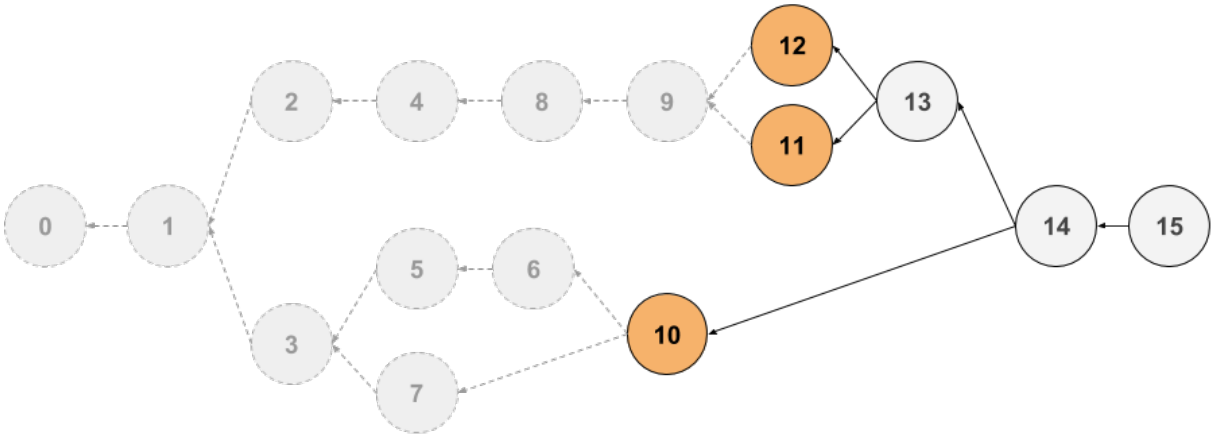

Figure 13: Tails Example


```
[
  {
    "hash": "zdpuAmL9zAgC4KFdMWW6yjpEcSYTnBunjTeijHv8GTKrxoz7D",
    "id": "example-log",
    "payload": "one",
    "next": [],
    "v": 1,
    "clock": {
      "id": "04524f55b54154aa9ec5a54118821c666d19825c0b4e4e91fac387035dbf679803cbdc858b052558f75d5e52ed5
      "time": 1
    },
    "key": "04524f55b54154aa9ec5a54118821c666d19825c0b4e4e91fac387035dbf679803cbdc858b052558f75d5e52ed5b
    "identity": {
      "id": "0281c2c485e5f03f006f8e7ccb34f4281d2d7ae7d8571660e745eb4e07b4d8f35d",
      "publicKey": "04524f55b54154aa9ec5a54118821c666d19825c0b4e4e91fac387035dbf679803cbdc858b052558f75c
      "signatures": {
        "id": "30450221100c7fe5bab3844e197cccc9dc0962977f0381b96acaf9518688a278e0f8ddbc78d0220137e7d154a0
        "publicKey": "3044022043dc1a586910538cd534666d8c66599e6349e82c6ba7fe5bf4d43cfefc4a3e9f02205838a4
      },
      "type": "orbitdb"
    },
    "sig": "3045022100e85408f20d8917eea076e091a64507c1115f9314cf8578b1c483810e82dc6b8902206649fea6407261
  },
  {
    "hash": "zdpuAoGa1MseCvKwbk9xdGK6aNyB5JqdYX8WrkUA8UCz7fRut",
    "id": "example-log",
    "payload": {
      "two": "hello"
    },
    "next": [
      "zdpuAmL9zAgC4KFdMWW6yjpEcSYTnBunjTeijHv8GTKrxoz7D"
    ],
    "v": 1,
    "clock": {
      "id": "04524f55b54154aa9ec5a54118821c666d19825c0b4e4e91fac387035dbf679803cbdc858b052558f75d5e52ed5
```

```
      "time": 2
    },
    "key": "04524f55b54154aa9ec5a54118821c666d19825c0b4e4e91fac387035dbf679803cbdc858b052558f75d5e52ed5b
    "identity": {
      "id": "0281c2c485e5f03f006f8e7ccb34f4281d2d7ae7d8571660e745eb4e07b4d8f35d",
      "publicKey": "04524f55b54154aa9ec5a54118821c666d19825c0b4e4e91fac387035dbf679803cbdc858b052558f75c
      "signatures": {
        "id": "3045022100c7fe5bab3844e197cccc9dc0962977f0381b96acaf9518688a278e0f8ddbc78d0220137e7d154a0
        "publicKey": "3044022043dc1a586910538cd534666d8c66599e6349e82c6ba7fe5bf4d43cfefc4a3e9f02205838a4
      },
      "type": "orbitdb"
    },
    "sig": "304402207cb815276e16d222759f65558d22c3067b54be86f1ce66f1be057e7c188367d6022050e4b93ddc40bc17
  }
]
```


- **hash**: the hash of the entry, in cidv1 format (this will switch to base32 soon)
- **id**: the user-supplied ID of the log
- **payload**: the actual content of the log entry, can be any JSON-serializable object
- **next**: an array of hashes that point to previous log entries from the *head* of the log.
- **v**: the version of the log schema. Typically for internal tracking only and migration purposes
- **clock** the lamport clock values. explained above
- **key** the orbitdb-generated public key for verification purposes
- **identity** the identity object. defaults to the standard OrbitDB identity but can be customized
- **sig** the signaure of the entry, signed by the orbitdb private key, for verification purposes

References

1. https://citemaster.net/get/10b50274-7bc5-11e5-8aa1-00163e009cc7/p558-lamport.pdf
2. https://hal.inria.fr/inria-00555588

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 between peers.

Table of Contents

- Replication, step-by-step
- On "Sharding"

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.

Chapter 5 - The RESTful API

Integrate OrbitDB into your applications using alternative programming languages and frameworks.

Table of Contents

- Setting up
- Obtaining the SSL certificates
- Interacting with OrbitDB over HTTP
- Replicating
- Key takeaways

Setting up

Running an OrbitDB REST server is relatively straight-forward but some knowledge of working on the command line will be required. These steps assume you are running Linux or some other Unix-based operating system. For Windows users, you will need to translate the commands to your environment. Prerequisites

Firstly, it is assumed that you can use a command line and install software as all commands will be run from the terminal.

You will also need two machines running since we will be replicating a decentralized database. This can either be two physical computers, a couple of virtual machines or docker containers.

Lastly, because the OrbitDB server uses Node.js you will also need npm (bundled with Node.js) to install the dependencies. This tutorial will not cover the installation and configuration of these requirements.

Running IPFS OrbitDB uses IPFS to distribute and replicate data stores. The OrbitDB HTTP server runs in one of two modes; local or api.

When run in Local mode, OrbitDB will run its own IPFS node. When run in api mode, OrbitDB will connect to an already-running IPFS node.

For this tutorial we will connect to a running IPFS daemon and will assume you already have this installed. You will also want to run IPFS daemon with pubsub enabled.

Start your first IPFS daemon by running:

```
ipfs daemon --enable-pubsub-experiment
```

Building the REST server Now get a copy of the code. You can grab it via Github at https://github.com/orbitdb/orbit-db-http-api:

```
wget https://github.com/orbitdb/orbit-db-http-api.zip
```

Alternatively, you can clone the git repo:

```
git clone https://github.com/orbitdb/orbit-db-http-api.git
```

Install your dependencies:

```
npm install
```

Obtaining the SSL certificates

The latest version of the OrbitDB HTTP API incorporates HTTP/2. Therefore, to run the server, you will need to generate SSL certificates.

There are a couple of options available for obtaining certificates; you can issue a certificate using a certificate authority such as Let's Encrypt, or, you can become your own CA. For development environments, the second option may be better and a thorough overview on how to do this is covered in the section Generating a self-signed certificate.

The rest of this tutorial will assume you have a trusted SSL certificate set up and that curl will use your trust store to validate the certificate. If not, you will need to tell curl to ignore the certificate verification by passing the -k flag:

```
curl -k -X GET ...
```

Generating a self-signed certificate To get started, you are going to create a root certificate which you will use to sign additional SSL certificates with.

First, create your root CA private key:

```
openssl genrsa -des3 -out rootSSL.key 2048
```

```
Generating RSA private key, 2048 bit long modulus
..................+++
.....................................................................................+++
e is 65537 (0x010001)
Enter pass phrase for rootSSL.key:
```

You will be prompted for a password. Be sure to specify one that is long enough as you may encounter errors if your password is too short.

Next, use your CA private key to create a root certificate:

```
openssl req -x509 -new -nodes -key rootSSL.key -sha256 -days 1024 -out rootSSL.pem
```

Once launched, you will need to re-enter the password you assigned to your private key:

```
Enter pass phrase for rootSSL.key:
```

If successful, provide information about your certificate:

```
 You are about to be asked to enter information that will be incorporated
 into your certificate request.
 What you are about to enter is what is called a Distinguished Name or a DN.
 There are quite a few fields but you can leave some blank
 For some fields there will be a default value,
 If you enter '.', the field will be left blank.
 Country Name (2 letter code) [AU]:
 State or Province Name (full name) [Some-State]:WA
 Locality Name (eg, city) []:
 Organization Name (eg, company) [Internet Widgits Pty Ltd]:
 Organizational Unit Name (eg, section) []:
 Common Name (e.g. server FQDN or YOUR name) []:localhost
 Email Address []:
```

You are now ready to install the new CA certificate into your CA trust store. The following commands will copy the root certificate into Ubuntu's CA store so you may need to modify the paths if you are on a different distribution or OS platform:

```
sudo mkdir /usr/local/share/ca-certificates/extra
sudo cp rootSSL.pem \/usr/local/share/ca-certificates/extra/rootSSL.crt
sudo update-ca-certificates
```

Now it is time to generate a certificate for your development environment. Create a private key for your new certificate:

```
openssl req \
 -new -sha256 -nodes \
 -out localhost.csr \
```

```
 -newkey rsa:2048 -keyout localhost.key \
 -subj "/C=AU/ST=WA/L=City/O=Organization/OU=OrganizationUnit/CN=localhost/emailAddress=demo@example.com
```

Next, create the certificate, signing it with your Root CA:

```
openssl x509 \
 -req \
 -in localhost.csr \
 -CA rootSSL.pem -CAkey rootSSL.key -CAcreateserial \
 -out localhost.crt \
 -days 500 \
 -sha256 \
 -extfile <(echo " \
    [ v3_ca ]\n \
    authorityKeyIdentifier=keyid,issuer\n \
    basicConstraints=CA:FALSE\n \
    keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment\n \
    subjectAltName=DNS:localhost \
   ")
```

Your self-signed SSL certificate is now ready to use.

Interacting with OrbitDB over HTTP

Starting the HTTP API server Start up the OrbitDB server and connect to your running ipfs daemon:

```
node src/cli.js api --ipfs-host localhost --orbitdb-dir ./orbitdb --https-key localhost.key --https-cert
```

The `-https-key` and `-https-cert` options above assume you are using the self-signed certificate you created earlier. If not, replace with your own certificate and key.

Consuming your first request

The REST server is now running. You can test this by running something simple (we are going to use cURL to run the rest of these command so make sure you have it installed):

```
curl -X GET https://localhost:3000/identity
```

This will return a JSON string representing your OrbitDB node's identity information. This includes your public key (which we will use later).

Creating a database Creating a data store is very easy with the REST API and you can launch a store based on any of the supported types. For example, you can create a feed data store by running:

```
curl -X POST https://localhost:3000/db/my-feed --data 'create=true' --data 'type=feed'
```

You can also use JSON to specify the initial data store features:

```
curl -X POST https://localhost:3000/db/my-feed -H "Content-Type: application/json" --data '{"create":"tr
```

Adding some data Let's add some data to our feed:

```
curl -X POST https://localhost:3000/db/my-feed/add --data-urlencode "A beginner's guide to OrbitDB REST
```

And viewing the data we have just added:

```
curl -X GET  https://localhost:3000/db/my-feed/all
["A beginner's guide to OrbitDB REST API"]
```

Be aware that there are two different endpoints for sending data to the store, and which endpoint you use will depend on the store's type. For example you will need to call /put when adding data to a docstore.

Replicating

Replicating is where the real power of distribution lies with OrbitDB. Replication is as simple as running an OrbitDB REST node on another machine.

Assuming you have a second computer which is accessible over your intranet or via Docker or a virtual machine, you can replicate the my-feed feed data store.

Getting ready to replicate Before you replicate your feed data store, you will need to make a note of its address. You can do this by querying the data store's details:

```
curl https://localhost:3000/db/my-feed
{"address":{"root":"zdpuAzCDGmFKdZuwQzCZEgNGV9JT1kqt1NxCZtgMb4ZB4xijw","path":"my-feed"},"dbname":"my-fe
```

Copy the id. We're going to use it in the next step.

Running another copy of the data store On your second machine, make sure you have IPFS running and the OrbitDB REST server installed and running.

Replicating the my-feed data simply requires you query the first machine's my-feed data store using the full address. Using the address of the my-feed data store I queried earlier, request the data:

```
curl https://localhost:3000/db/zdpuAzCDGmFKdZuwQzCZEgNGV9JT1kqt1NxCZtgMb4ZB4xijw%2Fmy-feed/all
["A beginner's guide to OrbitDB REST API"]
```

You may need to run the curl call a couple of time; OrbitDB will take a small amount of time to replicate the data over.

There are two important things to note about the address; 1) we drop the /orbitdb/ prefix and 2) we need to url encode the /. The html encoded value of / is %2F.

And that's it. You have successfully created a new OrbitDB data store on one machine and replicated across another.

Let's test it out. Back on your first machine, add another entry to the feed data store:

```
curl -X POST https://localhost:3000/db/my-feed/add --data-urlencode "Learning about IPFS"
```

On your second machine, retrieve the feed list again:

```
curl https://localhost:3000/db/zdpuAzCDGmFKdZuwQzCZEgNGV9JT1kqt1NxCZtgMb4ZB4xijw%2Fmy-feed/all
["A beginner's guide to OrbitDB REST API","Learning about IPFS"]
```

Adding data in a decentralized environment What happens if you want to add more entries to the my-feed data store from your second machine:

```
curl -X POST https://localhost:3000/db/my-feed/add --data-urlencode "Adding an item from the second Orbi
{"statusCode":500,"error":"Internal Server Error","message":"Error: Could not append entry, key \"03cc59
```

If you check the output from your REST server you will see a permissions error. By default, any replicating node will not be able to write back to the data store. Instead, we have tell the originating OrbitDB instance that the second instance can also write to the my-feed data store. To do this, we must manually add the public key of the second OrbitDB instance to the first instance.

It is important to note that the data store must be created with an access controller pre-specified. Start by deleting the data store on the first machine:

```
curl -X DELETE https://localhost:3000/db/my-feed
```

We must now set up the my-feed database again and add some data:

```
curl -X POST https://localhost:3000/db/feed.new -H "Content-Type: application/json" --data '{"create":"t
```

Note the accessController property; this specify the controller type and the key which can write to the database. In this case it is the first machine's public key, which can be retrieved by running:

```
curl https://localhost:3000/identity
```

On the second machine, retrieve the public key:

```
curl https://localhost:3000/identity
```

Grab the publicKey value. We will now enable write access to the my-feed database:

```
curl -X PUT https://localhost:3000/db/feed.new/access/write --data 'publicKey=04072d1bdd0e5e43d9e10619d9
```

publicKey will be the publicKey of the second machine. We must execute this request from the first machine because only the first machine currently has write permissions to the data store.

With the second machine's publickey added, we can go ahead and add a new my-feed from the second machine:

```
curl -X POST https://localhost:3000/db/my-feed/add --data-urlencode "Adding an item from the second Orbi
```

Key takeaways

- You can communicate with OrbitDB using something other than NodeJS,
- The RESTful API provides you with a rich set of features which you can use to implement powerful distributed data stores in other programming languages,
- The data stores you create using the RESTful API can be replicated elsewhere.

A full list of available RESTful endpoints as well as installation and other documentation is available on the official OrbitDB HTTP API Github repository.

Workshop: Creating Your Own Store

- Resolves #342 Data persistence on IPFS

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

Part 5: Customizing OrbitDB

In the first chapter of this Field Manual, we described how you might create a couple of built-in OrbitDB databases and write code to modify it.

We shall try to build on this tutorial to show you, how you can customize the built-in OrbitDB stores and indices or how you can define your own.

What will we implement?

We have already implemented a sheet music sharing app, where musicians can upload and share their notes as files and view those of other musicians.

But what they cannot do, is cooperate with each other. And isn't it with cooperation, that the greatest strengths of the internet come to bear?

So we want to allow our users to not just share musical notes, but also comment on each others notes and discuss them.

What do you need to read to understand this chapter?

It is not expected that you have read all of this Field Manual to understand this chapter.

But you should have read these documents before reading this chapter:

1. The Tutorial
2. ipfs-log

You may not need to understand entirely what the `ipfs-log` is, but you should know that it exists and that it is the basis of all OrbitDB Stores.

Next: What are Stores, AccessControllers and Indicies.

What are Stores, AccessControllers and Indicies.

Before we start to implement our custom store, access controller, and index to allow our users to comment on each others notes, we should probably define what a `Store`, `AccessController`, and `Index` are and what function they have in OrbitDB.

The Store

You have already worked with several Stores in the Tutorial. The `docstore` was used to store the individual pieces (or rather their CIDs) and the `kvstore` type was used to represent a user.

Both of those were instances of a `Store`. They calculated some state from `ipfs-log` and provided easy functions to access them, like `get`, `put` or `set`, with which you could easily interact with the database itself.

The Index

But a store doesn't actually store the current state of the database in the RAM, because this is done in the `Index`.

The index parses the log of operations generated by the store while modifying the database and generates an easy to access and use representation of the current state of the database.

The index is then used by the `Store` to implement its API.

The Access Controllers

Access Controllers or short ACL (Access Control List) were already discussed 02: Managing Data.

They are a piece of code that is invoked whenever you try to write to the database to ensure that you have the right to do so.

It is setup when you create a database and stored in the Manifest of the database. (The multihash `zdpuAmQhKvDytoSn6NapRYZYTAWgqQpbJBfPLmUiSRBYgN7ah`in is refering to the Manifest file.`/orbitdb/zdpuAmQhKvDytoSn6Nap`

Anyone, who supports this type of access controller, can follow the rules of the access controller.

But this also means, that you cannot change the type of your Access Controller, without changing the Manifest file and thus the OrbitDB Address.

Next: Choosing a data structure

Choosing a data structure

Let's start implementing a comment system by choosing how we want to represent the comments in the Index.

Choosing a data structure first and working our way outwards to the store API, that best fits to the data structure and then implementing the Access Controller, since the Access Controller is very much independent from the Store and Index.

Requirements

Our data structure should achieve two things:

- It should store the CID of the sheet music and a string denoting the instrument.
- It should also store the comments, which themselves refer to a sheet music CID or a previous comment.

The simplest data structure that could fulfill these requirements would be a collection of trees. Each tree's root would be the Notes pieces and the children of that root would be the comments, the comments comments and so on and so on.

Generating the Trees from the `ipfs-log`

Before starting to implement this data structure, we'll also have to consider, how to generate these trees from the oplog or `ipfs-log`.

In the index, we'll see the oplog as an Array of operations, each containing these fields:

- `op` to denote the operation being made
- `key` some key, that we can set if we so choose.
- `value` the value of the actual operation.

For our purposes there can be these operations:

- `ADDNOTES` to add a new piece of notes.
- `DELETENOTES` to delete a piece of notes.
- `ADDCOMMENT` to add a comment to a piece of notes or some other comments.
- `DELETECOMMENT` to delete a comment.

This seems pretty straight forward.

What happens when we delete notes or comments, with all those comments referring to them?

I propose using a simple rule: If a comment or piece of notes is deleted, all those refering to it are deleted too. Otherwise this tutorial becomes a complicated mess.

Implementing the Index.

Let's start by adding a new file to your project folder (if you haven't yet created one, do that now).

Isomorphic Bookends.

You know the drill, before starting with the actual implementation of the Index, we have to define the bookends, that make our code work in the browser and in NodeJS:

```
"use strict"

try {
  module.exports = NotesIndex
} catch (e) {
  window.NotesIndex = NotesIndex
}
```

Not that complicated, really. Because an Index does not actually depend libraries from OrbitDB.

It is using duck typing, instead of inheritance.

Defining the `NotesIndex` class.

Next, let's define the actual class of the `NotesIndex`. Add this to your file, between the bookends and the `use strict`.

```
class NotesIndex {
  constructor() {
    this._index = {}
  }
}
```

We initialize the index to an empty object at first, because we don't yet have any data.

Defining the `TreeNode` helper class.

After this, let's first implement our own `TreeNode` data type, that'll be used to represent the trees mentioned above.

Because Trees are fundamentally recursive data structures, a Tree and a `TreeNode` are the same thing. Both hold some data and have some children, who are of type `TreeNode`.

```
class TreeNode {
  constructor(data) {
    this.data = data
    this.children = []
  }

  addChild(data) {
    let node = new TreeNode(data)
    this.children.push(node)
    return node
  }
}
```

With this utility class, we can now get on to implementing the Index API.

Next: Defining the Index

Defining the Index

In the last chapter we considered how we could define our `Index` data structure.

I settled us on a Tree data structure, where each notes piece are a root and the comments would be children of that root.

We then laid out the skeleton code for the Index and in this chapter we will actually define the Index or what makes the `NotesIndex` class an `Index`.

A few utility functions.

Let's first define a few utility functions for fetching content from the `Index`.

- Fetching a notes piece.
- Fetching the comments on a piece of notes
- Getting comments in a chronological order for a specific piece of notes.

The last will be very helpful to UI Designers who have the crucial task of translating this data structure into pleasant UI.

`getNotes` But let's start with a simple `NotesIndex.getNotes` function.

```
getNotes(cid) {
  return this._index[cid].data
}
```

We fetch the tree with that CID and then only care for the `data` field of the `TreeNode`, because that's where the sheet music is actually stored.

`getComments` Now implement the `NotesIndex.getComments(cid)` function like this:

```
getComments(cid) {
  return this._index[cid].children
}
```

Get comments is almost entirely identical to `getNotes`, except that we don't return the `data`, but the `children` field of the `TreeNode`.

`getComments` with chronological order Now, let's get to our third and final `get` functions: `getComments`, but now with an argument: `flat = true`. The purpose of this function is, to if the `flat` argument is true, flatten the comments into a neat chronological list, that can be easily displayed. To do this, we first gather all of the comments into an array and then sort them based on an ever increased `id` field:

```
getComments(cid, flat = true) {
  function flatten(children) {
    return children.reduce((comments, comment) => {
      comments.push(comment.data)
      return comments.concat(flatten(comment.children))
    }, [])
  }

  if(flat) {
    return flatten(this._index[cid].children).sort((a, b) => a.id - b.id)
  } else {
    return this._index[cid].children
  }
}
```

Replace the `getComments` function above, by this.

What happens here? We first define a helper function `flatten`, which goes through the array of children and adds each node of the tree therein to a flat array. Then we sort it based on an id field in ascending order. If you pass in `flat = false`, you'll still get the old behavior.

The `updateIndex` function Up until this point, we have been writing utility functions that are nice to have for our index to be used by our still to be defined store, but is not strictly necessary for an `Index` to work as such.

The only method all `Index` classes have to have is `updateIndex`.

`updateIndex` receives as an argument the `oplog` of database, which is an Array of Operations compiled from the `ipfs-log` in chronological order.

So, let's lay out the skeleton:

```
updateIndex(oplog) {
  let order = 0
  oplog.values.reduce((handled, item) => {
    if(!handled.includes(item.hash)) {
      handled.push(item.hash)

      switch (item.payload.op)) {
        case "ADDNOTES":

          break;
        case "DELETENOTES":

          break;
        case "ADDCOMMENT":

          break;

        case "DELETECOMMENT":

          break;
        default:

      }
    }
  }, [])
}
```

The `updateIndex` starts with a line, that initializes an `order` variable to 0. This will help us when we get to `ADDCOMMENT`.

We start by reducing the `oplog.values` Array. The accumulator of the reducer contains the hashes of the items that have already been handled. The current item is checked, not to have been handled already and is then added to the handled array.

After this, the handling actually starts. We use a switch statement to handle the item's differently based on the `op` value. As we discussed in the previous chapter, there are four operations, that this Index can handle:

- `ADDNOTES` to add a new piece of notes.
- `DELETENOTES` to delete a piece of notes.
- `ADDCOMMENT` to add a comment to a piece of notes or some other comments.
- `DELETECOMMENT` to delete a comment.

Implementing `ADDNOTES` handling Add notes is by far the simplest operation to handle, since we just need to add a new `TreeNode` to the `_index`.

```
case "ADDNOTES":
  this._index[item.hash] = new TreeNode(item.payload.value)

  break;
```

Implementing `DELETENOTES` handling And deleting notes is the inverse:

```
case "DELETENOTES":
  delete this._index[item.hash]

  break;
```

Implementing `ADDCOMMENT` handling Adding comments is a little more complicated. We first have to find the parent of the comment in the notes or among the comments themselves.

To do this properly in adequate time, we have to add a `_comments` property to our `Index` in the `constructor`:

```
constructor() {
  this._index = {}
  this._comments = {}
}
```

In this object, we store each comment's `TreeNode` by it's hash or rather the hash of the Oplog `Entry` that added them for easy access.

```
case "ADDCOMMENT":
  let reference = item.payload.key
  let node = {
    comment: item.payload.value,
    author: item.identity.id,
    id: order
  }
  order++

  if(this._index[item.payload.key] !== undefined) {
    node = this._index[item.payload.key].addChild(node)

  } else if(this._comments[item.payload.key] !== undefined){
    node = this._index[item.payload.key].addChild(node)
  } else {
    break;
  }

  this._comments[item.hash] = node

  break;
```

This branch of the `ADDCOMMENT` switch starts by utilizing the `key` field of the Operation, which is interpreted as the hash of the notes pieces or the comments. Then the node of the comment is created containing both the comment, an `author` field, and an `id` field. The `author` field is set to the ID of the OrbitDB instance that is passed to the operation and already verified by OrbitDB.

And for the `id` we use the `updateIndex`-wide `order` variable, which is incremented afterwards. Because the `order` variable is increased for each comment on each note. This makes it possible to sort the comments in

`getComments`.

After this, the `TreeNode` that is referred to by the Operator's `key` field, gets a new child in the form of the comment.

And at last, we store the created `TreeNode`, in `_comments` for later.

Implementing `DELETECOMMENT` handling After the monster of a branch above, this case is pretty relaxing in comparison:

```
case "DELETECOMMENT":
  let comment = item.payload.key
  delete this._comments[item.hash]

  break;
```

Conclusion

We have now defined the complete `Index` for the comment system. You can now read it through the `getNotes` and `getComments`.

But we haven't discussed two topics yet:

- How do you add data to the database?
- And how can you ensure, that the database isn't modified incorrectly? How can we ensure that user A doesn't delete the comment of user B?

Both of those question will be addressed in the next chapters of this Tutorial. First, in the Store chapters, we discuss, how you can add data to the database.

And in the chapters about the Access Controller we will at the end of this Tutorial discuss, how you can control, how can change what in your databases.

Next: Defining the Store

Defining the Store

We now have our own `Index` in `NotesIndex.js`.

But how can we use it? Through the `Store`.

The Store, as we know, actually does not store the state of the Database, because this is the job of the `Index`.

The function of the job is to provide an easy to use, intuitive API for the end-user, such that they do not have to bother about things, like CRDTs, `ipfs-log` or the `DAG`.

You already know a lot of stores, at least if you read the first Tutorial in this Manual.

Because there are five built-in stores, which will I not bother recounting here.

The `NotesStore` class

You should now add a new JavaScript file and call it `NotesStore.js`.

Isomorphic Bookends For the third time in this Manual: Let's define an isomorphic bookend:

```
function notesStore(IPFS, OrbitDB, NotesIndex) {

}

try {
  const IPFS = require("ipfs")
  const OrbitDB = require("orbit-db")
  const NotesIndex = require("./NotesIndex")

  module.exports = notesStore(IPFS, OrbitDB)
} catch (e) {
  console.log(e)
  window.NoteStore = notesStore(window.Ipfs, window.OrbitDB, window.NotesIndex)
}
```

We'll from now on be working inside the `notesStore` function.

Defining the `NotesStore` class

To define a Store, we have to extend an existing Store class. For this Tutorial, we shall use the `EventStore` class.

Add the following as the body of the `notesStore` function:

```
class NoteStore extends OrbitDB.EventStore {
  constructor(ipfs, id, dbname, options) {
    if(!options.Index) Object.assign(options, { Index: NotesIndex })

    super(ipfs, id, dbname, options)
    this._type = NoteStore.type
    this._ipfs = ipfs
  }

  static get type () {
    return "noteStore"
  }
}
```

```
return NoteStore
```

What is happening here? The Store receives four parameters in it's `constructor`:

- `ipfs` - the IPFS instance passed to OrbitDB.createInstance
- `id` - the id of this database
- `dbname` - the name of the database and last part of the orbitdb addresses.
- `options` - Object of options. Most important for our purposes is the `Index` option. It is the `Index` class, used by the database.

In the first line of the `constructor` the `Index` option is set to `NotesIndex`, unless the options already has an `Index` specified.

Besides this, all stores have to have a `type` property, to uniquely identify the `NotesStore`.

Defining a `getNotes` and `getComments`

We should probably implement a few get functions. But this is really low effort, since we already implemented a `getNotes` and `getComments` function on the Index and thus we can implement the `NotesStore.getNotes` and `NotesStore.getComments`:

```
getNotes(cid) {
  return this._index.getNotes(cid)
}

getComments(cid, flat = true) {
  return this._index.getComments(cid, flat = flat)
}
```

As you can see, `this._index` is an instance of the `NotesIndex`, if no other Index was passed in to the constructor, and we can easily access the methods we defined in the Index.

Adding Data to the Store

Now, let's finally add some data to our Store.

First, we should make it possible to add a Notes Piece by it's CID:

```
addNotesByCID(cid, mime, instrument = "piano", options = {}) {
  return this._addOperation({
    op: "ADDNOTES",
    key: null,
    value: {
      cid: cid,
      mime: mime,
      instrument: instrument
    }
  }, options)
}
```

The `NotesStore.addNotesByCID` method

As you can see, we use an `_addOperation` method, which adds an operation to the `ipfs-log` or `oplog`, which meets us again in the `updateIndex` method.

`_addOperation` returns a `Promise<CID>`, so a promise of the hash of the operation entry in the oplog.

And we also define a format for the notes:

```
value: {
  cid: cid,
  mime: mime,
  instrument: instrument
}
```

addNotesBinary

But we can still do more for the user in this store. `addNotesByCID` adds a notes file by it' CID. But this expects the notes file to have been added to IPFS, something that we can do automatically, since we stored the IPFS Node in the `this._ipfs` variable.

```
async addNotesBinary(binary, mime, instrument = "piano", options = {}) {
  let {cid} = await this._ipfs.add(binary)

  if(options.pin) await this._ipfs.pin.add(cid)

  return await this.addNotesByCID(cid.toString(), mime, instrument = instrument, options = options)
}
```

This method adds the `Uint8Array binary` to IPFS. It then pins the data to the local IPFS Node, if the `options.pin` boolean is `true`.

Afterwards, it calls the `addNotesByCID` function, with the generated `cid`.

This is just an example of how you can make the Store API encompass more, than just adding operations to the `oplog`.

For some other ideas of what can be done in the store, prior to adding an operation:

- Formatting (with Protobuf)
- Encryption and Decryption

Adding Comments After a user added their musical notes to their Database, users might want to add comments to the notes.

Let's add an `addComment` method.

```
addComment(text, reference) {
  if(this._index.getNotes(reference) !== undefined || this._index._comments[reference] !== undefined) {
     return this._addOperation({
       op: "ADDCOMMENT",
       key: reference,
       value: text
     })
  } else {
    return null
  }
}
```

You might observe, that this function performs some validation prior to creating the operation, ensuring that the reference actually exists. This is one way how you can use the store, too, but you might want to do this in the index instead.

By the way, all operations are signed by your OrbitDB Identity on creation and we thus don't need to add any further information about the auther. See the Implementing `ADDCOMMENT` handling section of the previous chapter.

Other Stores

You might note that this is a very complicated custom store and index. But the actual techniques used are deployed already in all of the built-in Stores.

So, if you want further examples of how you could implement your own custom stores, reading the source code of the `*Store.js` and `*Index.js` file can be very illuminating and inspiring.

I would advice reading the `EventIndex.js` and the KVStore's Store and Index files.

Key Takeaways

- Stores inherit from each other. So you can extend built-in stores.
- Stores work with the `Index` and the `this._addOperation` mostly.
- `this._addOperation` adds an operation to the oplog and you can specify the `payload` field of each operation here.
- Stores can also do more, than just reading and writing from the database, like:
  1. You can use them to implement custom encryption/decryption
  2. Format your data in specific ways (protobuf?)

Next: Conclusion

On Customizing OrbitDB

In conclusion this part of the Field Manual of OrbitDB tried to introduce the reader to customizing the OrbitDB Stores by defining `Index` and `Store` types.

It started with the `Index` to define the current working state of the database.

Here the tutorial was trying to introduce the parsing of the `oplog` - an Array of Operations - to create this state.

After the `Index` - state of the database - being defined, the tutorial defines a `Store` class.

The `Store` class does not manage the current state, but use the `Index` to create an easy to use API for a User, who just wants to use the Database and has no time to bother about CRDTs, `oplogs` and other such complexities.

Where to go from here?

The source code in this tutorial is incomplete by design. You can extend it, add more features, change the API or parse the state differently.

Some ideas, that you could pursue - but don't have to:

- Event Handling: Firing events when receiving *new* comments, notes and deleting old ones.
- Make it possible to edit comments.
- Use a different data structure to represent the comments: A lazy hash table, where each notes piece and comment can be looked up and parsed upon request, instead of upon receiving new entries.
- Use a different delete heuristic: Don't delete comments, if the comments or music sheets they refer to are deleted. Do something else! Maybe move them up the tree or put them into a `detached comments` array. It's a question of your imagination.

You are free to change the source code here however you want. If you want to compare the code, that you have written over the course of this Tutorial with the code, we used to create this tutorial, you can download the final version of the code here: final.

We also have an Appendix to this part of the tutorial, that describes how you can use the `AccessControllers` with OrbitDB to moderate the discussions about the notes.

If you want to read that, go to this: **Moderating your Comment Threads.**

Moderating your Comment Threads.

We have now gotten a store that you can publish notes and comments with.

But you cannot actually control who and what gets to comment and count as a valid comment.

In other words: How can you moderate a Peer-to-Peer Database?

Well, you can't really. Or at least, you cannot moderate data on another persons computer.

I cannot delete a file on your computer, anymore than you can delete a file on mine.

If either of us could do that, we would consider that malware.

And this is not a Tutorial on writing malware, so we'll not go into it.

Moderating is generally considered to be two separate tasks:

1. Restricting the writing to a database
2. Restricting the reading from a database and sharing of contents.

In a decentralized network, sharing data is easy. Once any peer has some data, they can share it essentially until their bandwidth runs out.

But writing control is more complex. We cannot prevent somebody from writing to their own local database, but we can decide, whether we will accept the local changes from another peer in our own database.

And there OrbitDB `AccessController`s come into action.

They are invoked when an OrbitDB instance receives new entries for a specific database and they determine, whether the OrbitDB Instance should accept and use these entries or deny and trash them.

Remember: These rules, that you write into the AccessController can be changed or ignored by other peers, if they so wish. But then again, consider whether it's important to you what other peers do with their own database, as long as your database is clean and conforming to all the rules you laid out?

Additionally, if most peers follow your rules, then most content that violates these rules will not persist anyway, because nobody is around to pin it. Although in some cases for some specific data, somebody might be really insistent and pin it anyway.

Next: Implementing a custom AccessController

Implementing a custom Access controllers

And after these things, let us now consider how we might implement a custom access controller.

Starting with the rules, that we want to implement:

- The creator should be the only one to upload notes pieces.
- Anyone can comment by default.
- The author of a comment and the creator can delete a comment.
- The creator can ban somebody from commenting on the entire database.

Implementation

Similarly to the `Stores`, Access Controllers are implemented using inherited classes. Let's implement a `NotesAccessController.js` file with these Isomorphic bookends:

Note: TBD

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.

A small note: We don't have entries for abbreviations, such as IPFS for Interplanetary File System. Use your native search function to find the relevant entries.

Terms to use

- **Conflict-free Replicated Data types (CRDTs)**: In distributed computing, a conflict-free replicated data type (CRDT) is a data structure which can be replicated across multiple computers in a network, where the replicas can be updated independently and concurrently without coordination between the replicas, and where it is always mathematically possible to resolve inconsistencies which might result. Wikipedia
- **database**:
- **datastore**:
- **eventual consistency**: Operations can be taking place at places and times that you are unaware of, with the assumption that you'll eventually connect with peers, share your logs, and sync your data. This contrasts with Blockchain's idea of *strong consistency* where entries are added to the database only after they have been verified by some distributed consensus algorithm.
- **Filecoin**:
- **InterPlanetary File System (IPFS)**:
- **isomorphic**:
- **js-ipfs**: The JavaScript implementation of IPFS. GitHub.
- **packet switching**: Packet switching is a method of grouping data that is transmitted over a digital network into packets. Packets are made of a header and a payload. Data in the header are used by networking hardware to direct the packet to its destination where the payload is extracted and used by application software. Packet switching is the primary basis for data communications in computer networks worldwide. Wikipedia.
- **Paxos**: Paxos is a family of protocols for solving consensus in a network of unreliable processors (that is, processors that may fail). Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communication medium may experience failures. Wikipedia.
- **strong consistency**: In this structural process, entries are added to a database only after they have been verified by some distributed consensus algorithm.