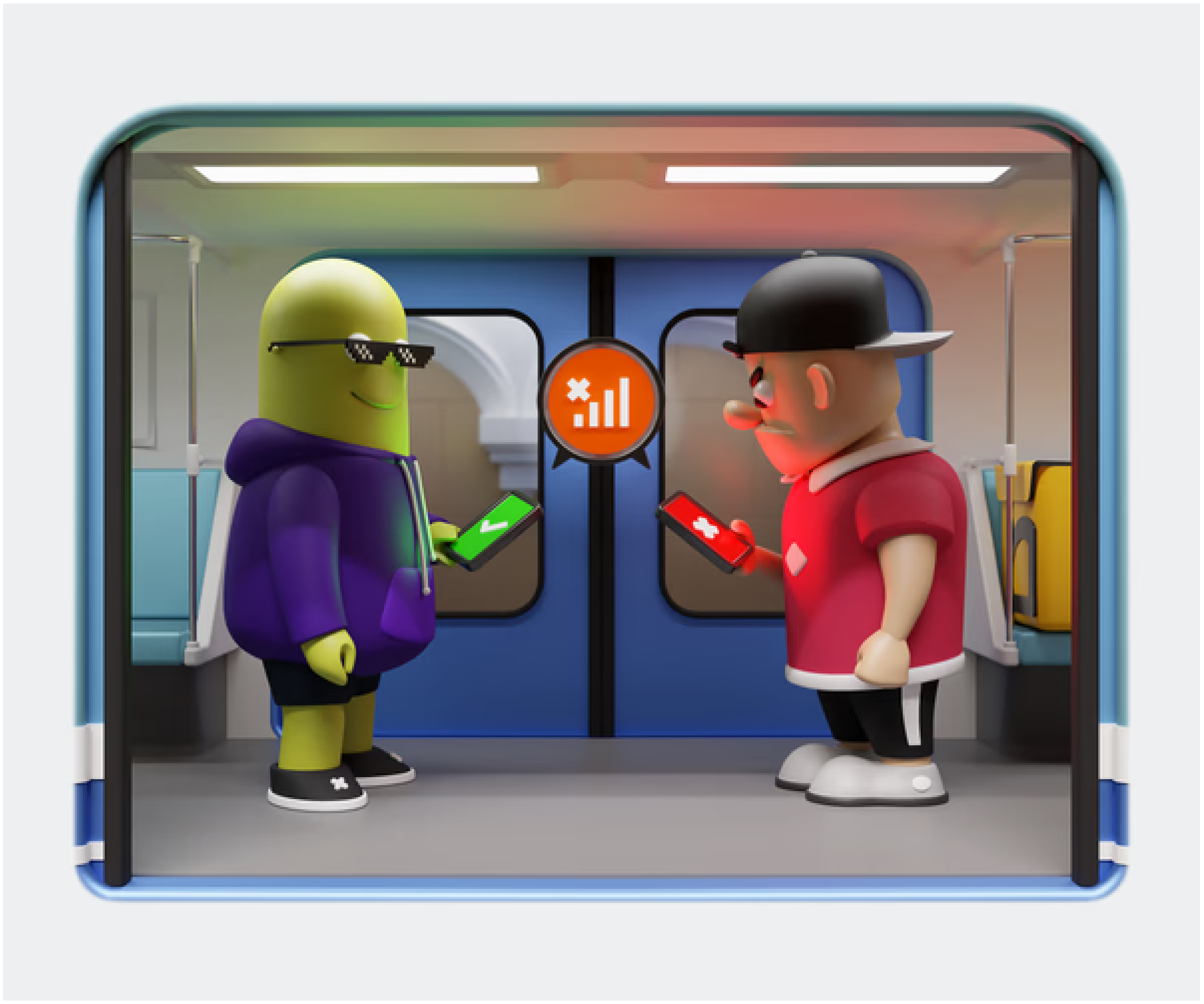
Cool frontend arts of local-first: storage, sync, conflicts

March 15, 2023



Topics

Frontend Design Full Cycle Software Development Product Launch CRDT JavaScript Lean Software Development

Share this post on

Twitter Facebook LinkedIn

Pavel Grinchenko
Frontend Engineer

Travis Turner
Tech Editor

Years ago, apps were offline only. Nowadays, they're mostly dependent on internet connections, but some allow you to work offline. When developing a "local-first" app, even if it relies on data stored online, it should still work. But, for frontend engineers, when it comes to the big 3 "local-first" tasks: storage, synchronization, and conflict resolution, actually implementing them with grace is almost like a fine art (or even a lost art, in some cases). Let's go digging.

We helped develop **HTTPie**, an API testing tool who wanted to be the canonical app in their field. Read that "local-first" success:

Read also

UI design for HTTPie: macOS vibes for the API testing client

January 30, 2023

But, before we get to those big 3 tasks, let's take a step back and paint a picture of the full context.

Within their niche, every developer has one or two "perfect" applications they've pretty much mentally canonized as "the cream of the crop" inside their field. Take your pick, **Linear**, **Figma**, whatever. (And, I'd say, most engineers place the same few collections of apps in their mental canons.)

So, here's a natural question: what exactly is it that you like about [**insert your canonized app here**]? For me, it's all about how we communicate with an app, in particular, the data inside it.

> It's not so important for end users whether their data is saved on the server immediately, or 10 seconds after connection has re-stabilized. After all, connection loss isn't the rarest problem on mobile connections, even in modern megalopolises.

Let me elaborate: when it comes to direct data editing, I vastly prefer as little intermediary steps as possible to get the job done: users can see their data change instantly, there aren't infinite levels of modal windows to get lost in, and, in cases where users have a poor internet connection, the end user's work should not be affected. That last point brings us to the **local-first** approach to product development.

What is a "local-first" app?

The traditional approach to data processing in a network application works like this: first, all the data is stored on a server. Second, the client application loads only the data necessary to open a particular page.

We've probably all experienced those annoying "something is wrong with your connection, please try again later" errors. Super annoying!

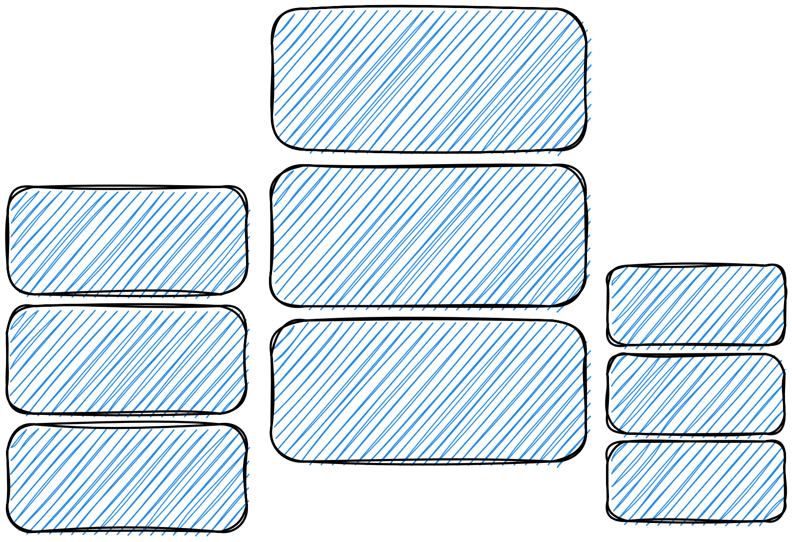
With local-first apps, the main idea is that users should be able to work effectively with an application, even if they're offline.

Here are the important takeaways: technically speaking, the local-first approach implies that the user's data and work must exist both on servers and clients. Users should be able to continue working with a local-first app even if it relies on data stored online. This requires that these local-first applications are engineered in such a way so that when an internet connection becomes available again, the user's local data must be then synchronized with the server.

> Engineers sometimes try to solve this using an Optimistic UI approach, but while, on the UI side, users often see their operations successfully completed, eventually, some error, like a connection drop, will cause them to see a "rolled-back interface".

With all this in mind, let's tackle the big 3 tasks for engineers working on an application where "local-first" features are a requirement. Those are: **storage**, **synchronization**, and **conflict resolution**.

Storage



Currently, web developers have access to several storage types: local storage, session storage, and IndexedDB, to name a few. Session storage and local storage have a 5MB limit, and, thus, are clearly unsuitable for any serious endeavor involving storing user data. So, let's talk about IndexedDB. It's crucial to understand that IndexedDB is asynchronous storage, and with some frameworks, this can cause integration problems.

> There are experiments to bring SQLite (**sql.js** and **absurd-sql**) and PostgreSQL (**PostgreSQL in WASM**) to the browser, but they look far from production-ready status. Still, if you're developing an Electron-only web app, you could consider them as an alternative solution.
>
> Read more about integration pitfalls:
>
> Read also
>
> **How to avoid tricky async state manager pitfalls in React**
> February 21, 2023

Let's talk about these integration problems: when you add new functionality, you often need to change the data storage scheme. But simply changing the schema is not so easily done, is it? First of all, you have to take care of the already existing data. This means you'll need to invent a data migration mechanism. (This is not only the case with IndexedDB, but all storage systems).

So, we have to store data, and we have to interact with it. Clear enough. But working within a distributed environment (like the little world of a user's local data and the "bigger" server data, or even a peer-to-peer environment with no server) imposes some limitations in this field.

(And by the way, yes, sometimes we'll use several storages simultaneously, as is typical when developing standard web applications.)

Storage: record collection

Each local client itself is responsible for record creation and, accordingly, must generate record IDs. With the local-first approach, ordered IDs aren't a fit due to collisions between different clients.

The most typical method is to use v4 UUIDs. Although there is still the possibility of collisions, this is relatively small.

For related data, or data that needs a consistent ID based on any attribute with a constant value, it's convenient to use v5 UUIDs, since we can provide a UUID namespace and input string and get a non-random, unique ID, a deterministic UUID.

Storage: deletion

In classic applications, in most cases, we can simply delete data from the database as needed. But in the case of a distributed system, we shouldn't do this.

> By the way, for some soft-deletion fun, check this post:
>
> Read also

Why? Well, we could face a situation where one internet-connected (although, they could be offline, as well) user deletes a post, and meanwhile, another user will have also edited the same post using the offline version of the app.

This is a problem. During synchronization, it will be impossible to tell if this record was “really deleted”, or even if it originally existed in the first place, and now the user is trying to update a non-existent record. Therefore, we recommend using soft-deletion and just marking the data as deleted.

Storage: dealing with inconsistent data

Most applications use relations between records. But in a local-first application, the existing ID of a related record doesn’t mean this record has already been synced with your client and is available for use. Perhaps, in the synchronization process, you’ll receive only the main record with the first batch, and the related records will be downloaded in the next batches. Your application should handle these cases correctly.

If you have ordered data in your application and users can change that order, then you shouldn’t use classic indexes for controlling element positions. It simply won’t work if multiple users simultaneously decide to change the record orders. Moreover, naive solutions increase DB write counts, and therefore, items to sync.

To solve this problem, there is an elegant and simple solution: **fractional-indices**. The index of a new position is calculated based on the indices of its new neighbors, while the index of other records does not change. At the same time, sorting itself continues to work as before. Let's take a brief look at how it works.

With the regular approach:

```
const items = [
 { id: 'a', sort: 1 },
 { id: 'b', sort: 2 },
];

// Insert item in the middle
const items = [
 { id: 'a', sort: 1 },
 { id: 'c', sort: 2 }, // New item
 { id: 'b', sort: 3 }, // Index was changed to 3
]

// Insert item at the start of array
const items = [
 { id: 'd', sort: 1 }, // New item
 { id: 'a', sort: 2 }, // Index was changed to 2
 { id: 'c', sort: 3 }, // Index was changed to 3
 { id: 'b', sort: 4 }, // Index was changed to 4
]
```

With fractional indices:

```
const items = [
 { id: 'a', sort: 1 },
 { id: 'b', sort: 2 }
];

// Insert item in the middle
const items = [
 { id: 'a', sort: 1 },
 { id: 'c': sort: 1.5 }, // New item
 { id: 'b': sort: 2 }
]

// Insert item at the start of array
const items = [
 { id: 'd', sort: 0.5 }, // New item
 { id: 'a', sort: 1 },
 { id: 'c': sort: 1.5 },
 { id: 'b': sort: 2 },
]
```

Note what items have changed during inserts in both examples! You can also read **this article on reordering fractional indices** or even use a ready-made library.

Synchronization

For local-first apps, the synchronization process is one of the most critical factors to deal with because it directly affects the user experience. To effectively implement synchronization, we must take into account many nuances: for instance, the version of data a user currently has, what changes they have made since the last synchronization, whether these need to be synchronized with the server, and so on.

To simplify a bit, synchronization usually comes down to 2 operations: Pull and Push.

Pull is responsible for getting new changes from the server while Push sends local changes to the server. But the data they send in these operations can be presented differently.

In the most simple case, push can send a completely updated model: in this case, other users' changes in other fields of the same model will be overwritten. It also unnecessarily enlarges traffic between server and client. (This approach only fits some very limited cases.) This problem can be solved by using the most atomized models, but in terms of storing and performing operations with data, since this atomized approach means you need to store every field independently, this option is not very convenient.

Sending atomic changes from a client is the more convenient way. We can send only the model's ID and its updated fields. This allows storing data in the usual way, and it reduces conflicts that may arise when users are working on the same record at the same time. If users have changed different fields in the same record, they will not have any conflicts, and both fields will be saved correctly.

We can also send operations instead of changed data, instead of sending changed data. For example, instead of `{ id: 1, amount: 42 }`, the operation might look like this:

```
// let's suggest the original value was 41
{ target: `event[1].amount`, op: { type: 'add', value: 1 } }
```

Or, instead of `{ id: 1, address: "Lisboa, Portugal" }`, it looks like this:

```
// let's suggest the original value was "Lisbon"
{ target: `location[1].address`, op: { type: 'insert', at: 6, value: ', Portugal' } }
```

At first glance, the "operations" variant looks redundant, but it has its advantages. The client can keep a log of such operations and consistently apply them over existing data. This is similar to the "rebase" operation in Git, where we can apply commits one by one over a new base commit. In our case, the new “base” is fresh data pulled up from the server, and “commits” are the operation log.

By adjusting the operations’ atomicity, you can reduce possible conflicts. E.g., in the example above, when we edit a text field, we do not entirely rewrite it but insert a value into an already existing text according to a specific index.

To implement this approach, the data must be **designed in a special way**. The most famous implementation for managing such data is the **automerge** library, and here’s a more **in-depth video on automerge**.

Combined options are also possible. For example, operations can be sent with Push, and we can request patches for data again with Pull.

Examples of synchronization implementations:

- **Example with WatermelonDB**
- **Example with Replicache**
- **Example with RxDB**

By the way, we have our own Evil Martians solution for collaborative apps, **Logux**.

Conflict resolution

Anyway you slice it, we will inevitably have data conflicts. The frequency of these conflicts directly depends on how your app is used, but you shouldn't rely on the fact that, even if users only work with their own data, it won't trigger conflicts somewhere.

For instance, your user might work from different devices, which can lead to "change" conflicts. Additionally, the app's communication model is also critical: do you have an Authoritative server, or are you building a fully distributed system (P2P)? In the first case, you can assign the responsibility for the conflict resolution strategy to your server as a single source of truth. But in this scheme, conflicts can also be resolved on a client:

- **WatermelonDB docs on conflict resolution**
- **Replicache's vision on conflict resolution**
- **RxDB's docs on handling conflicts**

Your conflict resolution model is tightly intertwined with the model of interaction between your users: both in terms of their interaction with your application and with each other. In some cases, last-write wins at the record field level will be enough; in others, we strongly need a full-fledged CRDT.

And with that, we've considered the three main pillars upon which your application will rely on: storage, synchronization, and conflict resolution. But that's not all.

What else should we care about?

It's important to consider the following: because of an app's distributed nature, users might actually end up working with different versions or different releases of the same app simultaneously. For instance, this could be the web version of an application versus an Electron application.

And, if we consider offline work, in some cases, it's possible that a user hasn't connected for so long that the data structure or server API had been changed during that time. And that's a problem we need to solve.

So, before we synchronize the data, we need to update the application. For a typical web application, a browser tab refresh is sufficient, but with a desktop Electron app, you should incorporate an automatic update system; otherwise, you may lose a significant number of users who simply will get tired of the need to manually update their apps. Today's most comprehensive tool for building Electron applications is the electron-builder.

Multiple tabs

If your application can be used in multiple windows, you'll likely encounter problems with leader determination and cross-window communication. Why is this so important to consider? Some services should only work in the leader:

- We have to have just one instance of the database for an application, meaning its initialization must occur just once.
- When updating the data structure, you need to migrate existing data, and it makes no sense to do this several times in all windows.
- Data synchronization is best done centrally from the main application instance.
- It's also better to manage authorization from one place. But at the same time, you need to understand that all other app instances must contain the current state of the application, so you need to synchronize this state somehow between windows.

With regards to the last point, the modern web platform provides some APIs to solve these problems:

- For window-to-window interaction, there is **BroadcastChannel** which boasts good support in browsers.
- Defining a leader in an asynchronous environment can be a non-trivial task. Luckily, there is the **Web Locks API**, an API with good browser support.

Note: There is an alternative approach with just one instance of main logic using SharedWorker and communicating with it from all tabs. This has its pros and cons. It makes it unnecessary to choose a leader and doesn't block the main thread. The main disadvantage is limited browser support.

Here's an example of determining a leader using the Web Locks API (the code author is **Ivan Buryak**):

```
async function awaitLeadership() {
  const NEVER = new Promise(() => {})

  return new Promise((resolve) => {
    window.navigator.locks.request('isLeader', () => {
      resolve()
      return NEVER
    })
  })
}

awaitLeadership().then(() => {
  console.log('Hello, I am leader')
})
```

The benefits of the local-first approach

In this post, I covered many of the pitfalls you need to avoid if you're developing a local-first application. But despite these tricky cases, the local database model also has a number of advantages over the classic client-server model.

Since you update the data locally, you can immediately display UI changes. In the regular model, you are forced to implement a long request indication (like a spinner) for users to understand that the application isn't broken, it's just busy working. With some backend APIs you have to handle all possible API errors per each request individually. In local-first applications, you have a centralized sync protocol for this purpose.

As mentioned elsewhere, the Optimistic UI approach partly solves this, but you need to remember that the network may disconnect or a server may return an error and then you need to roll back the changes correctly, and this is also some additional work—sometimes, for each separate operation.

If the application needs to receive updates from the server, then usually you have to implement this process separately via persistent connections (Server-sent events or WebSocket). With synchronization implemented, you can simply update the state on the server and it will automatically be synchronized with the client, and vice versa.

Finally, beyond UI/UX we have less stress on the server (there's no need to perform a request on every change, as we can buffer logs. And, for frontend-centric teams, there's no need to build a ton of backend stuff, APIs, etc.

Out-of-the-box solutions

At the moment, there are a lot of toolkits for building local-first applications. The most popular, which we've already met, are:

- **Replicache**
- **RxDB**
- **Watermelon**

And, of course, we also have **Logux**, the aforementioned Evil Martians open source project. There and some others, like **Dexie**, and even **Google Firebase has something related to local-first apps**.

I'd highlight that these ready-made solutions don't mean you won't have to modify them to fit your own needs, so keep this in mind, should you opt for this route.

In the case of a local-first application, we have to deal with a transfer of complexity from the server to the client, and the server itself often becomes "skinny." With this approach, the model of interaction with the server is completely changed and simplified, and application development starts to look like backend development, as we have a database, and a schema, validations, and migrations coming along.

Not every single application requires a local-first approach because this approach definitely complicates the application's development. But at the same time, the benefits it provides can become a killer feature of your application because users can work with it more comfortably. This choice is up to you.

One more thing: Evil Martians are here and we're ready to solve your problems: frontend, product design, backend, devops or beyond, no matter you're local or across the globe, we're here! **Get in contact now** for more info!

In the same orbit

Client **Developer Tools** **Product Launch** **Design**

HTTPie is an open-source API testing client. The team called on Evil Martians to create the Web & Desktop version of the application. Though we started by building a complex professional UI design, we also helped add new features on the frontend and backend.

$6.5M
total funding

Backed
by Coatue

Product Software Development Frontend CRDT WebSocket JavaScript

Logux

A new way to connect clients and server. Instead of sending HTTP requests (AJAX/REST), it synchronizes the log of operations between client, server, and other clients through WebSockets.

Explore more posts

Frontend React JavaScript

How to avoid tricky async state manager pitfalls in React

February 21, 2023

Frontend CSS

How to Favicon in 2023: Six files that fit most needs

February 6, 2023

Frontend Design Case Study JavaScript React Gatsby

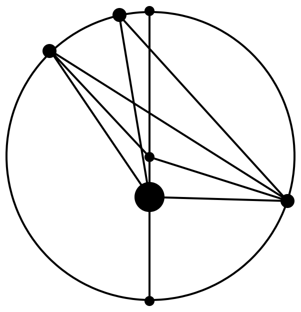
How to build a better React map with Pigeon Maps and Mapbox

January 24, 2023

Design Machine Learning Neural Networks

Midjourney vs. human illustrators: has AI already won?

December 13, 2022



Contact us

We’d love to hear from you! We’re not really all that evil, and we love discussing potential projects, intriguing ideas, and new opportunities. Complete the form below or drop us a line at **surrender@evilmartians.com**.

How can we help you?

Your name and company

Email

Submit

United States

+1 888 400 5485

195 Montague St.
Brooklyn, New York
11201

Portugal

+351 308 808 570

Rua Alexandre Oneill, 38,
Porto
4400–008

Japan

+81 6 6225 1242

9F Edobori Center Building, 2–1–1 Edobori, Nishi-ku,

Osaka
550–0002

Join our email newsletter

Get all the new posts delivered directly to your inbox. Unsubscribe anytime.

Your email

Subscribe

Or subscribe to a feed