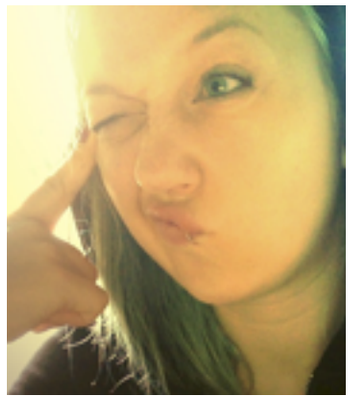
Say Hello To Offline First

How to build applications for
the real world

we <3 GOTOBerlin

I **hope**, you had a **great morning** and are not **too** tired!
Everyone got some **coffee** or **mate**?

Ola Gasidlo



- Developer for +10 years
- Core Member of Hoodie
- Co-Organizer of OTSConf && RejectJS

twitter: @misprintedtype

My **name** is…

Thank you for **having** me. I am really **excited** for the opportunity to **speak** today.

Today I'll tell you:

1. **What** the offline first **concept** is and **why** we need it in our **applications** so **badly**.
2. **Where** the problems **occur** on the **client** and the **server** side.
3. **How** we, as **developers** have to **rethink** our application **logic**
4. And I will **show** you **one** way to **implement** the concept

What & why?

@misprintedtype

Offline first is the **new** cool **kid** in **town**.

But **what** does offline first actually **mean** and **why** do we **need** it?

The internet turned **25** last year!

@misprintedtype

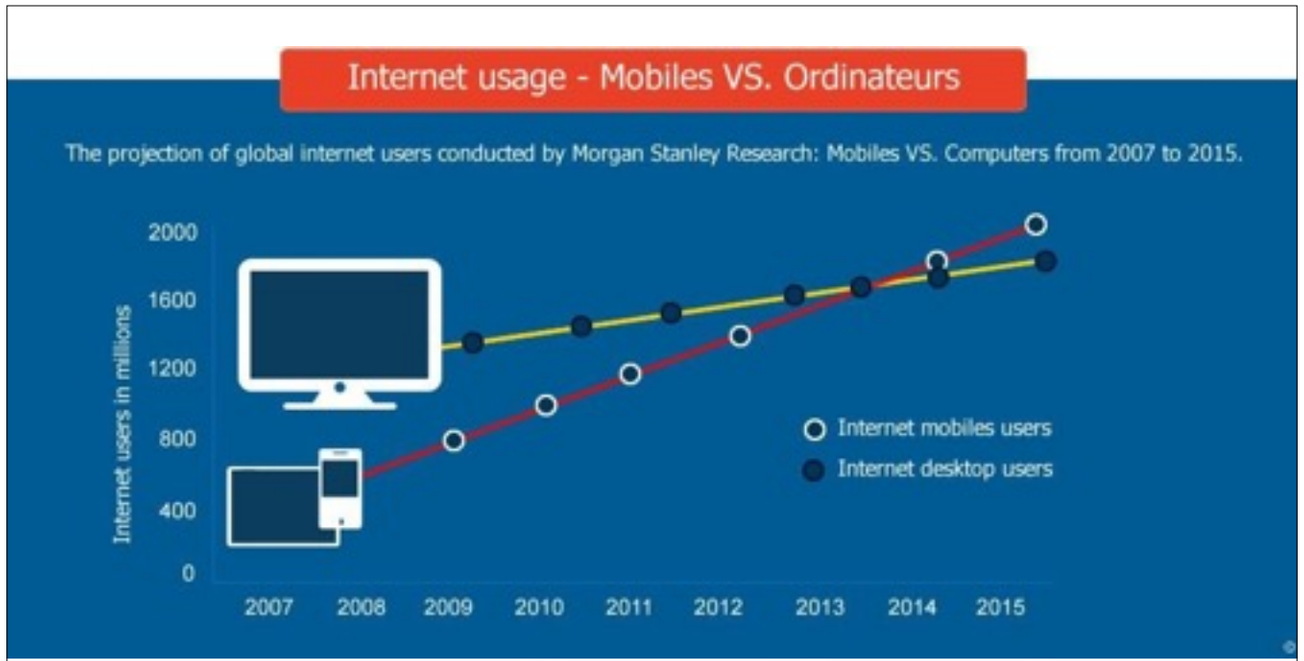
The internet turned **25 last** year.

In this time, our **relationship** with the **internet changed**. A lot.

We do **different things**, with **different devices** than 25, 10, **5 years ago**... and it is a part of our **everyday** life!

Who used their **phone today** for more than 5 different **tasks** today?
More than 10?
More than 15?

We rely on our **devices**... the **internet**, but our **needs** changed.



Like you see, we have **reached** the turning **point**!

Since **last** year, there are **more** mobile than **desktop** users out there.

amount of mobile-only web users is out of reach

@misprintedtype

In some **countries** there already are **mobile-only** users, like in **egypt, we have** 70%.
Just to make **sure**, **no** desktop device usage here **at all**!

This might **surprise** you, but **think** about it!
Mobile devices are much **cheaper** than a laptop for example.

For **many** people, this is **enough**. **Considering** what they are **using** their **device** for.

Tell me...

@misprintedtype

Some quick questions!

Who ever tried to **work online** on a **train** with a hotspot? On a **plane**?
How well did that **go**?

Who of you ever did a **screenshot** of an app, because they were **scared** of **losing their data**?

Who of you ever **stood** in a **supermarket** and try to **ask** someone what they should **buy**, but **realized** you are **offline**?

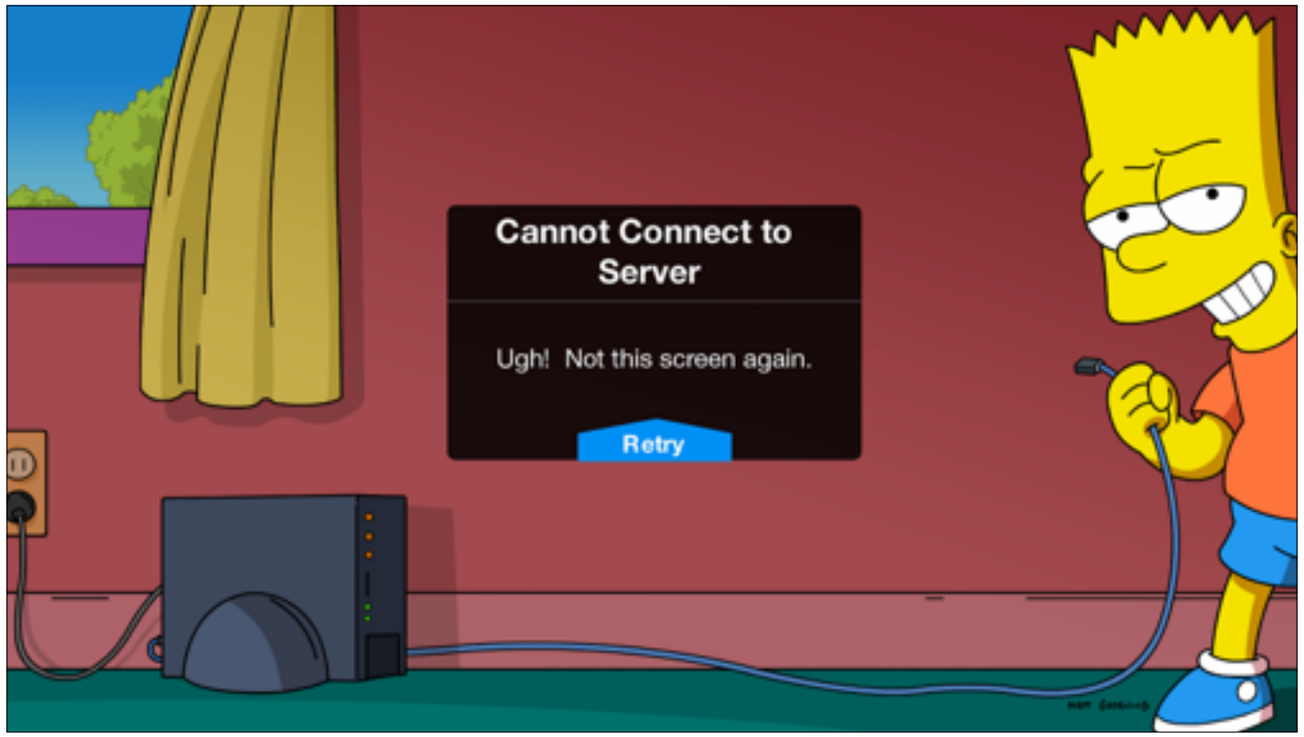
What if your **traffic limit** is **reached**? Your **Hotel WIFI** is way too **expensive** or the **roaming** cost is an **extortionist**?

Those are **issues** we **all know** even from the **wealthy countries** many of us **live** in.

@misprintedtype

Today our **work**, **social interactions**, our **lives** are **build** on **being online**.
If you are **not online**, it seems like you do not **exist**!

We need **new ways** to **deal** with these **issues** and **be able** to **stay a part** of the online **community**, stay in **touch** with our **friends and family**.



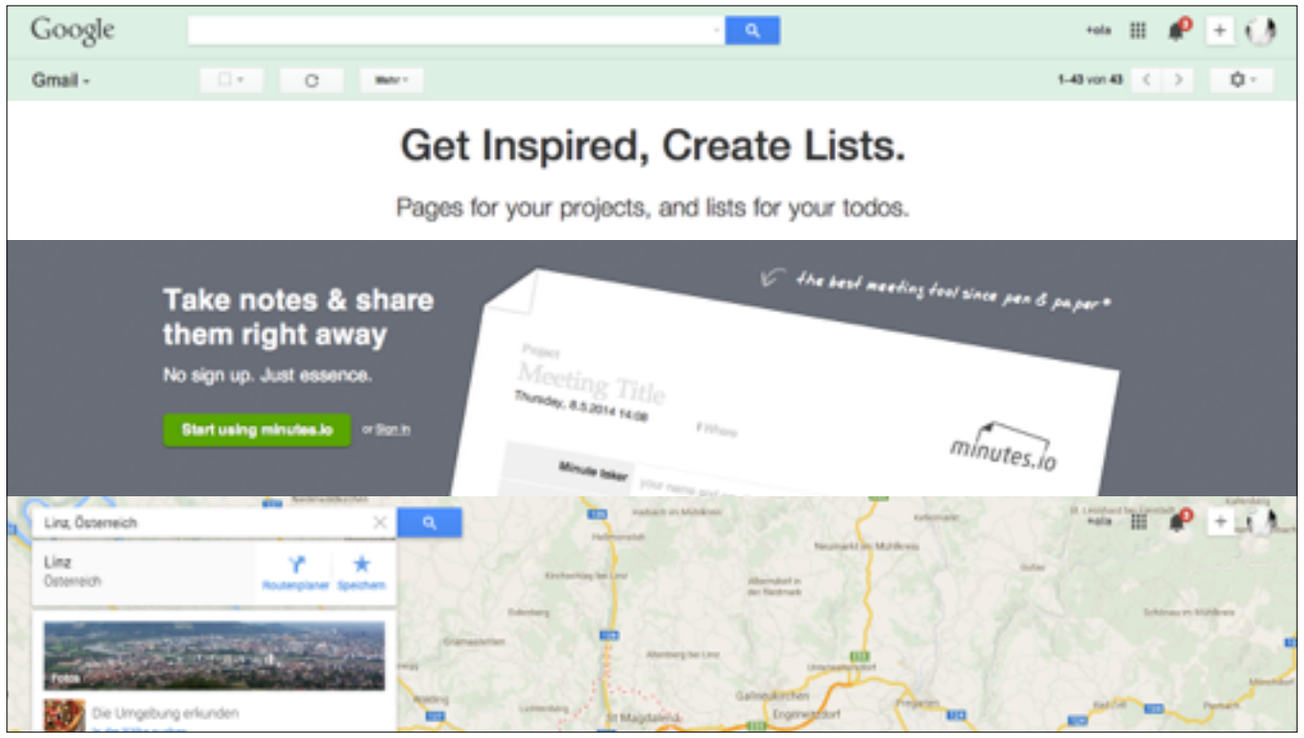
And the **situation** is way **worse** in **rural areas** and countries with **poor** or even **no network coverage**.

Zero reception, **unreliable** and **inconsistent connections** are still an **issue**, no matter where we go.

We need to **accept**, the technology has **reached** its **limits**.

Go **away** from the **desktop mindset** of **permanent**, **fast** connectivity by **accepting** being offline is not a bug, but a **reoccuring state**.

Let's **start** and **design** applications with **keeping** offline first in **mind**!



There are **already** some great **offline first applications** and **services** out there!

Starting with **email apps**, which **send** the emails we write, for us when we are **online** again.

Writing down **notes** during **meetings** with http://minutes.io or **creating** all lists we need with https://createlists.net/ **help** us stay productive **without** sending **error** messages as soon as our internet **connection is lost**.

Sending pictures, tweets, messages **during** a **concert** can be tricky when **everyone** is doing it in this **exact** time! applications like twitter **store** the message and it's at least not lost. Great! **Why** not **sending** it, when you are **online**? **Whatsapp sync works here pretty well!** At least on **Android**.

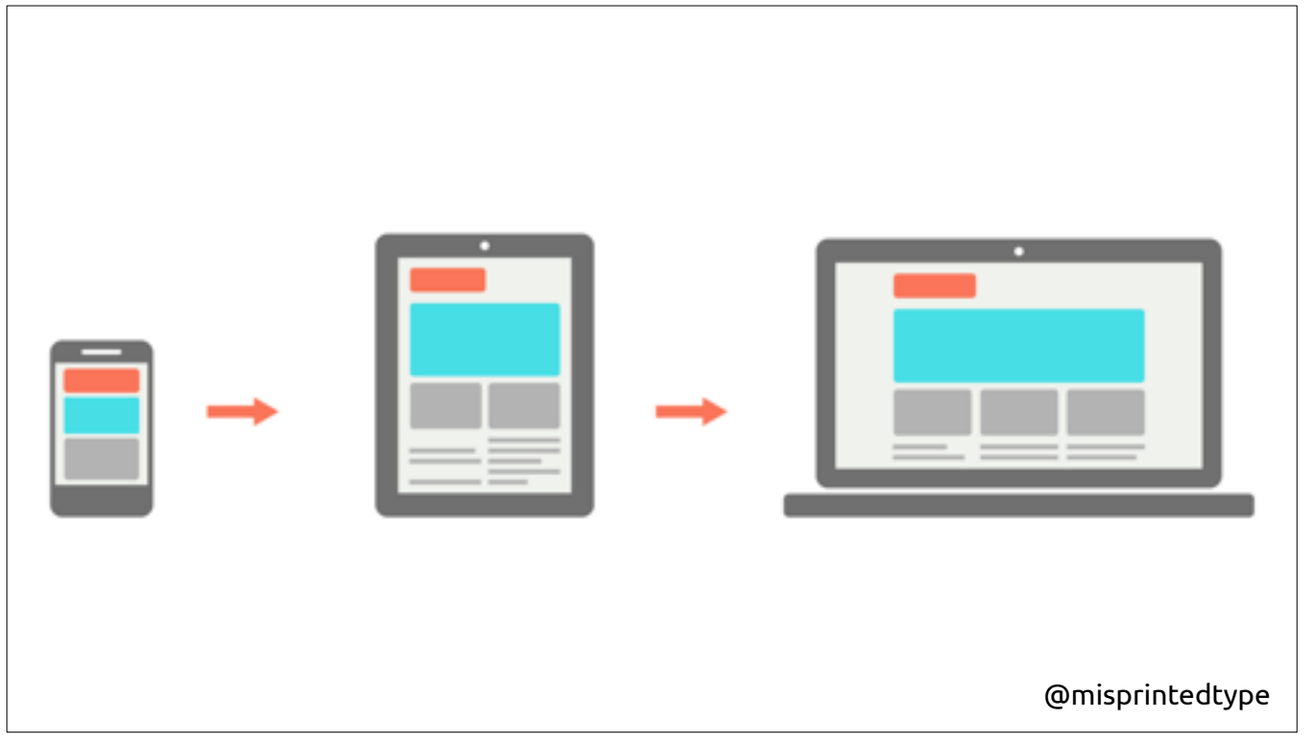
I see **amazing** apps in the **future**!

Imagine… You are in an **area** you've **never** been before? GREAT! I'm sure, you'd love to **explore** that, right? Wouldn't it be great if your **map** would realize **where** you are and just **download** the **map** for you when you have WIFi and so make it **available offline**? Showing you the **sightseeing points,** the next cafe with wifi, the nearby ATM… Help you to **feel** like almost at **home**!

Problems

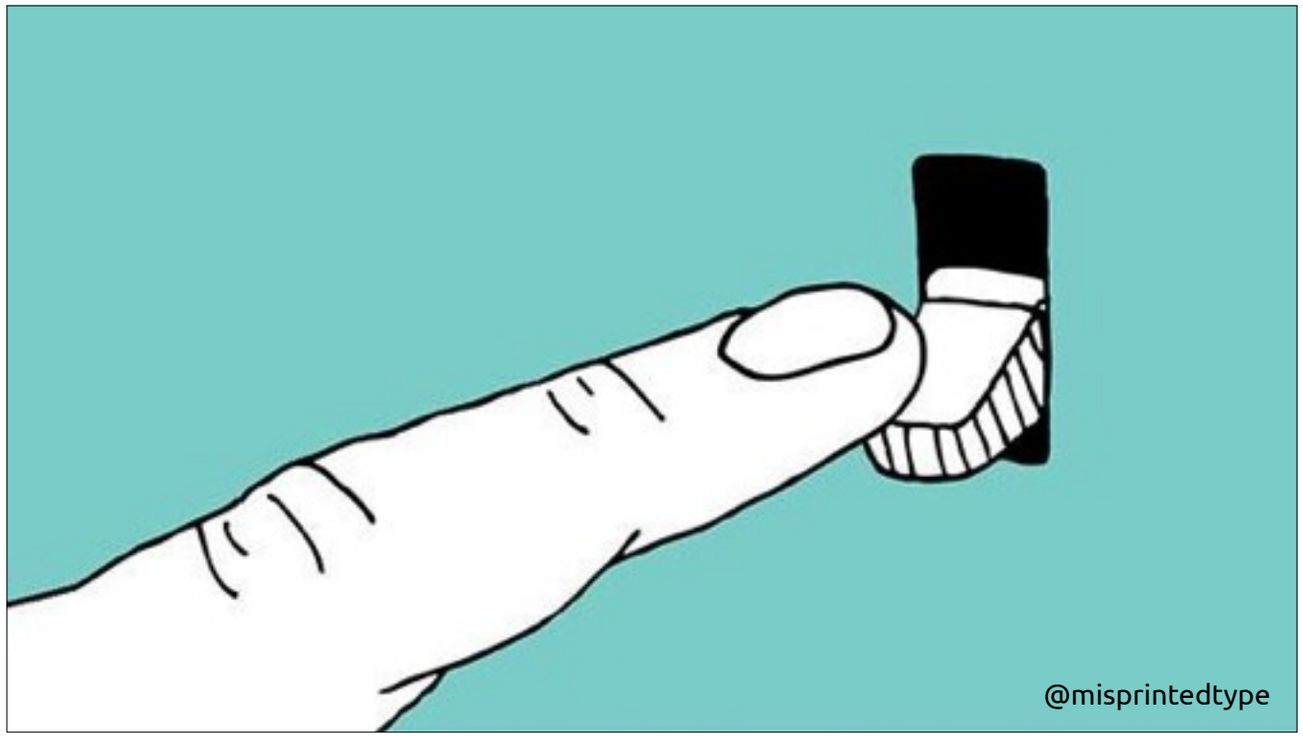
@misprintedtype

This **sounds** great! But **what** are the **real** problems **here**?



Let's **compare**!

Mobile First is a **design strategy** to help cope with the huge **variety** of devices and **capabilities**, like you see here!



In the **same way**, Offline First is a **design strategy** to help **cope** with the **unknowable circumstances** and **connection states** our users may **find** themselves **in**.

Offline First is simply an **acknowledgement** that this **lack** of **certainty** extends a bit **further** than we previously thought.

With an **offline mode** that isn't only a **contingency** for an **error scenario**, but a **fundamentally** more **flexible** and **fault-tolerant way** of application design.

offline > error handling

@misprintedtype

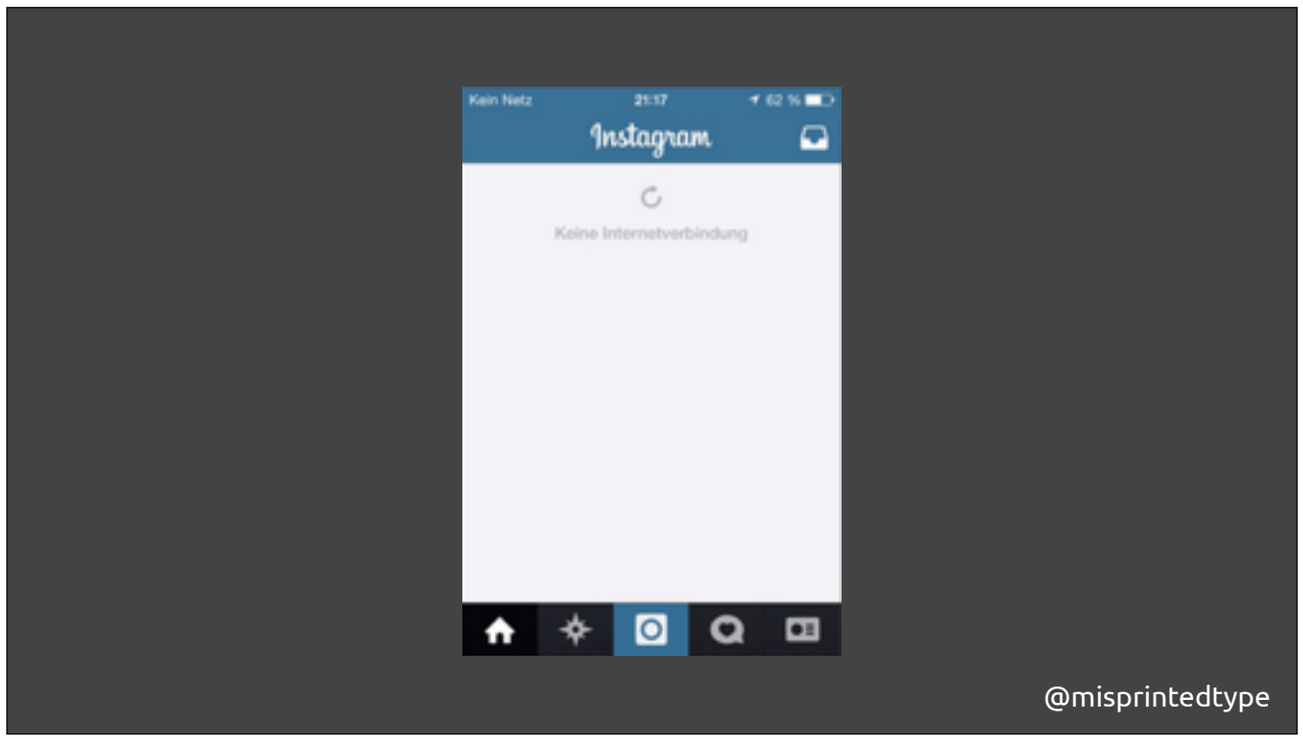
Offline is **more** than **error** handling! It's about **accepting** the reoccuring state!

inform user

@misprintedtype

We have to **ask** ourselves:

1. Does the app even **need to inform** the human of the **current connection state**?
Is this information even **relevant** to the human? If so, **how** can this **best** be **done**?

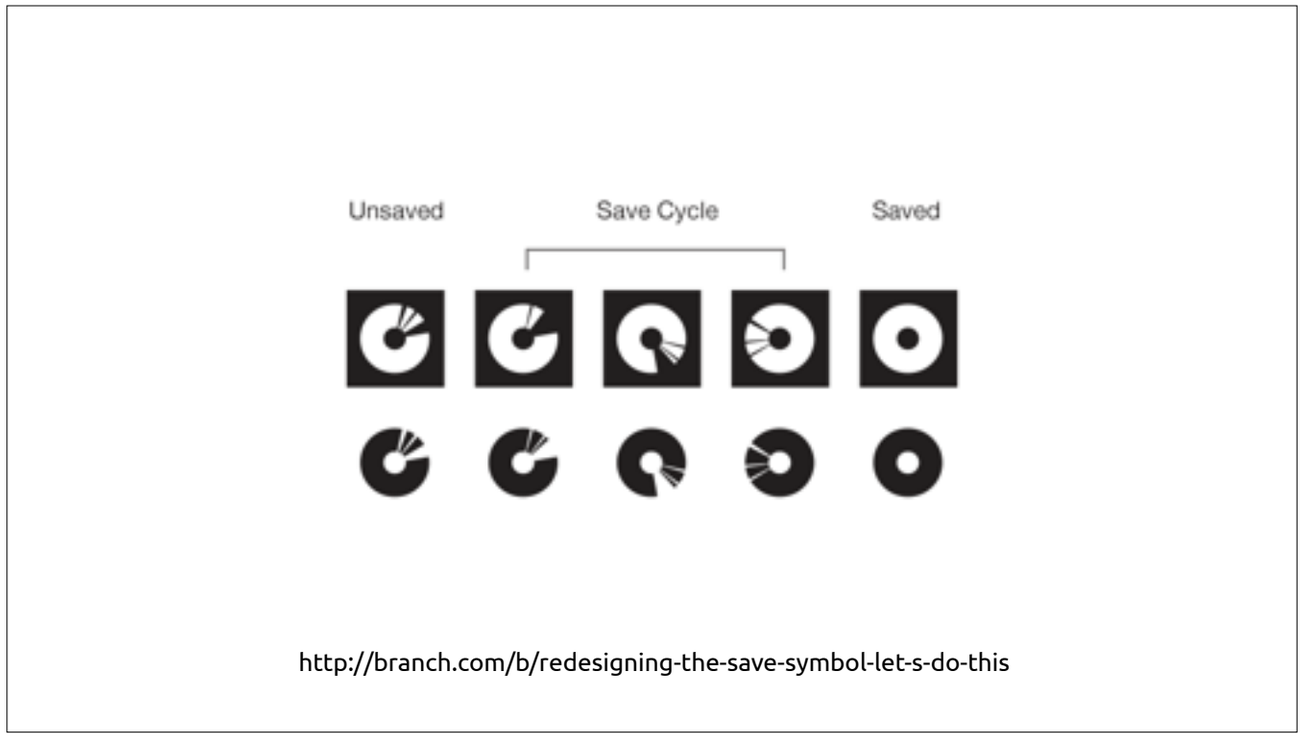


How can **THIS** be a **good** practice!?

Why not **showing** the user the **data**, we saved in the **cache** or **stored** locally?
All data can be **valuable** to the user, even **old** data.

2. Can I give the users more **trust** in the app, leaving them **safe** in the **knowledge** that it won't forget the last **state** it was in, and that it won't lose any data, regardless of what the **connectivity situation** is? So they do not need to **think** about making **screenshots** of their apps anymore?

3. How can I **communicate** to the **human** that **creating** data within the app is still **possible** offline, and that it will be posted/dispatched/properly **dealt** with in the future?



http://branch.com/b/redesigning-the-save-symbol-let-s-do-this

This is **one** of the **new concepts** for a **save icon** I really love.

There is the **unsaved** state, the **saving** cycle with **several** options and the **saved** state.

http://branch.com/b/redesigning-the-save-symbol-let-s-do-this

This is how it **would work**. You can see the icon in the **upper right** corner.

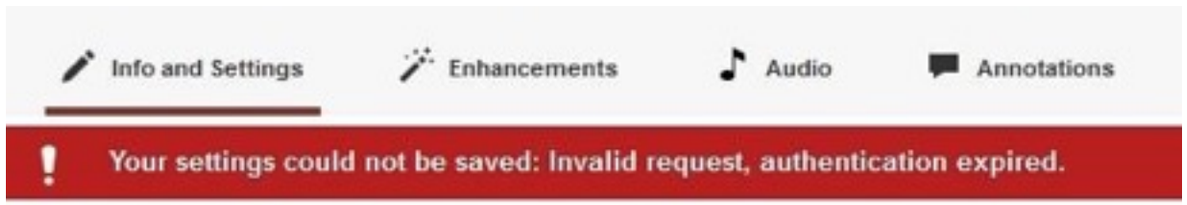
The user **types** something, the state changes to **unchanged**. Then it gets **updated** and it is **saved** again automatically.

This **really** looks like a **good** practice.

4. **How** can my **interface** convey **changes** that occur in the users **current view** when they **reconnect** and the server **pushes** new and changed data?

What to do with **deleted** items, things that **can’t be organised in lists**, objects that aren’t in **themselves immutable (like emails)**?

5. How can I make the **unavoidable** resolution of **conflicts** by the human as **painless** and **intuitive** as possible?



@misprintedtype

This is an **error message**. My data **hasn't** been **stored**.

This **won't help** anyone… It's **great** at least I know that my **data** haven't been **saved**, but what **now**?
Doing it all over **again**?!

And **finally**:

6. Can the app make any **prefered decisions** on the part of the human and **pre-load** any **data** they might need **later**?
What **metrics** (possibly **behavioural**) are there to base these **decisions** on?

How can I **word** all of these **scenarios** in a way that **doesn't worry** the users, that **embraces** offline as a **reality without causing problems**?

This are the **issues** from the **user side** we have to **think** about…

What does that **mean** for the **server behavior**?

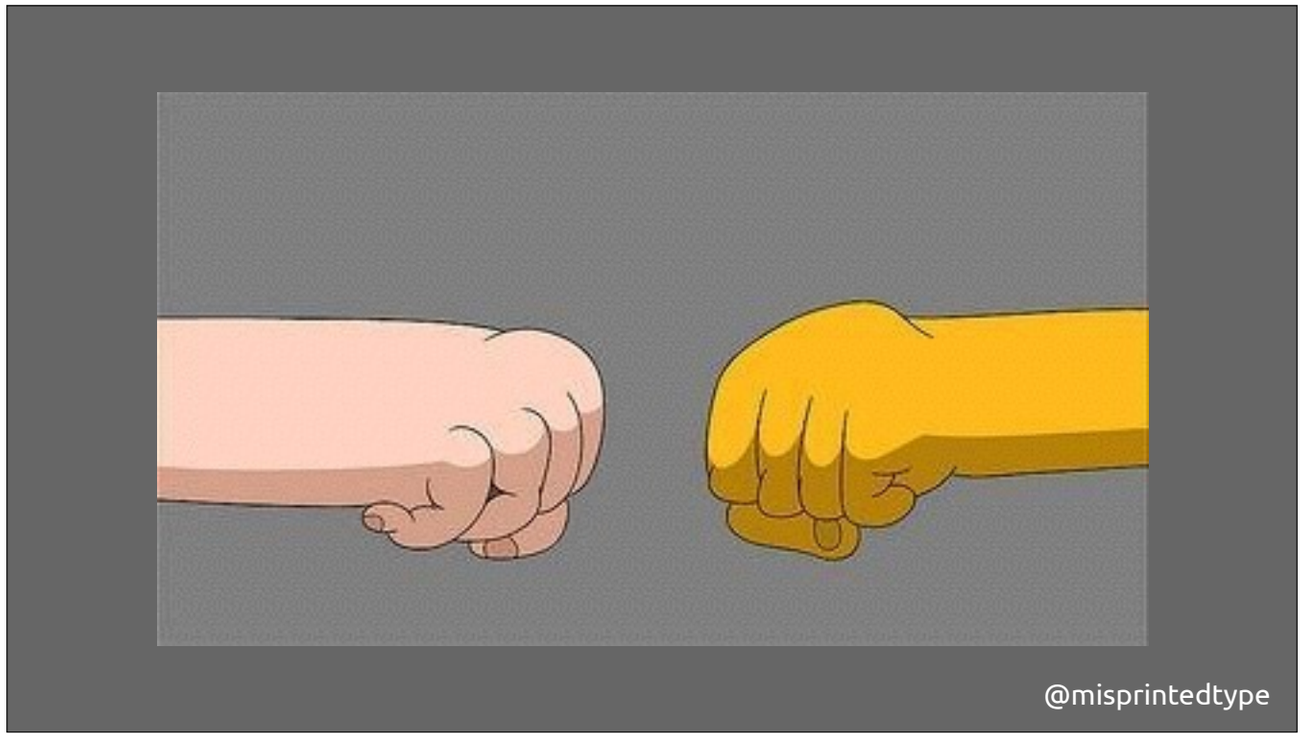
Meet the **staff**

Traditionally the server is the **source** of **truth**!

The **client** was always just the **prophet…** It **communicates** always with the **server**, sender **requests**, waited for **answers** and told him everything what **happened**. If the server is **not available…** the client **panicked** and was **helpless**, didn’t know what to do!

When we want to **post** some **data**, the client send out a **request** and waited for the **respond**, the data was **stored** and the **view** rendered and reloaded again.

With an async **request**, the client at least was **on call** and didn't need to **wait** all the time for the server… a step forward, but still… it was on call.



Let's **make** the client **independent**!

Times has **changed** and so has the **internet**. It **empowered us** and so it also empowered the **client**!

So, with offline first… we want to **decouple** this **relationship**.

But how shall we **deal** with the new **states**? What **ARE** the new **states**?

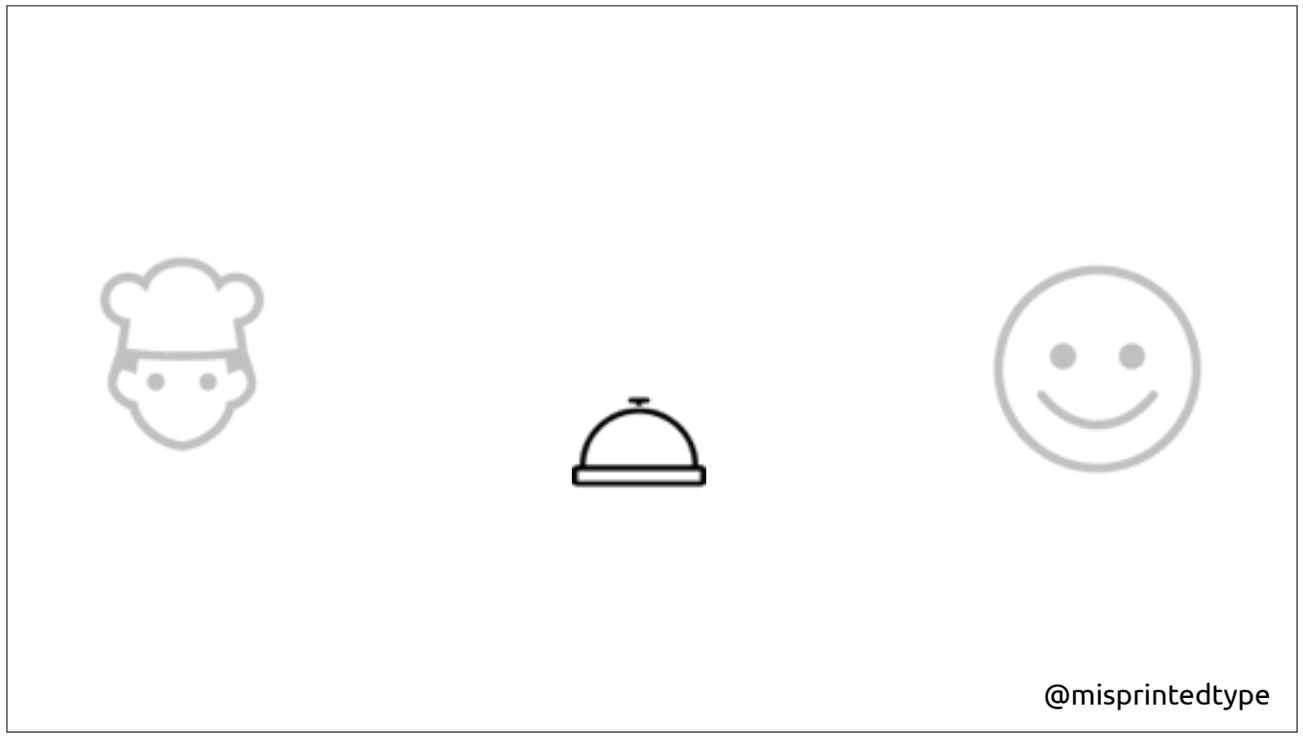
A **scencario** we all know:

Imagine a **restaurant**.

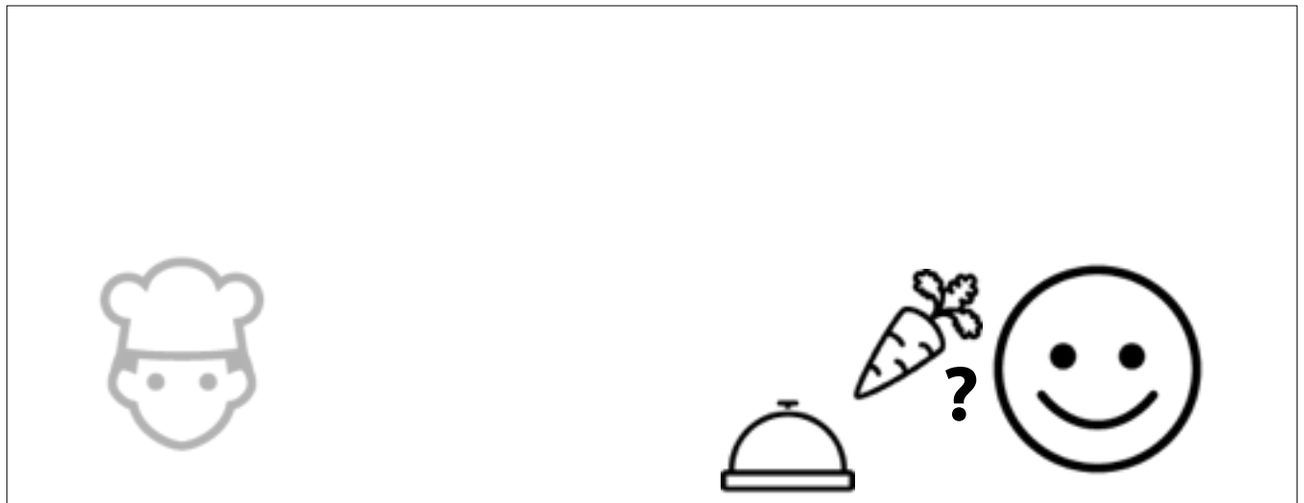
The **client** is the **guest**, how would **love** to **order** some **food** and enjoy their dinner.

@misprintedtype

The **kitchen** is the **server**. The kitchen **knows** all the **recipes**, **ingredients** and prepares all the **food** for the guest.



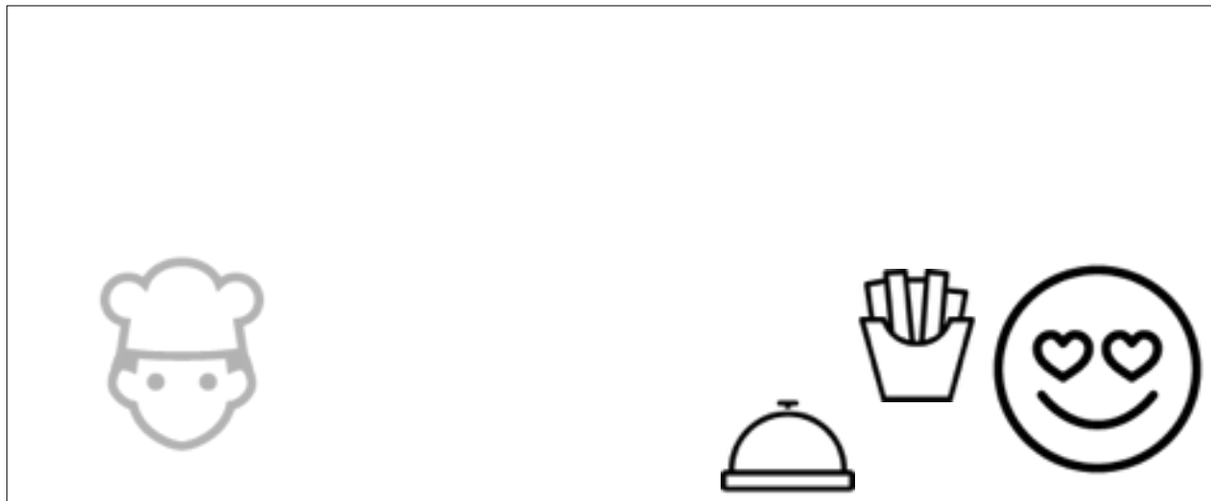
The **waiter** gets the **requests** and **serves** the **responses**, independent if it's a success or something failed.



So, as we know the concept, it works like this.

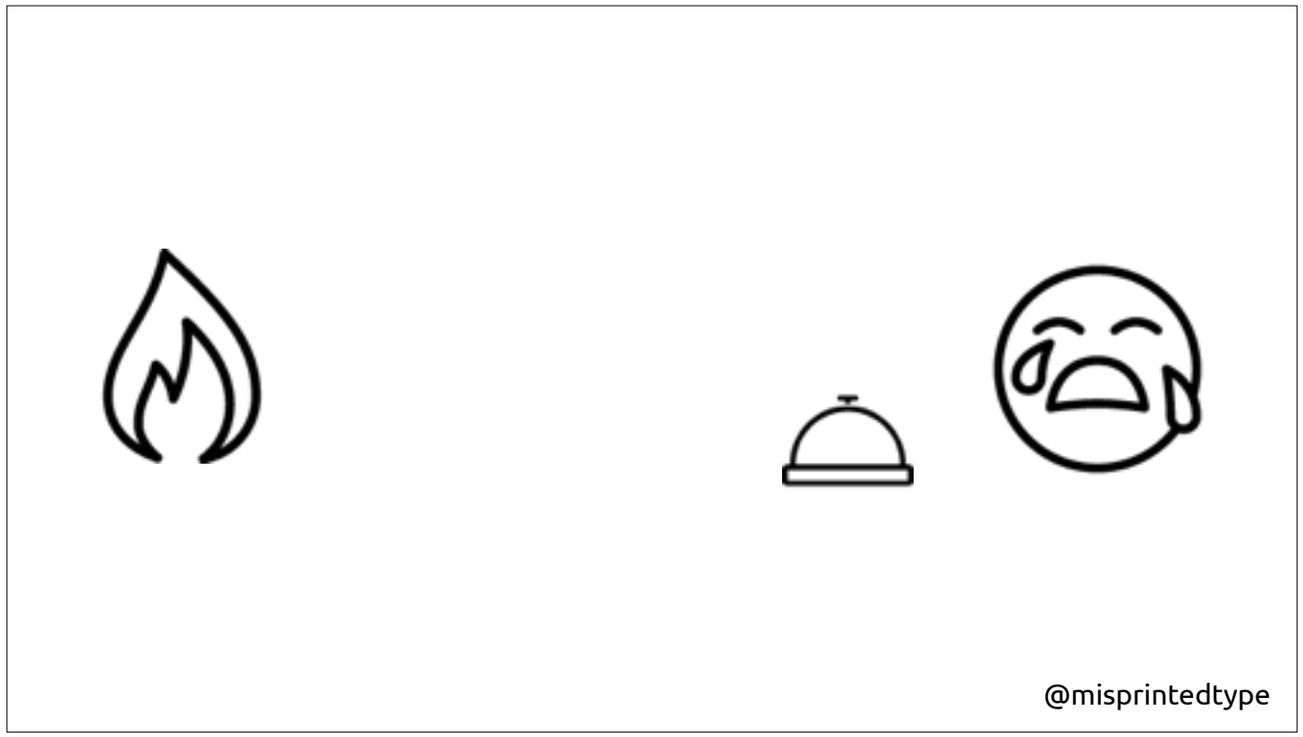
The **client** decides on a **dish** and orders.

The **waiter** serves the **request** to the kitchen. Kitchen **processes** the request.



And **serves** back the **result**. Here, it would be the **well-prepared dish**.

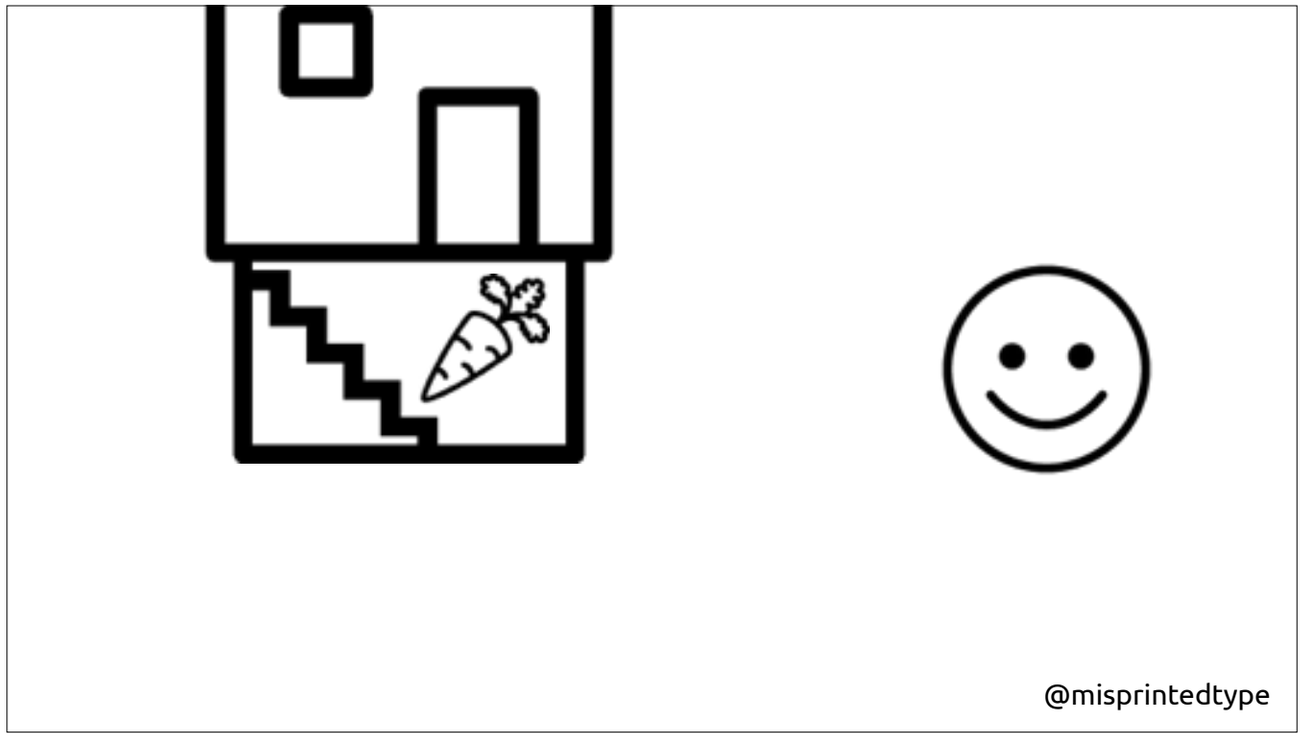
OR…



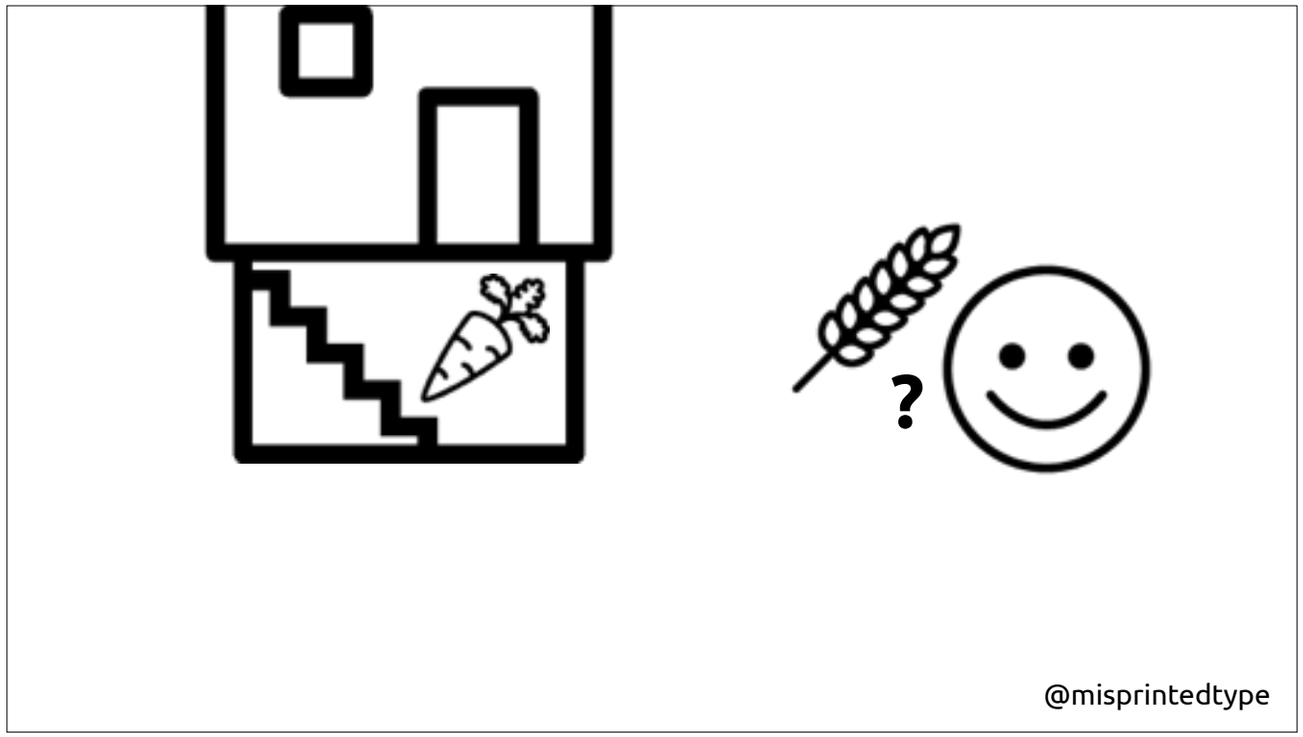
An **error** might **occur**… Like the kitchen **doesn’t** have the **recipe** or the kitchen is on **fire**… Well, so the waiter would get **back** to the **guest** and **report** the **error**.

Still… All sounds **familiar**, right?

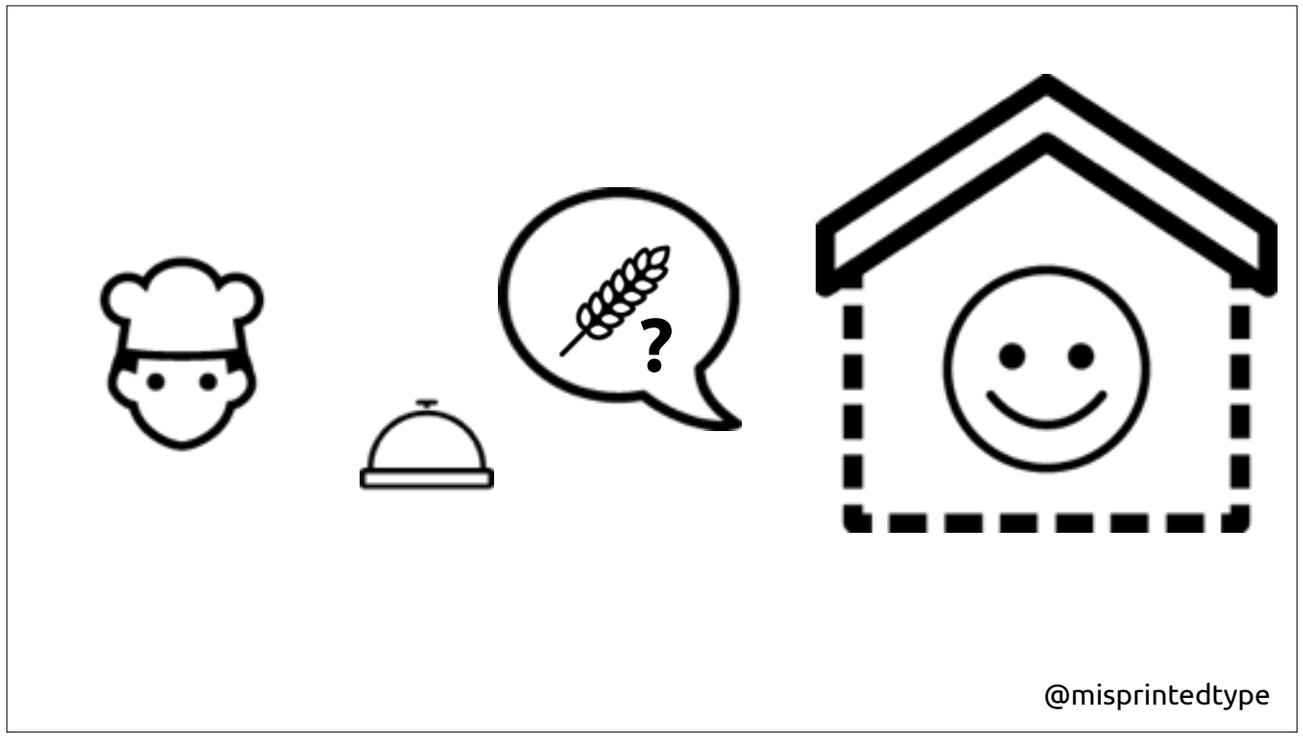
With **offline first**, it's like the guest would be at **home** and **not** at the **restaurant**.



So, all the **ingredients**, the **recipes** need to be **stored** at **home**. When the person, so the client **decides** on a **dish**, they can **cook** it by **themselves**.



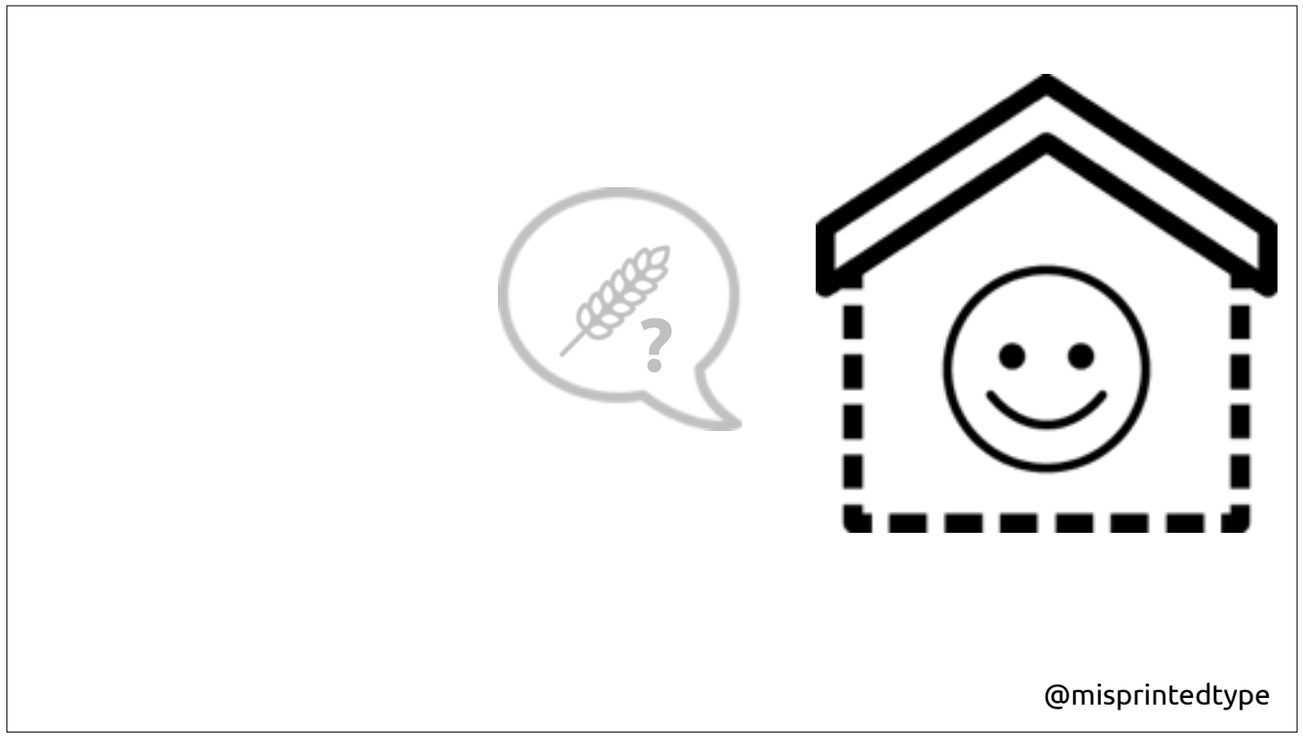
This works, if all the **ingredients** are **available** or the recipes are **known**. But what if the person **decided** to try something **new**?



As a **passionate cook**, the person needs to **ask** someone for the **recipe** first.

The **kitchen** loves sharing their **knowledge** (OPEN SOURCE, WOOHOOO!), so the **waiter** gets the **recipe** and gives it to the **person**, who can actually **start** to cook right away, they realize… hey! all the **ingredients** are **here**…!

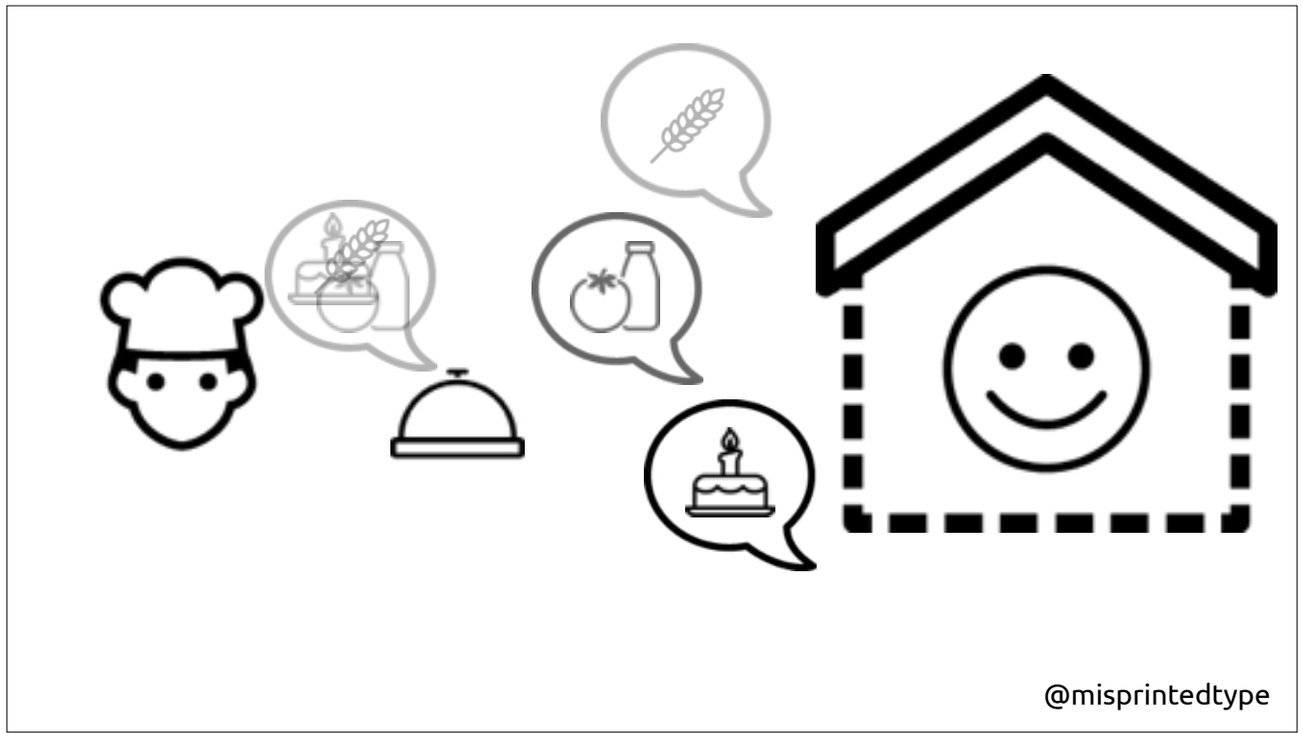
So offline first **means**, you can basically **cook** at home anytime you **want**. It just works. But for anything totally **new**, you need to **send** out a **request**.



Here are the „**new**“ scenarios:

The **waiter** is **not** available, so the **request** will **fail** right away.
But what **then**?

Storing the **draft**? Making a **queue** and trying to **send** it again, after 1, 2, 5, 10 **seconds**? **Blocking** the requests? Showing an **error** or just **ignoring** it and make it **disappear**, like it never **happened**?



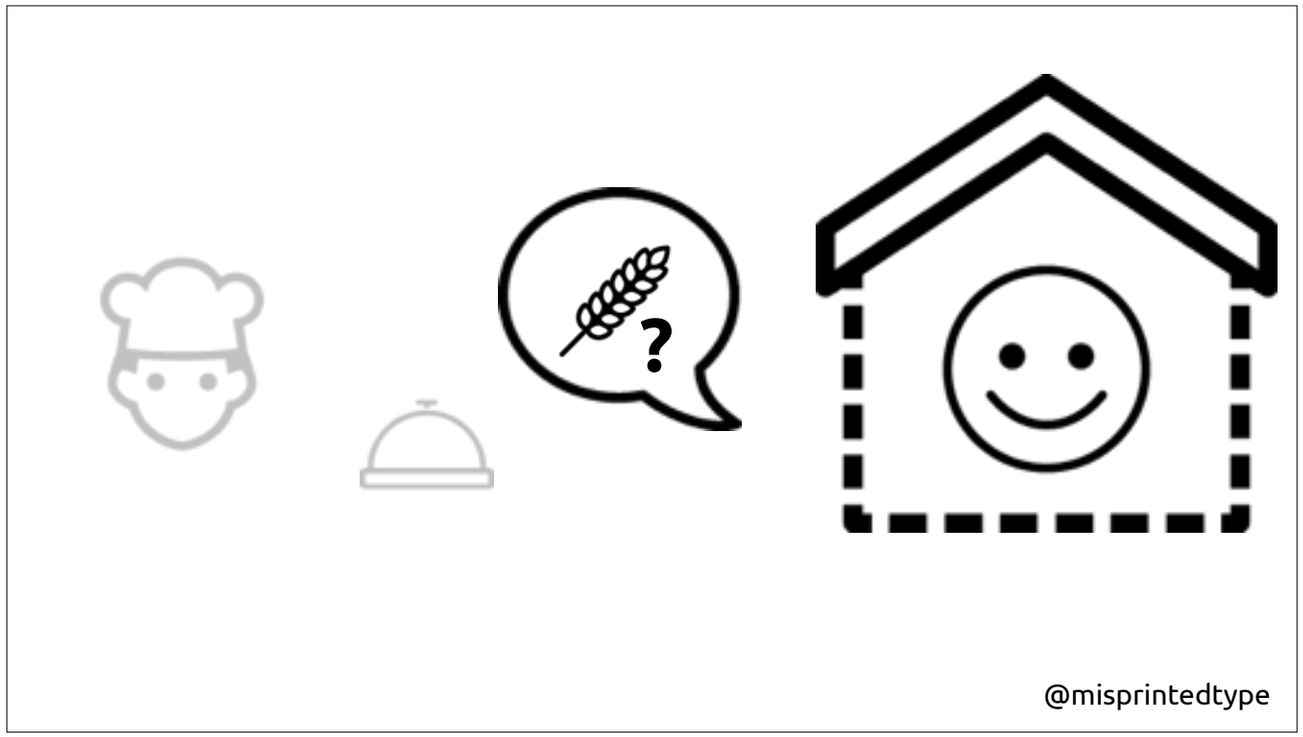
Another **scenario…**

A part of the **recipe** makes it **back** to the client, but **not all** of it! Because the **waiter** has so **much** to **do** and needs to **rush** back to the restaurant. So the person **couldn’t** understand **all** of it.

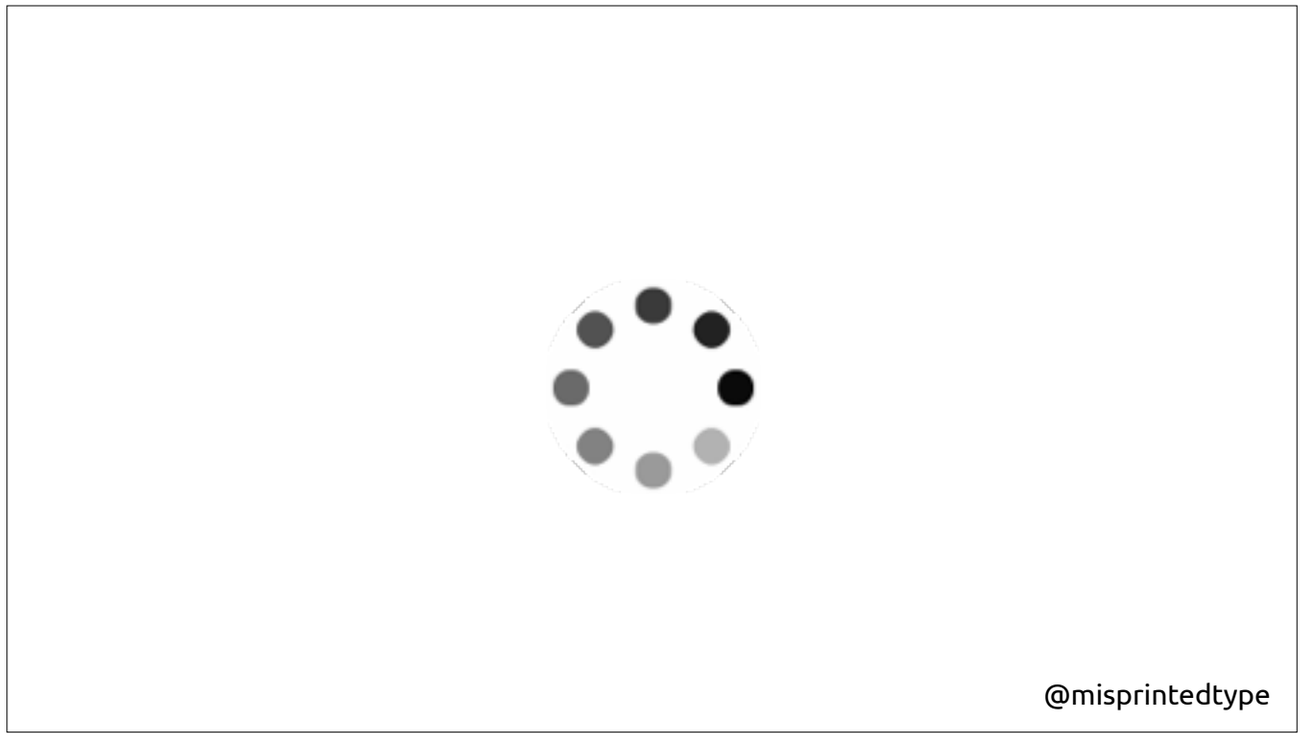
So, should the the person **cook** with just **half** of the recipe and see if this is **enough** to **produce** a **good** meal? **Don’t** cook at **all**? Cook as **far** as it’s possible and try again to **call the waiter** from time to time?

Can the person change the **request** for the **recipe** while he waits, because the craving for **wheat** got **replaced** by a **craving** for beans? How will the waiter **proceed** with the **new** request?

And how can we keep his **frustration level** as **low** as possible?

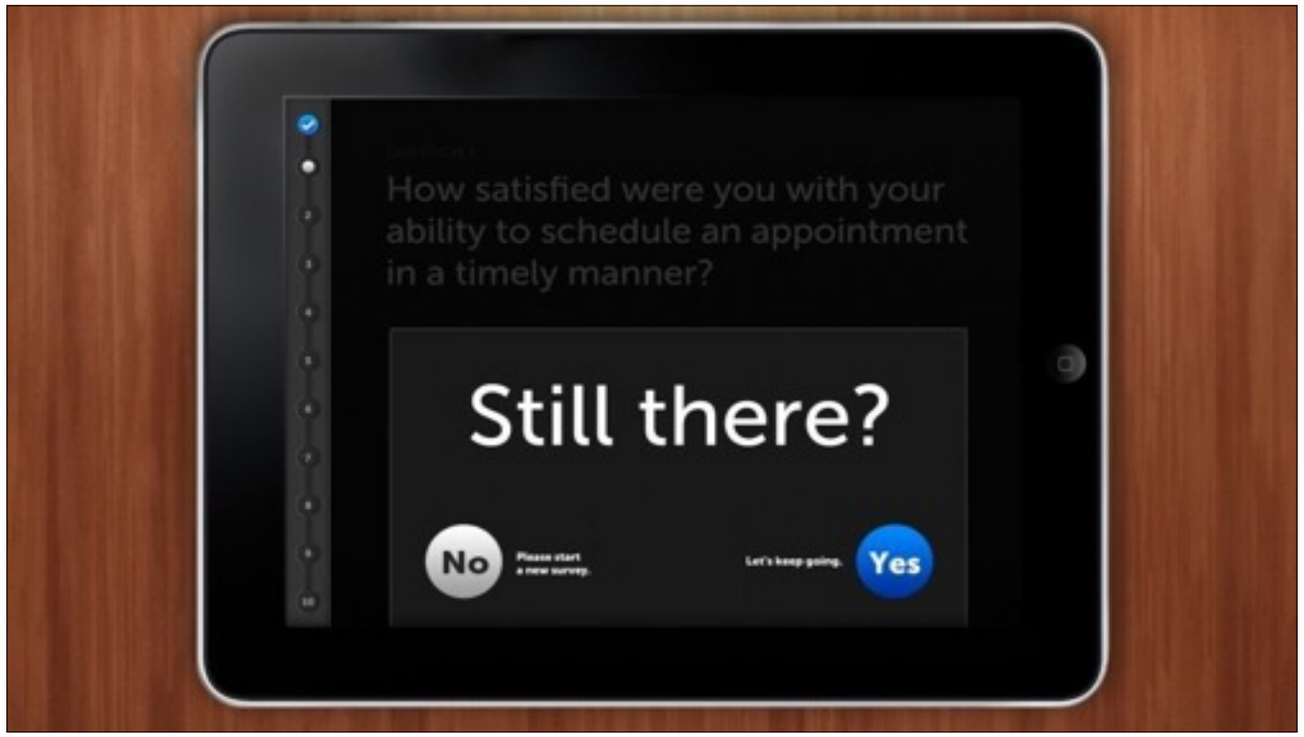


While the **guest** is making a **request…** Like **asking** for the **recipe**, the interface, here the interaction between the guest and the waiter should **start** by showing that a **process** of **unspecified** duration was started.



Usually done with **this…**

We all **love** that… right…? // 1, 2, 3… Annoying!

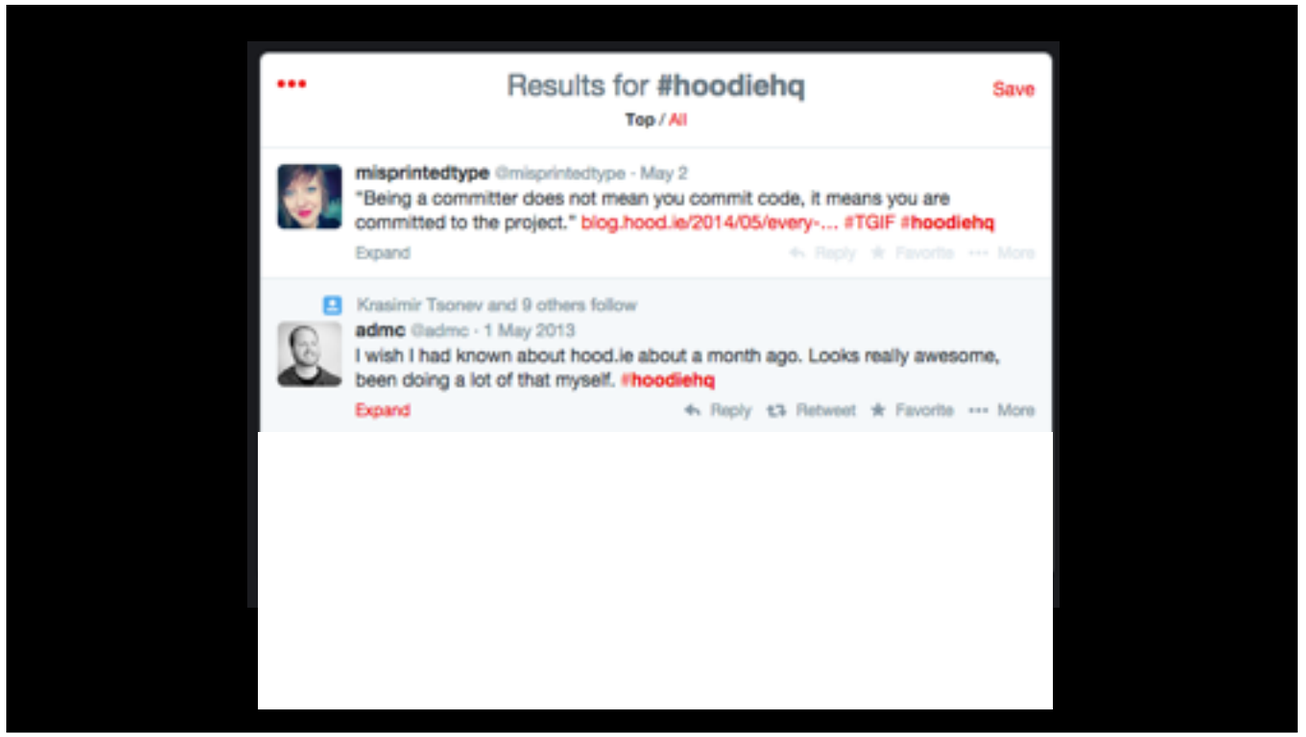


Based on the **different** error **scenarios**, the user interface needs to **account** for the fact that the **request failed** immediately, or after a **timeout**, or only a **partial** response was received.

Timeouts need to be **chosen** in a way that is **useful** for the end user, a common default is 1 **minute** and users generally **don't** wait that **long**. Or **would** you?

And always **remember…** when it's more that 1 **second**, keep the user **busy**.
Preload images, load **data** step by step if it is **available…** and so on!

In case of a **timeout**, there should be a user interface **affordance** for **cancelling** an operation, so that the **user** can apply a **shorter** timeout if that **makes sense** in the **situation**.



Partial responses are **tricky**, but can still be **useful**, e.g. starting to show the top **half** of a web page, or email, or other **message**.

The time it takes the user to **read** through that might be enough to **fetch** the **rest** of the data. Sometimes not enough data is provided to **do anything at all** and the **user interface** should **account** for that.

We **see**… the **offline first concept** became **really** important for us and we **have** to **rethink**, how we **concept** and **build** applications.

Rethink!

(but how?)

@misprintedtype

But **how** can we **implement** this?

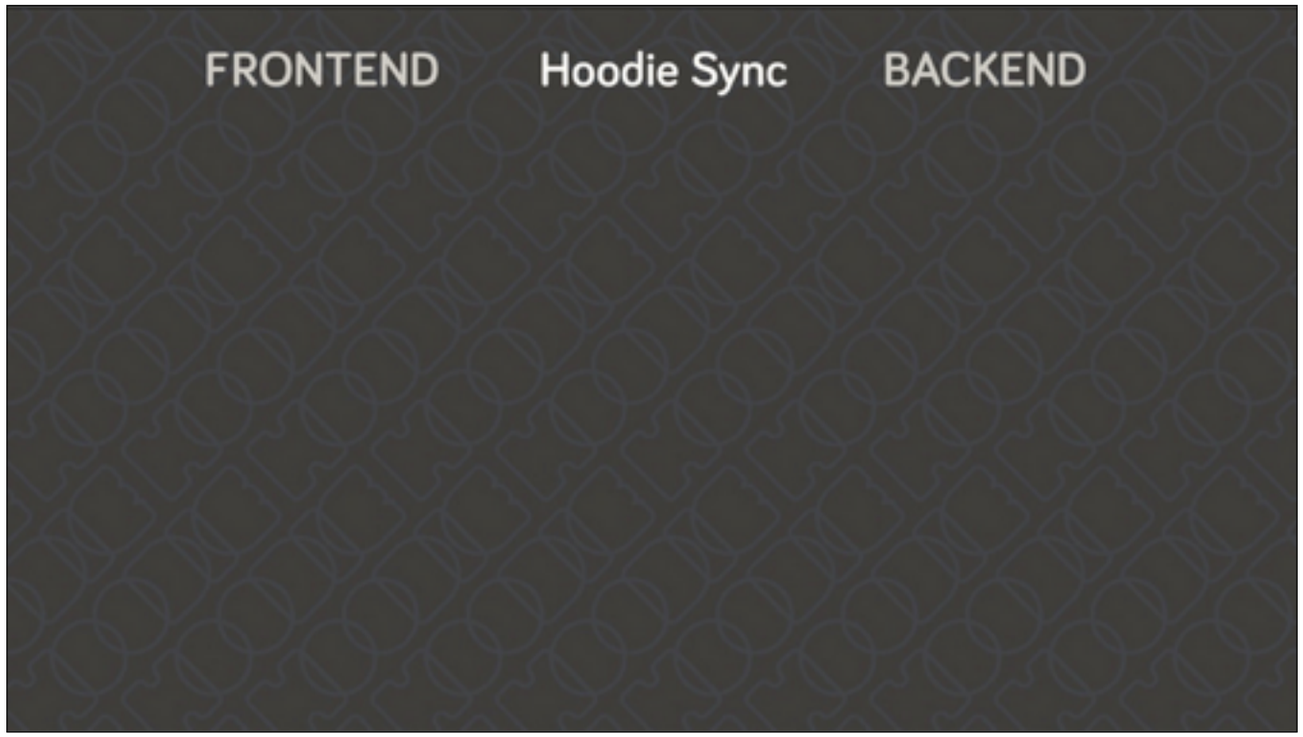
We need a **new** approach.
What about **decoupling** the **client** and the **server** side.

I'll **show** you a **way**, the **Hoodie team** came up with.

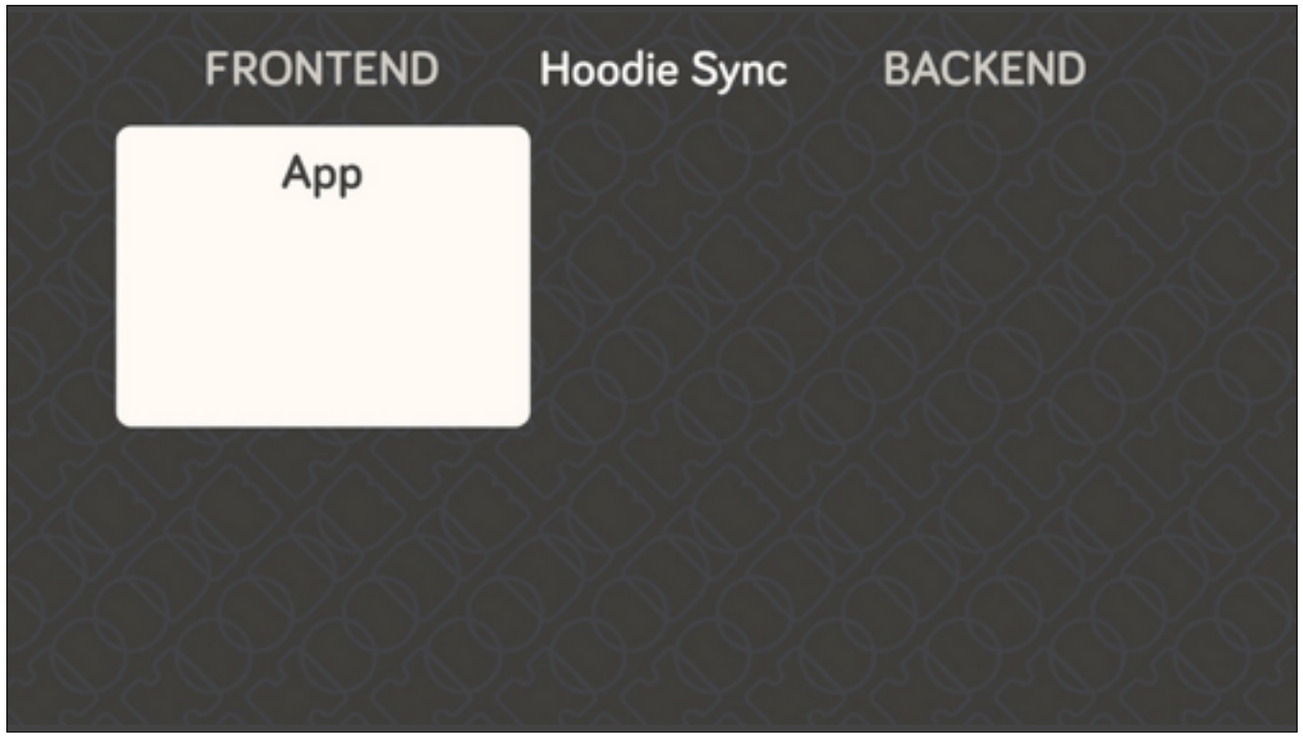
Hoodie is an **OpenSource** and free **JavaScript library**. It offers you a **whole backend**, with **user accounts, storage, shares, email**… it's **offline first** including a **localStorage** and automatic **sync**. It has a **dreamcode** API, every frontend dev **understands** in a second.

Hoodie **abstracts** away all the **boring boilerplate** backend stuff you **don't want** to **think** about.

Let's **take** a look at the **architecture**.

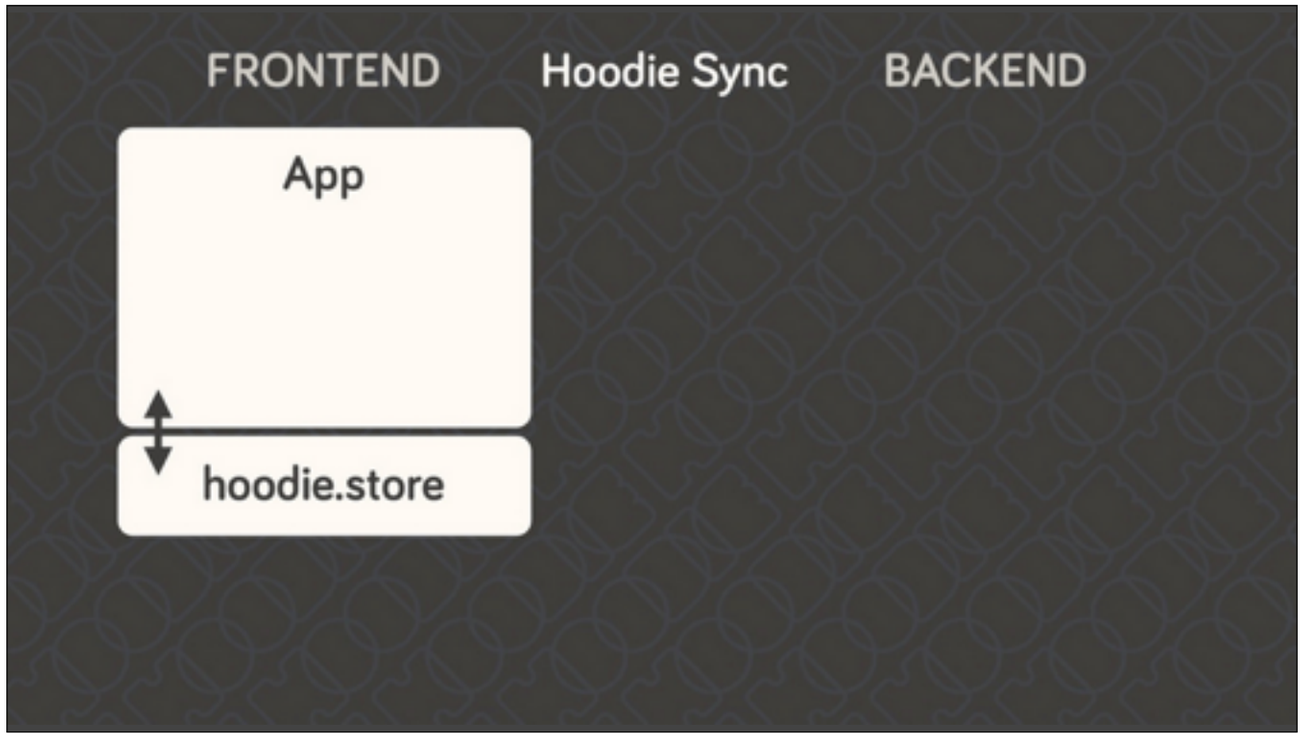


There for we have **3** Stages...



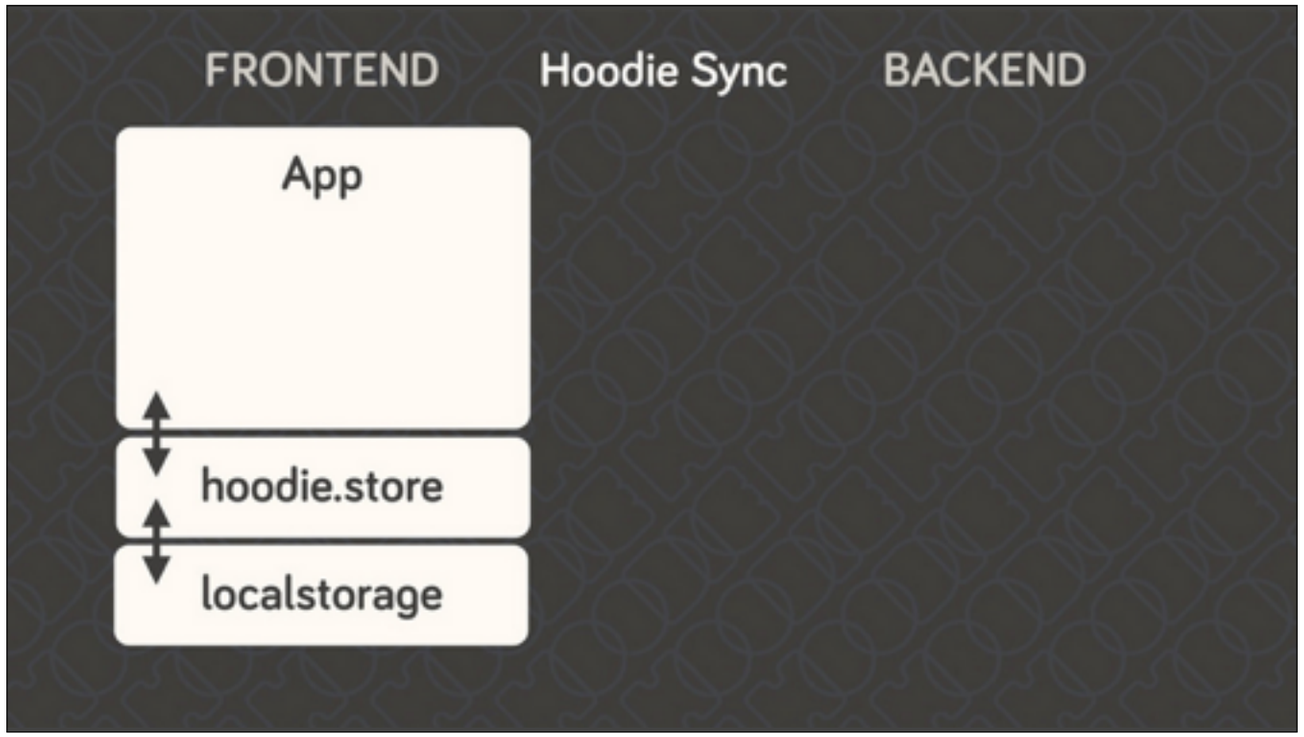
We have the **frontend**!

Here you have your app...



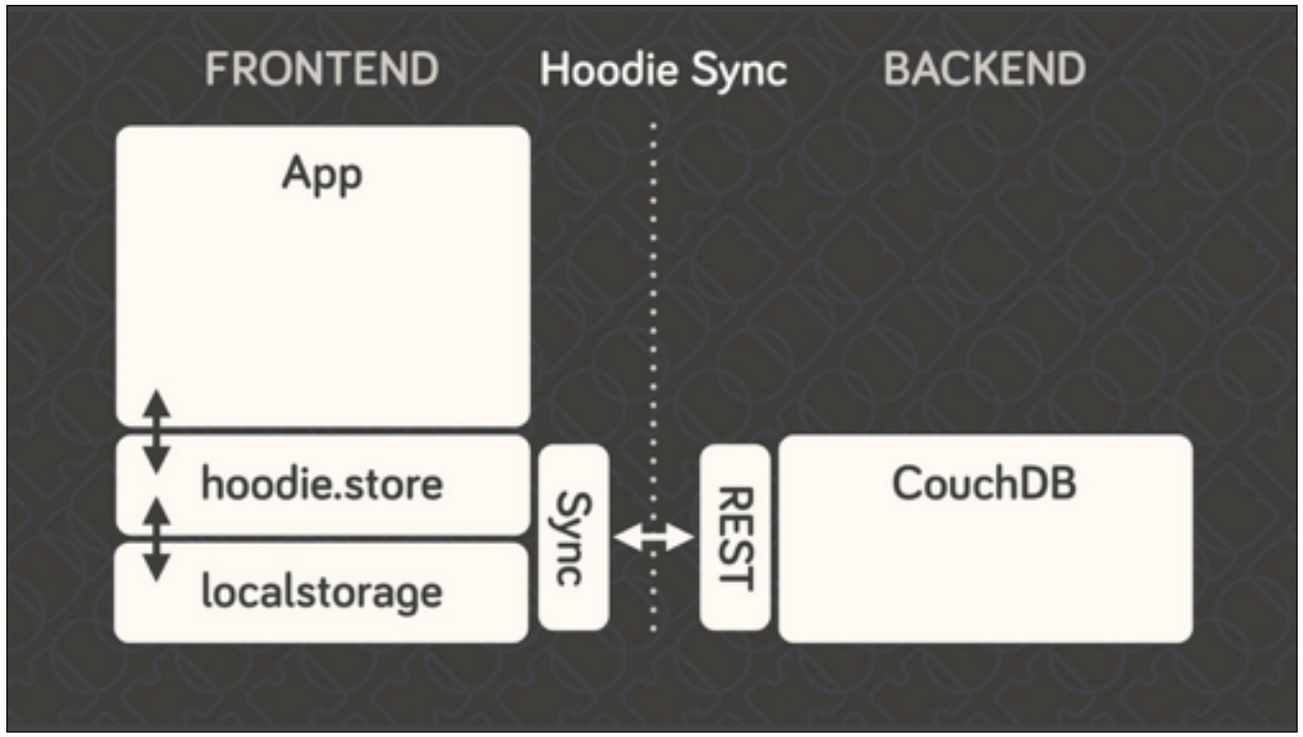
Then we have the Hoodie **layer**. Here we **use** hoodie.store.

The **app** only ever **talks** to the Hoodie API, **never directly** to the **server-side code**, **database** or **even** in-browser storage.



Feel free to **replace** the localstorage with any in-browser storage of **your choice**!

This, by itself, is **enough** for an app!

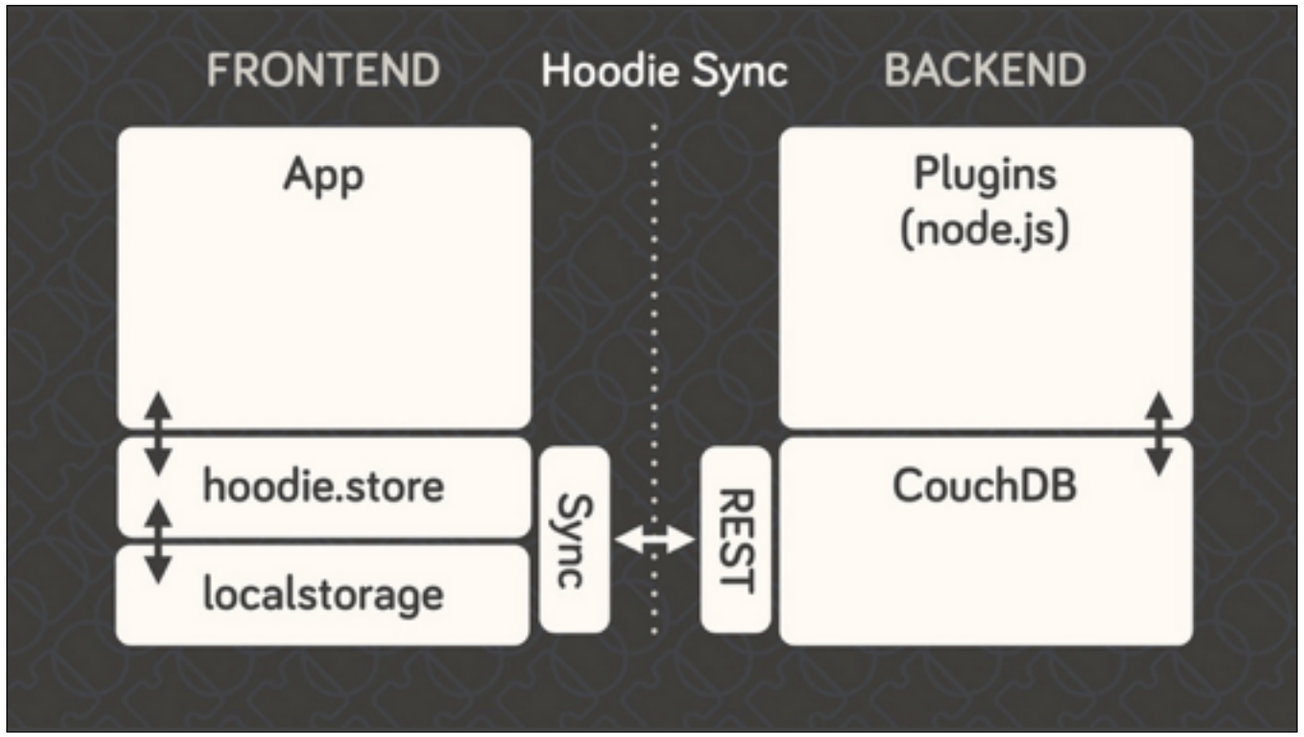


There we have a **sync** layer, a **REST** layer and the **database**.

We can do this because **each user** has their own **private** database which they **only** can **access**.
You can't **share** your data by **accident**! It's private by **default**.

So it's very **simple** to **decide** what gets **synced** and how to **handle changed** data...

You, of course, can **share** or **publish** it, if you want!



Additionally, there are **plugins**, which help us to **extend** Hoodie.

-> Explain a **direct message moving** through the system.

1. You **create** a **message** in your app and **send** it to the **hoodie layer**.
2. The hoodie **layer** sends out a **task** to the **localStorage**, so the **message** can be stored **locally** and **resolves** with a **promise**.
   At the **same** time, it sends a task to the **sync** layer.
3. When the device is online, **sync** layer **pushes** the message to the **CouchDB.** It get's **stored** and tasks are **send** out to **other** devices to **update** their client, because we **changed** the data.
4. And the plugin gets the **task** to send the message.

The main point here is, the client and the server, they never have to **talk directly** to each other. They only **leave** each other **messages** and **tasks**,

it's **all** very **loosely coupled** and **event-based**. Which means it can be **interrupted** at any **stage** without **breaking**.

Messages and tasks will be **delivered** and **acted upon** whenever **possible**.

Let's **talk** about the **implementation**.

We have to take care of **2 things,** which we **store** on the **devices**. **Files** and **data**.

Let's first **take** a **look** at the **files**, we **need** to **cache.**

Without files, **no** application and **no** data**,** right?

-> Who of you **heard** about the **app manifest**.
-> Who is **already using** it in their **applications**?

The **manifest** gives you the **possibility** to **fetch** the **files** you'd **love** to **load** from the **cache**, when you are **offline**.

I **won't** go any **further** here, because it's very **simple** to **implement** and there are a lot **tutorials** you can check out. Just **one** thing you **need** to **know** here…

App Cache

is a douchebag!

http://alistapart.com/article/application-cache-is-a-douchebag

App Cache is **great** to store your **files**.
But, it **turned** out… App Cache is a **douchebag**!

1. When you **visit** a **site**, the **files** always come **straight** from the **cache**! **After** the **rendering**, the browser will **look** for **updates** to the **manifest** and **cached** files. You **need** to **reload** the page to **get** the **new files**…

The **Appcache** only **updates**, if the **manifest changed**.

HTTP already **has** a **caching model**.

As a slightly **unusual workaround**, the **browser** will **only** look for **updates** to files **listed** in the **manifest** if the **manifest** file **itself** has **changed** since the browser last checked. **Any** change that **makes** the manifest **byte-for-byte** different will **do**.

App Manifest

3. App Cache is an additional cache

@misprintedtype

THE **APPLICATIONCACHE** IS AN **ADDITIONAL** CACHE, **NOT** AT **ALTERNATIVE** ONE

When the **browser updates** the **ApplicationCache**, it **requests** urls as it **usually** would. It **obeys** regular **caching instructions**:

If an **item's header** says "I'm good until 2022" the **browser** will **assume** that **resource** is indeed good until then, and **won't trouble** the **server** for an **update**.

Without specifics, the **browser** will take a **guess** at the caching.

All **files** you serve **should** have **cache headers** and this is **especially** important for **everything** in your **manifest** and the **manifest itself**. If a file is **very likely to update**, it should be served with **no-cache**.

App Manifest

4. Non-cached resources will not load

@misprintedtype

Last but not least:
NON-CACHED RESOURCES WILL **NOT** LOAD ON A **CACHED** PAGE

If you **cache** index.html but **not** the **background image**, that image will **not** display on index.html, **even** if you're **online**.

And **these** are just the **simple** problems…

App Cache nanny

https://www.npmjs.org/package/appcache-nanny

@misprintedtype

But there is a **solution** for that!

Gregor Martynus tought the App Cache some **manners** and **developed** the app cache **nanny**.
It **helps** us to **deal** with most of the **problems** the app cache **still** has, using an **iframe** for the manifest. So the **files** are **served** from the **server**, when you are **online**, it **checks** if the **manifest** is up-to-date.

And there is also **another** silver lining!

Work it!

@misprintedtype

So, you **might** have **heard** about the **service worker** already which seems to be the new **holy grail** of offline first…
The Service Worker is a **very specific** new **web worker**.

Web what?!

@misprintedtype

This sounds **great**, but **what** is a web worker actually?
So, let's take a **look** at **this** first.

The web worker was introduced with the **HTML5 specs** and is a **part** of the **standard**.

It **runs pre-defined** JavaScript scripts in your **browser**.

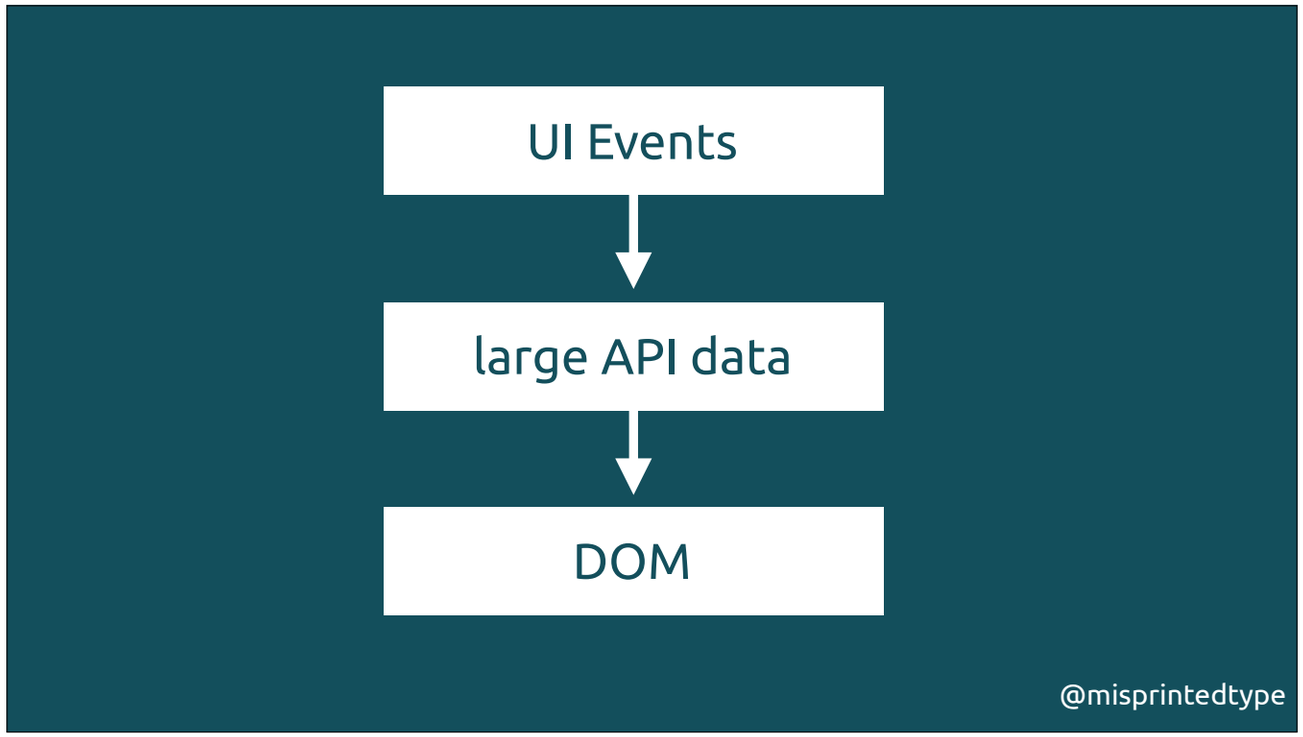
The **support** is pretty **good** like you can see **here**.
It's **supported** by all **major** browsers… **except** of **Opera Mini**, but well…

// waaaait! look at screen!

THIS is what **happens** if you are **running** too many **scripts**! Your **page** gets super **slow** and sometimes the **script** even gets **stopped** after a **timeout**.

This is like the **worse** case!

But sometimes we **need** to do things like that… for example… **Loading** a **huge** JSON file, running a **killer** algorithm on **some numbers** or **preload** sets of **data**.

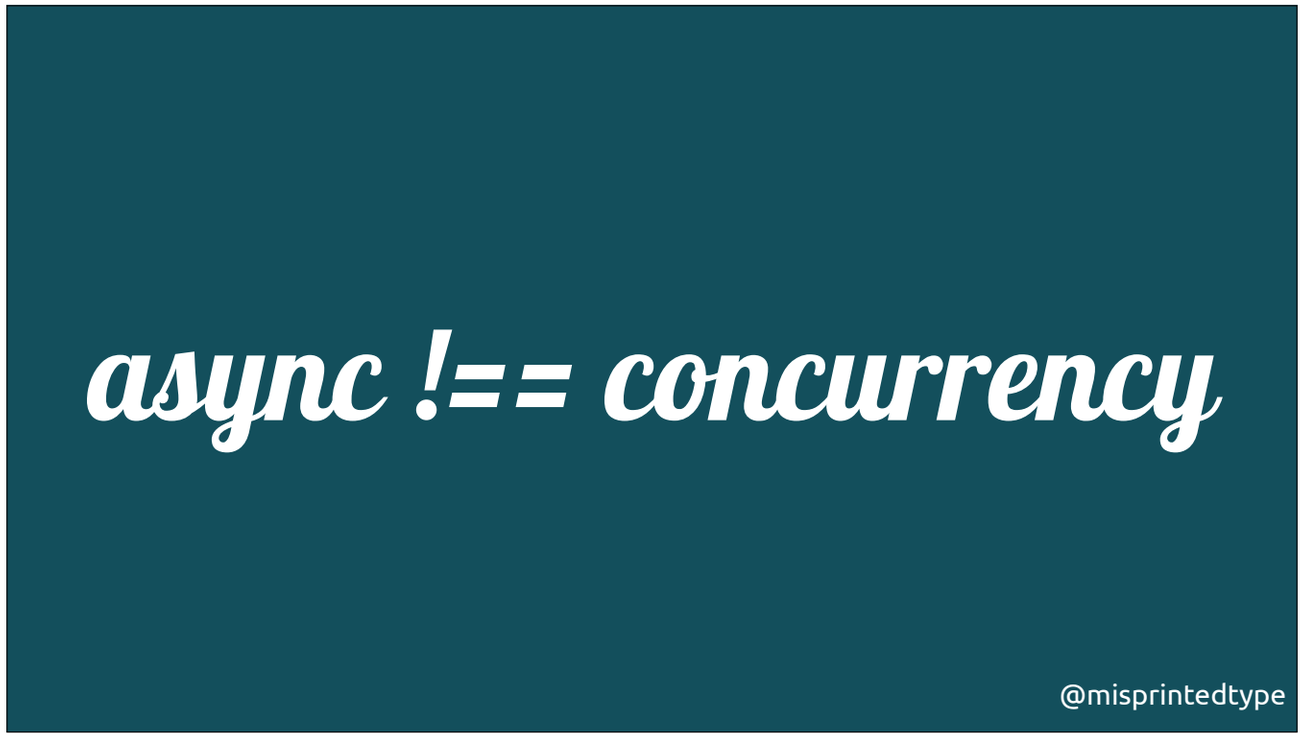


Imagine a site that needs to handle **UI events**, query and process **large** amounts of API **data**, and **manipulate** the **DOM**.

Pretty **common**, right?

Unfortunately all of that **can't** be done at the **same** time due to **limitations** in browsers' JavaScript **runtime**. Script execution happens within a **single thread**.

So this means, a **huge** front end logic can **block** your whole application.



Developers love to mimic ‚**concurrency**'.

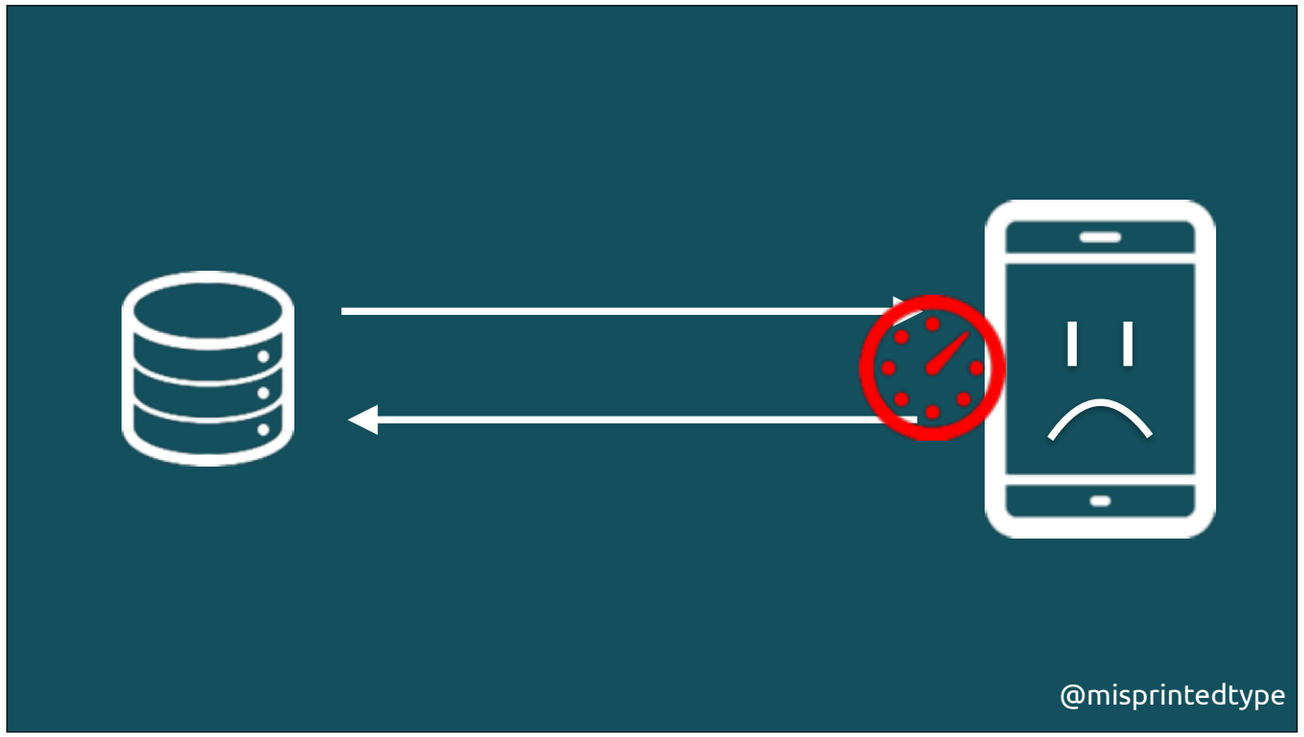
Async events are processed **after** the **current** executing **script** is done.

Async, but non-blocking **doesn't** necessarily mean **concurrency**.

Workers utilize **thread-like message** passing to **achieve** parallelism.
They're perfect for **keeping** your UI **refresh**, **performant** and **responsive** for users.

So here is your **application**.

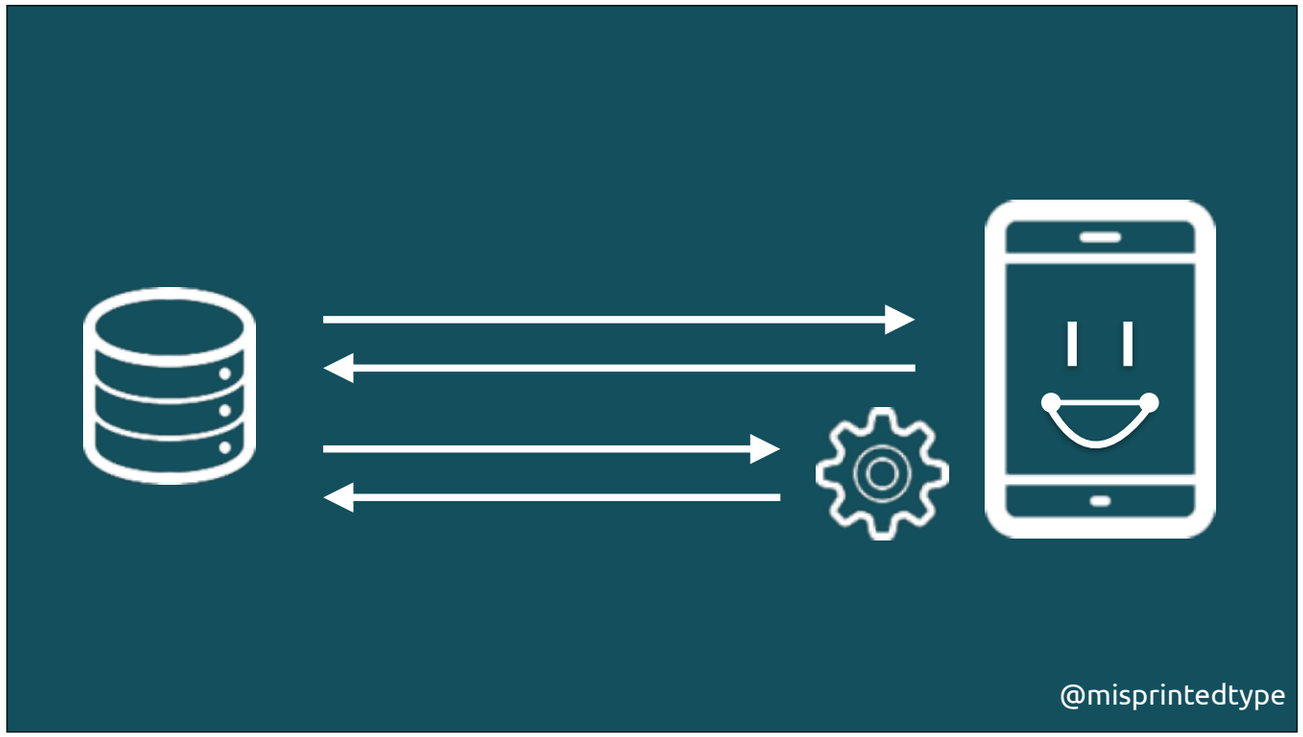
A very **happy** one, like you can see.



The moment you need to add **huge** tasks, like **pulling** a huge **pile** of data, your scripts **might** run into a **timeout**.

What is a **normal timeout** in the browser? What do you **think**? Hands up and please take a **guess**…

Consider that **80ms** is the point, where you should **start** to use workers. **50ms** should be a common **default**, **30ms** on **mobile** devices.



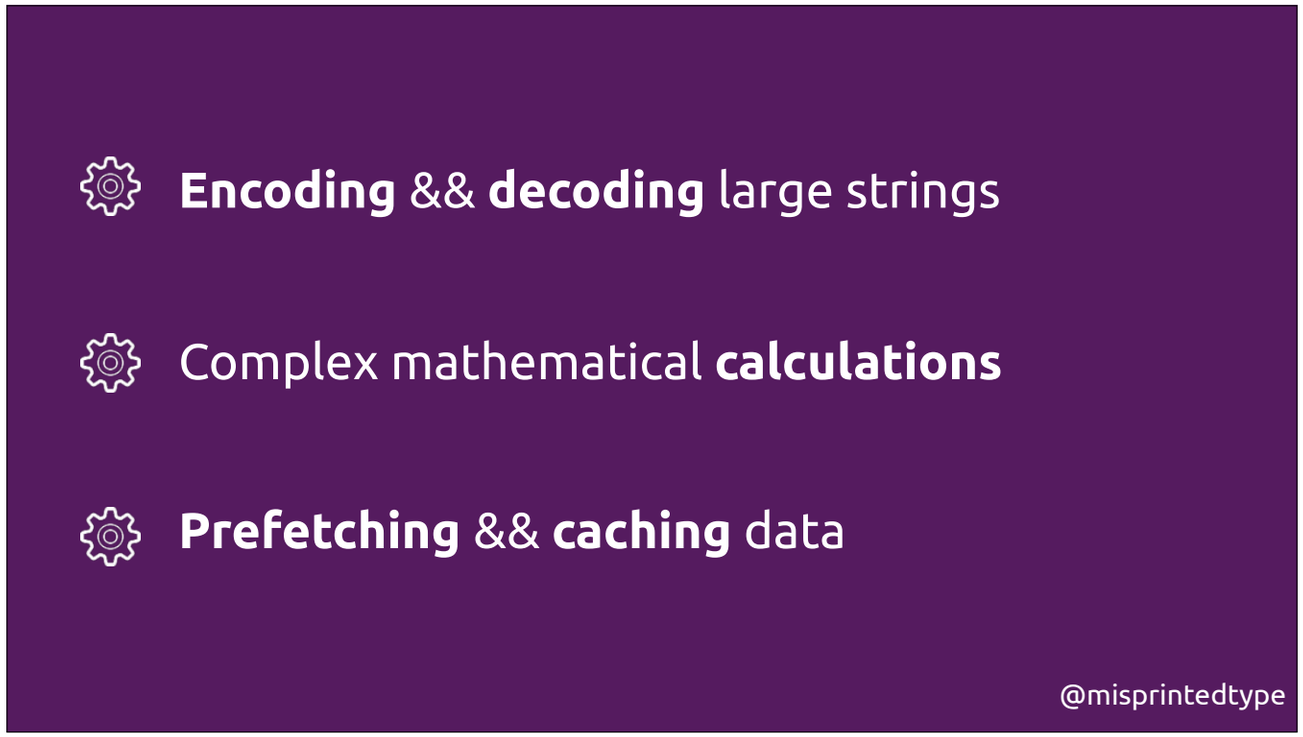
The Web Worker specs **defines** an API for **spawning** background **scripts** in your web **application**. It **allows** you to do things like **fire** up **long-running** scripts to handle **intensive** tasks, but **without** blocking the UI or **other scripts** to handle user interactions.

The web worker **threads** run their code from top to bottom, and then **enter** an **async phase** in which they **respond** to events and timers.

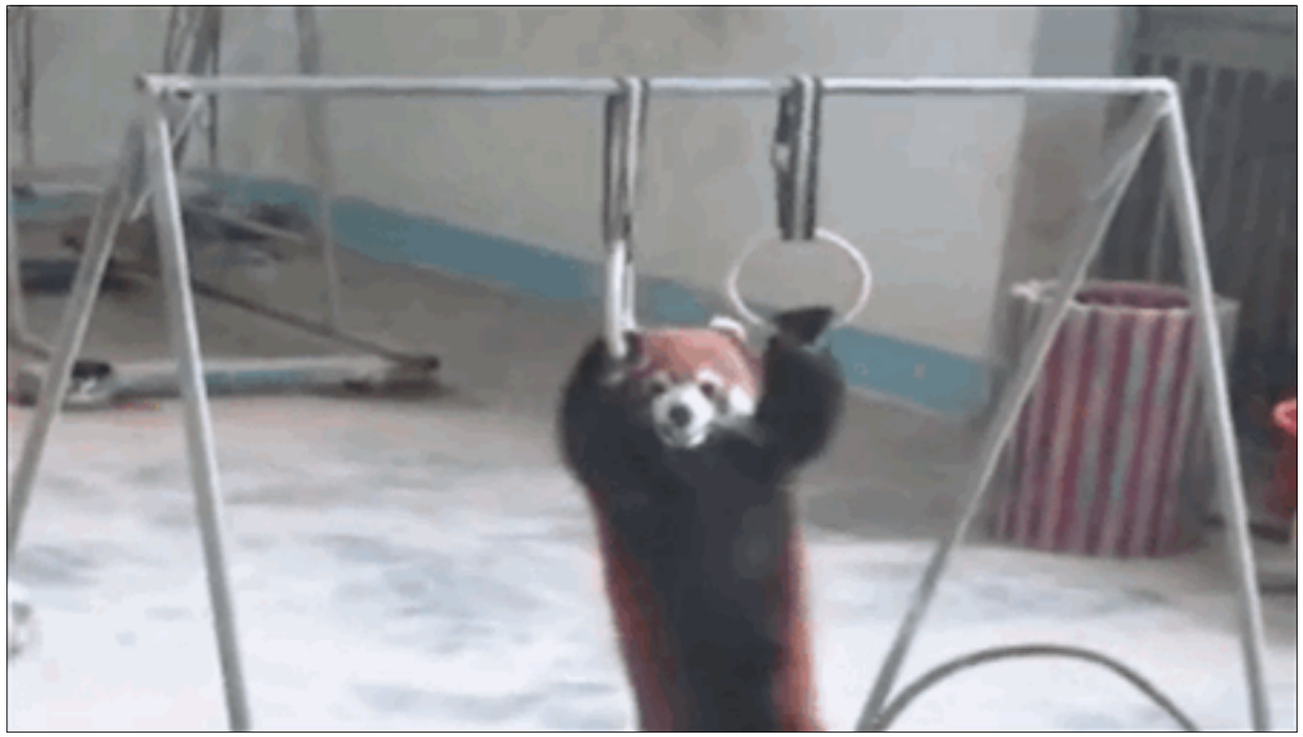
So you can still **interact** with your application, like you'd normally would. But **outsource** the **heavy** operations and if they **fail** or get a **timeout**, your applications is still **up** and **running**.

Sounds **awesome** right?!

So, **what** should we **use** workers for?



- Encoding/decoding a **large string**
- Complex mathematical **calculations** (e.g., prime numbers, encryption, simulated annealing, etc.)
- **Prefetching** and/or caching **data**

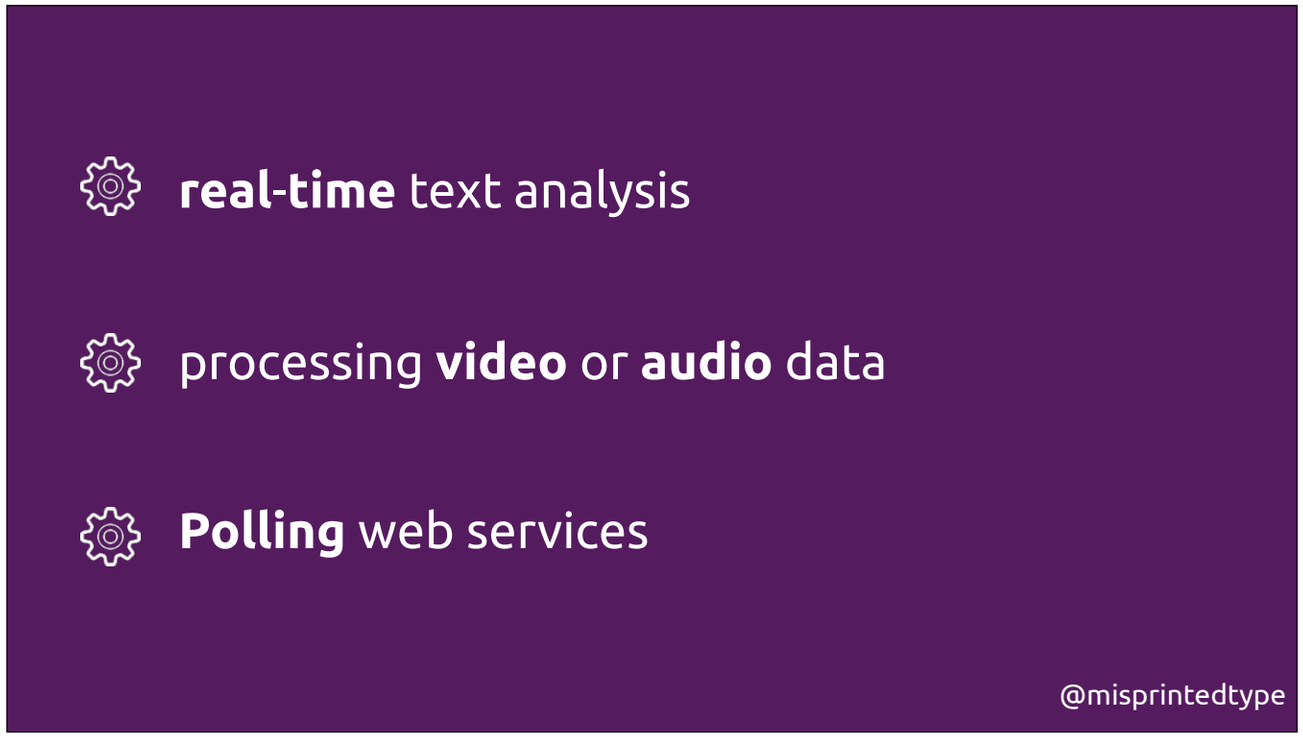


- Cool **right**?

There is **more**!

- Network **requests** and resulting data **processing** (especially Processing large **arrays** or huge **JSON responses**)
- **Calculations** and data manipulation on **local** and **session** storage
- **Image** manipulation

- Can it get any **better**?! Yes!



- Code syntax **highlighting** or other real-time **text** analysis (e.g., spell checking or real time search)
- Analyzing or processing **video** or **audio** data (including face and voice recognition)
- Background **I/O** and **Polling** web services

I'm sure you already **found** more then **2** use **cases** here, right?
Who did? Raise your hands!

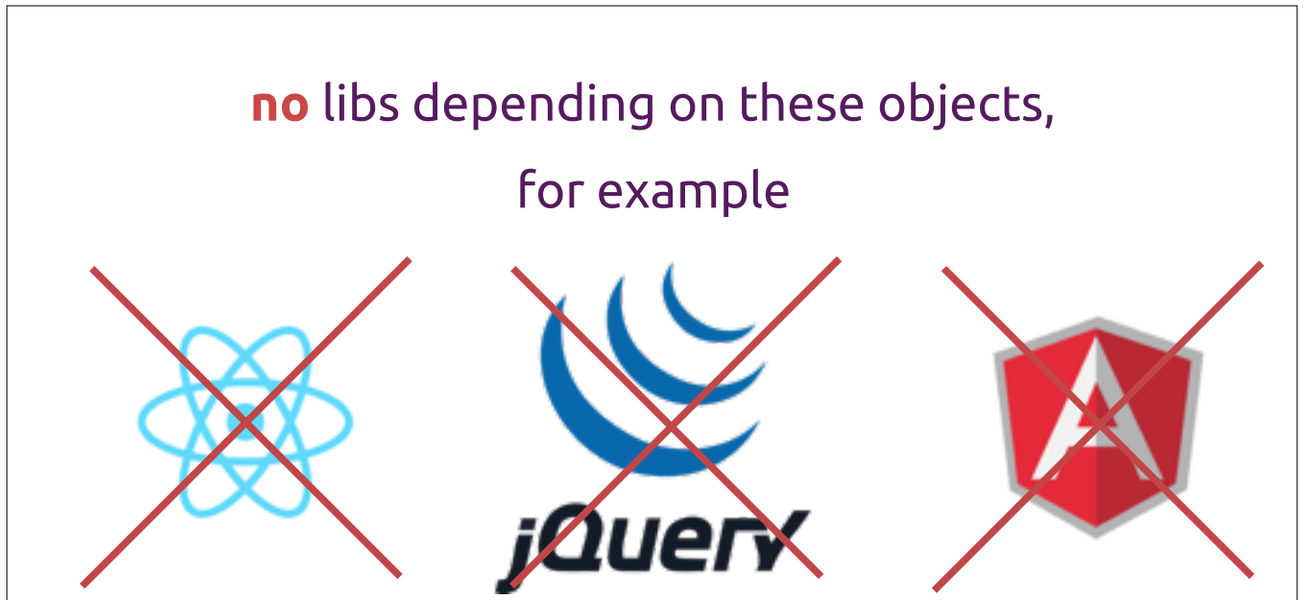
So, you fell in **love** with the worker already **right**?!

Even the **sky** has their **limits**… so let's take a look at what we **can** and what we **can't** do!

```
// worker.js - no access

// window
window.alert(`Work it!`);

// document
document.getElementById(`danceParty`);

// parent
window.globalFunction();
```

@misprintedtype

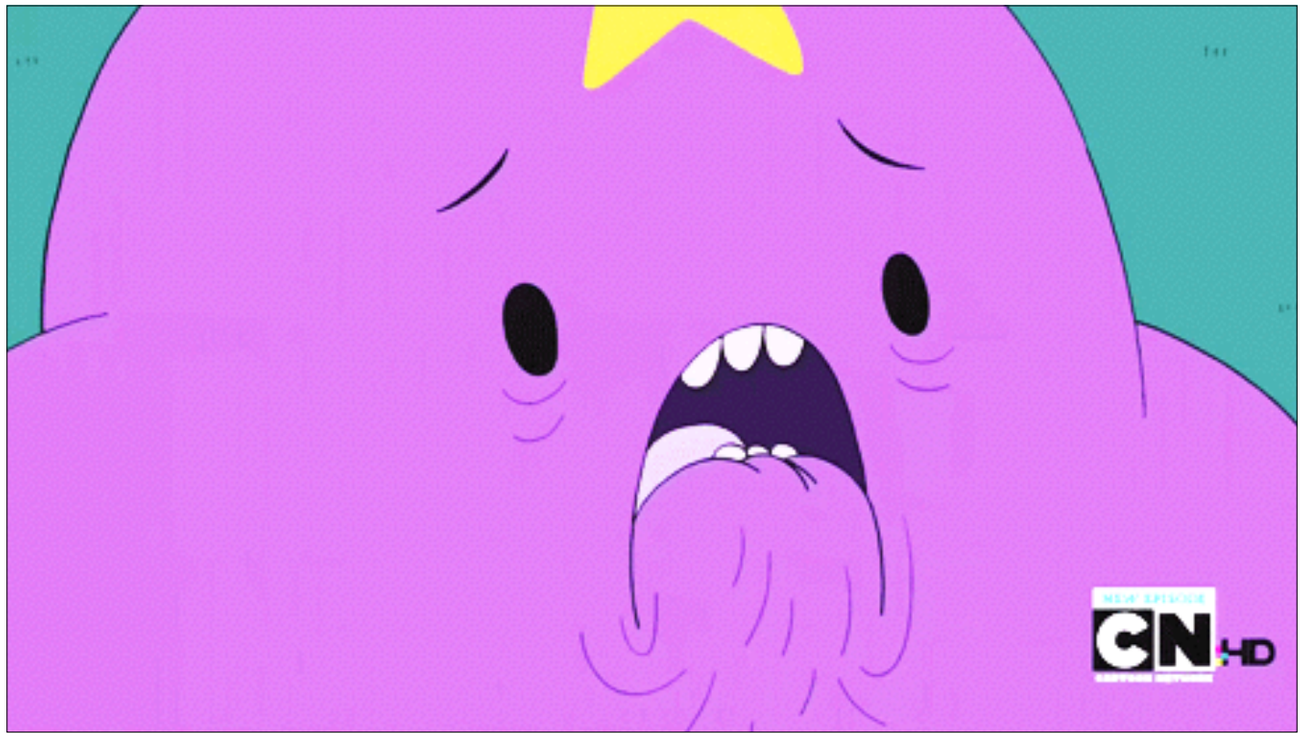
They **also** can't **access** any of the **following**:
- The **window** object
- The **document** object
- The **parent** object, so no access to page scripts, globals or functions



- And they can’t use **libraries** that **depend** on these **objects** to work. Nope, no **jQuery** or equivalent, sorry!

But you can **still** use those in **your UI**, but just **not** in your **worker** script.

The **upcoming** ReactJS version **won’t** rely on the bowser, but **still**… **ask** yourself, if you **really** need to **use** it.



BUMMER!

But what **can** we **actually** do?!

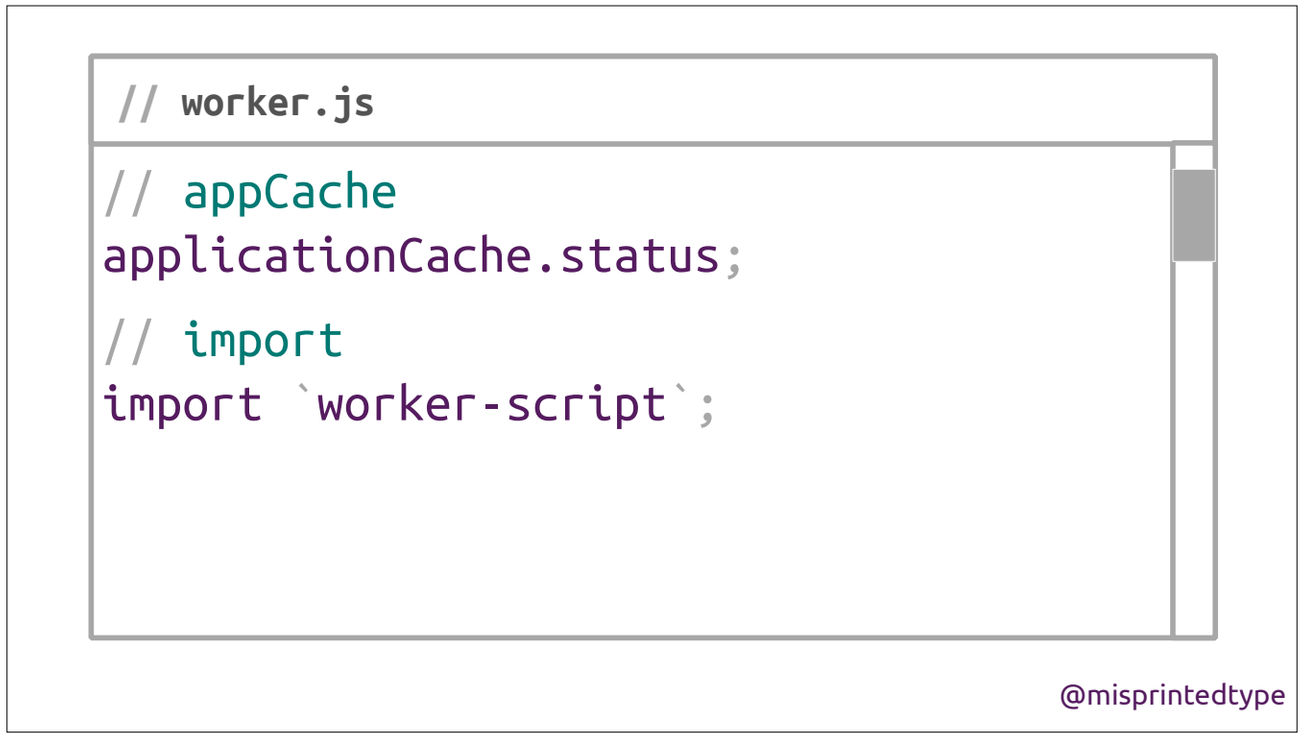
Because web **workers** have a **multithreaded** behavior, they can only **access** a **subset** of JavaScript's **features.**

```
// worker.js

// navigator
navigator.onLine;

// location
location.href;

// xhr
xmlhttp.send();
```

@misprintedtype

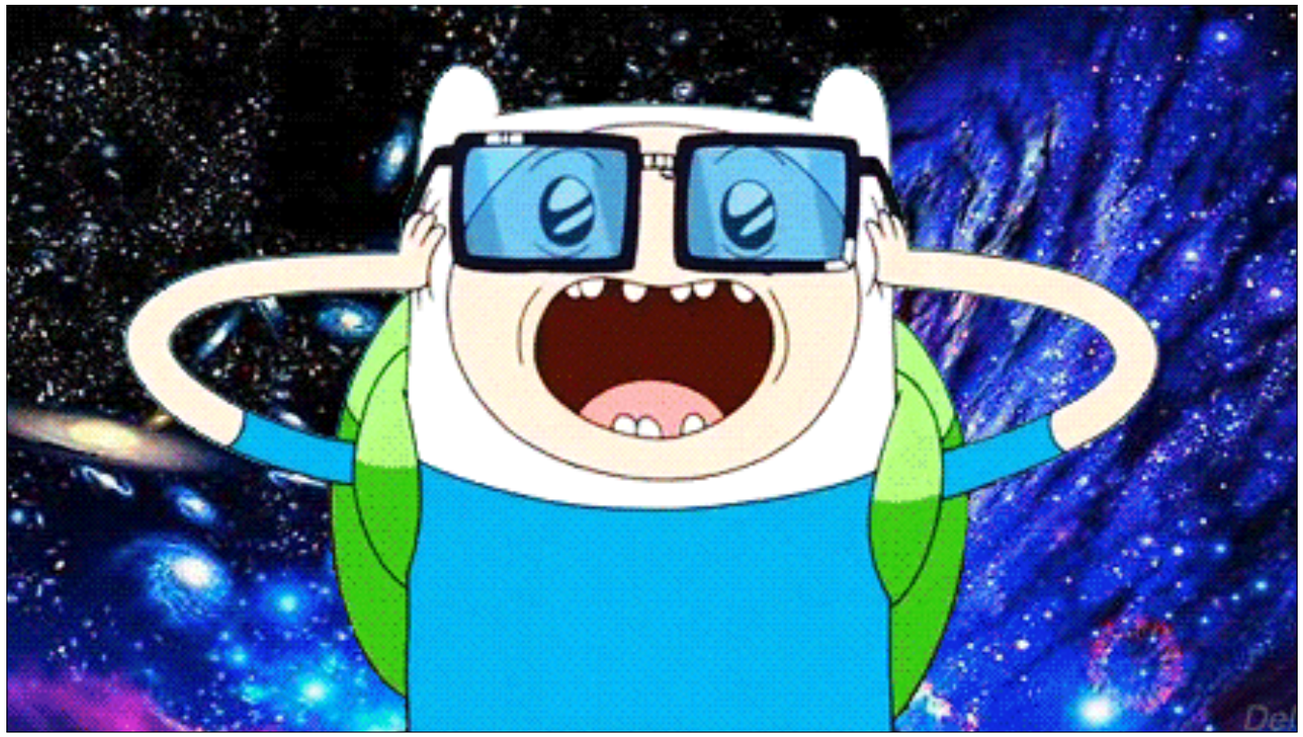
Web workers **still** can:

- **Access** the **navigator** object
- Use the **read-only location** object
- Execute **XMLHttpRequest** to send HTTP or HTTPS requests (please keep that in mind for later)
- Set a **time** or **interval** for an activity.



- **Access** the **application cache** (Keep that one in mind too!)
- **Import** external **scripts**
- **Spawn** other web workers (The **child**, also known as the **subworker** must have the **same origin** as the **main** page and be **placed** in the **same location** as the **parent** worker.)

// An origin is defined as a combination of URI scheme, hostname, and port number.[1]



WOW!

This is a **lot** we can use **still**!

So, how do we **start**?

Web workers

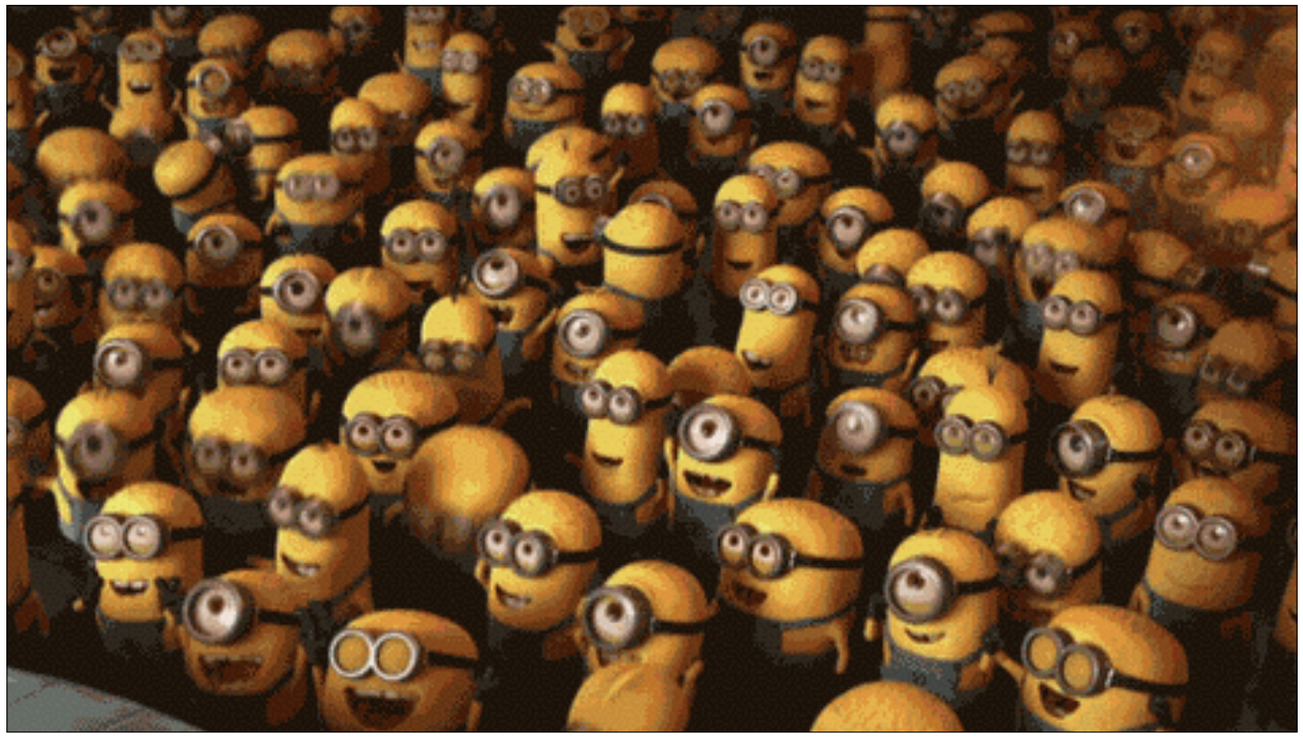
- amazing litte helpers

@misprintedtype

So **basically** a **workers** are this **amazing** litte **helpers**…

… that you **spawn**, then **you** can **pass** them some **work**…

…and **after** they are **done**, you get **back** a result.



You can **spawn** as many **workers** as you like, but be **careful**!
They do **not block** your **UI**, but they can **use** a **ton** of your **CPU** and basically freeze it all.

I mean… you know how chaotic **minions**… äh, I mean busy **workers** can **get**!

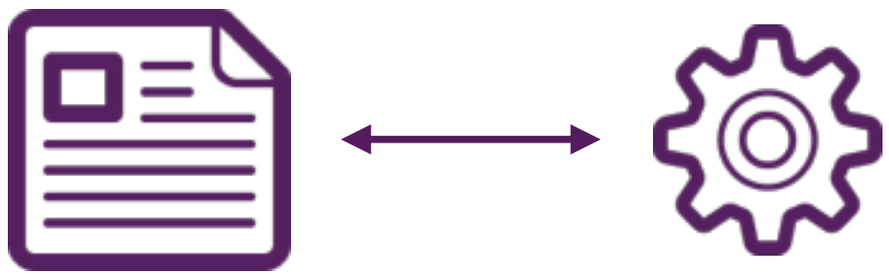
2 types of workers

- dedicated workers
- shared workers

@misprintedtype

There are **two** types of workers. **Dedicated** and **shared** workers.

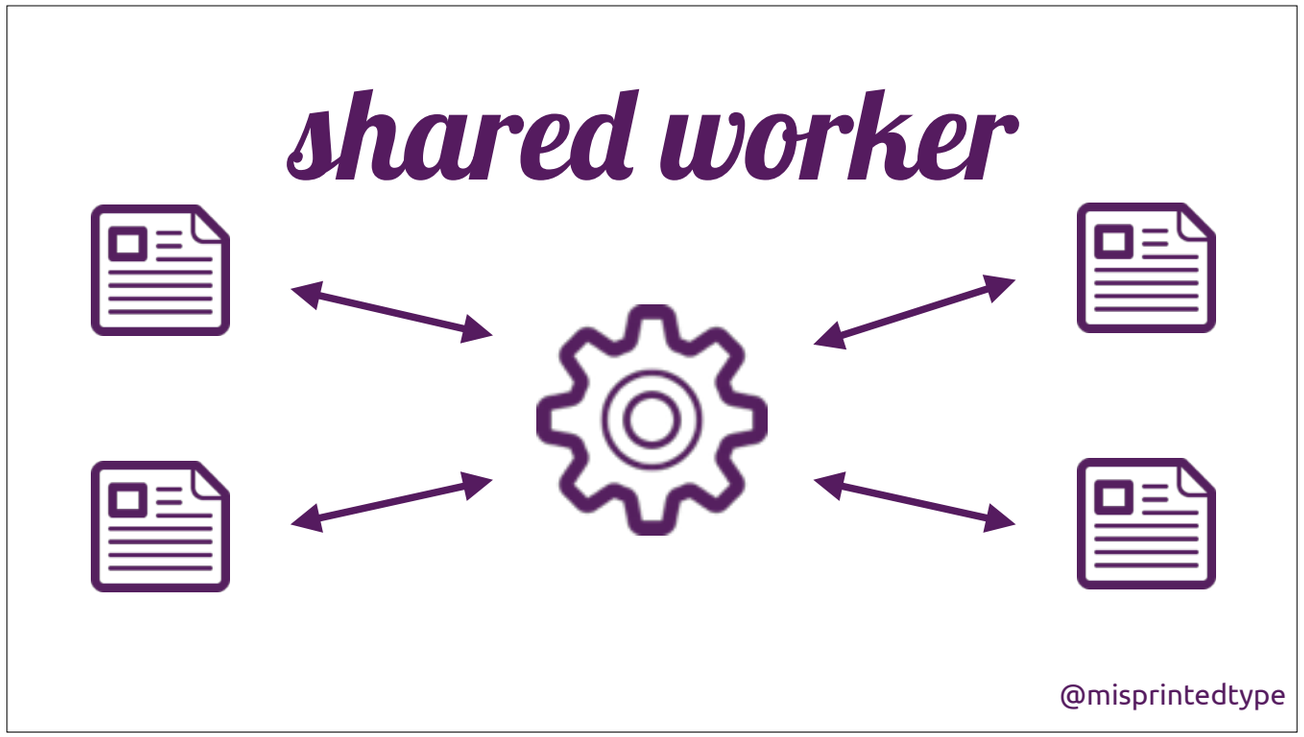
dedicated worker



@misprintedtype

Dedicated web workers are **linked** to their creator.

The creator is the **script** that **called** and **loaded** them.



Shared web workers allow **any** number of scripts to **communicate** with a **single** worker.

They are **identified** in two ways: either by the **URL** of the script used to create it or by **explicit** name.

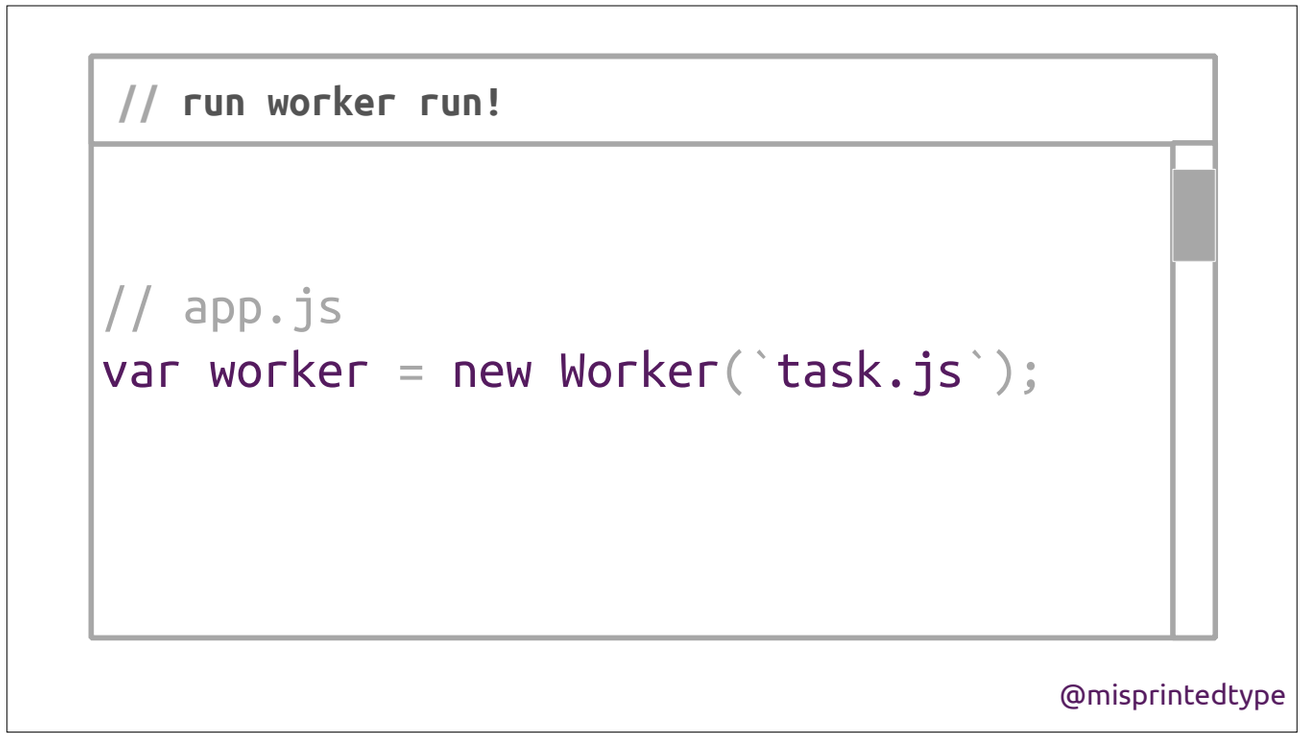
When we talk here about **WebWorkers**, we talk about **dedicated** workers.

```
// available?


function areWorkersAvailable() {
    return !!window.Worker;
}
```

@misprintedtype

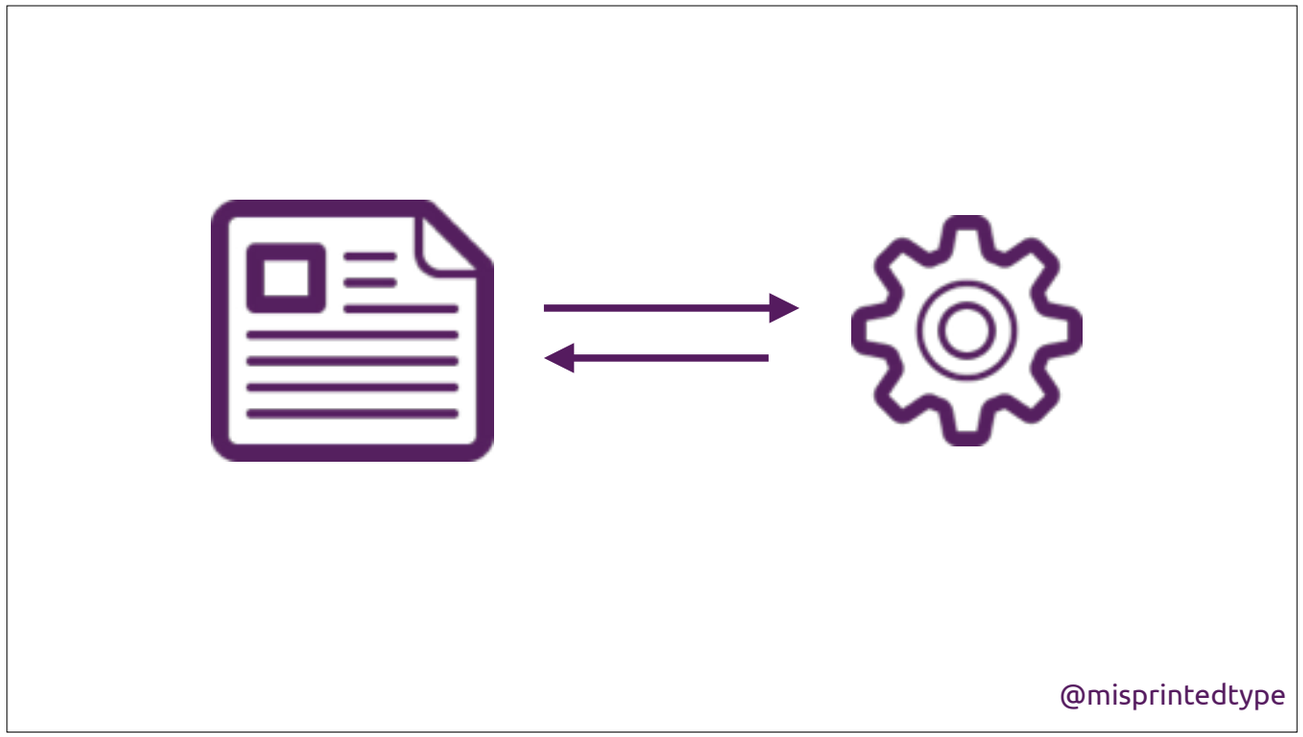
In order to find out if we can **use** web **workers**, we need to **check** if there is a Worker **property** on the global **window** object.

If our browser **doesn't** support the Web Worker **API**, the Worker **property** will be **undefined**.



Web Workers run in an **isolated** thread. As a result, the code that they execute needs to be **contained** in a **separate** file.

The first thing to do is **create** a new Worker **object** in your **main** script. The **constructor** takes the **name** of the worker script.



If the specified file **exists**, the **browser** will **spawn** a new worker thread, which is **downloaded** asynchronously.

The worker will **not** begin until the file has **completely** downloaded and **executed**.

If the **path** to your worker **returns** an 404…

… the worker will **fail silently**.

PSSSSST!!! I said, **silently**!

demo

```
// map.js
http://slides.html5rocks.com/#web-workers
```


-> OPEN in Browser

more demos

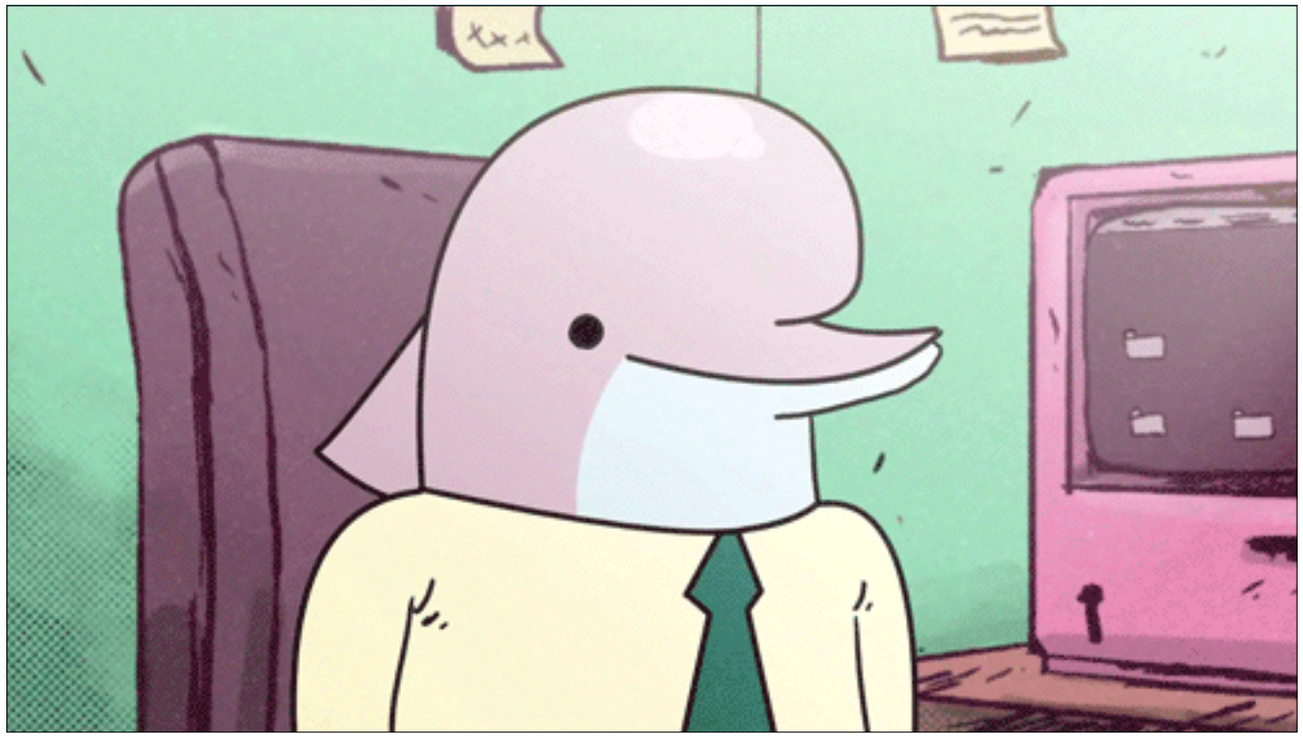
```
// examples.js
http://greenido.github.io/Web-Workers-Examples-/
```

@misprintedtype

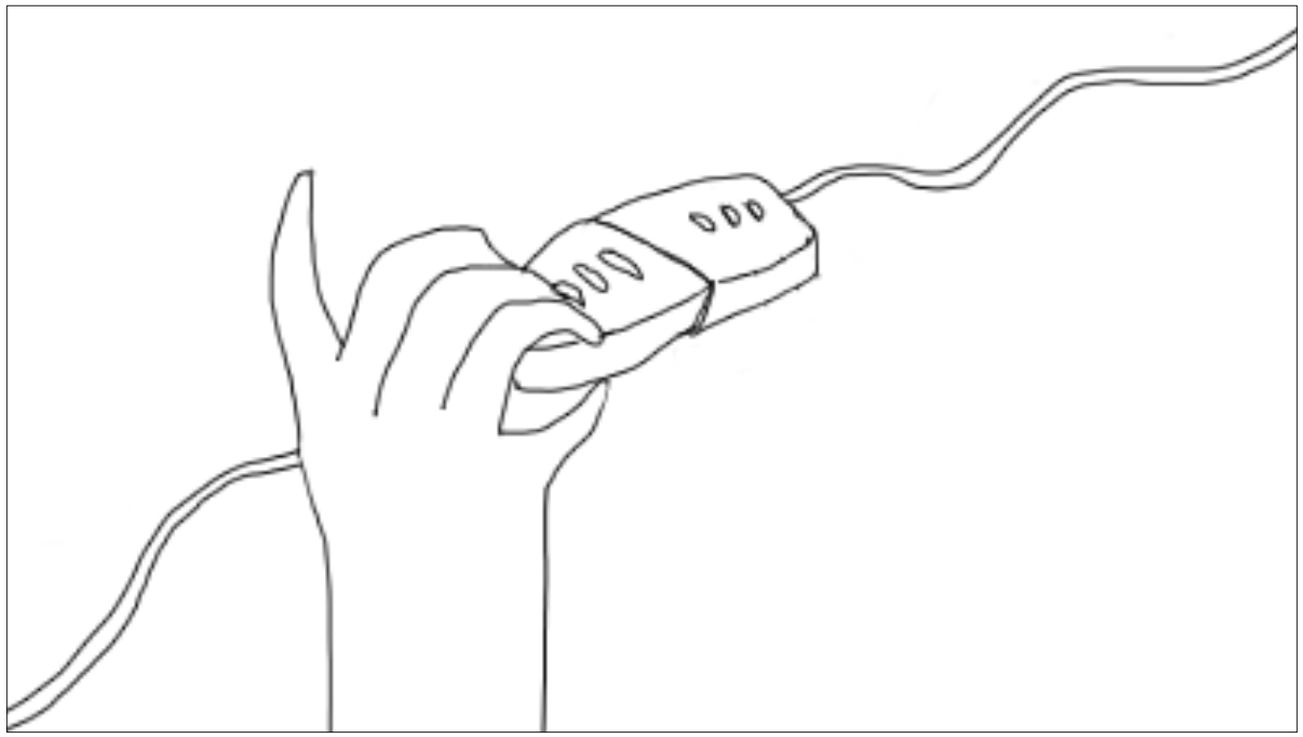
-> OPEN in Browser

To **run** workers **locally**, you need to **enable CORS**.

// enable CORS (Cross-Origin-Recource-Sharing)



So, we've **talked** about the **worker**. Let's now take a **focused** look at the **Service Worker**.

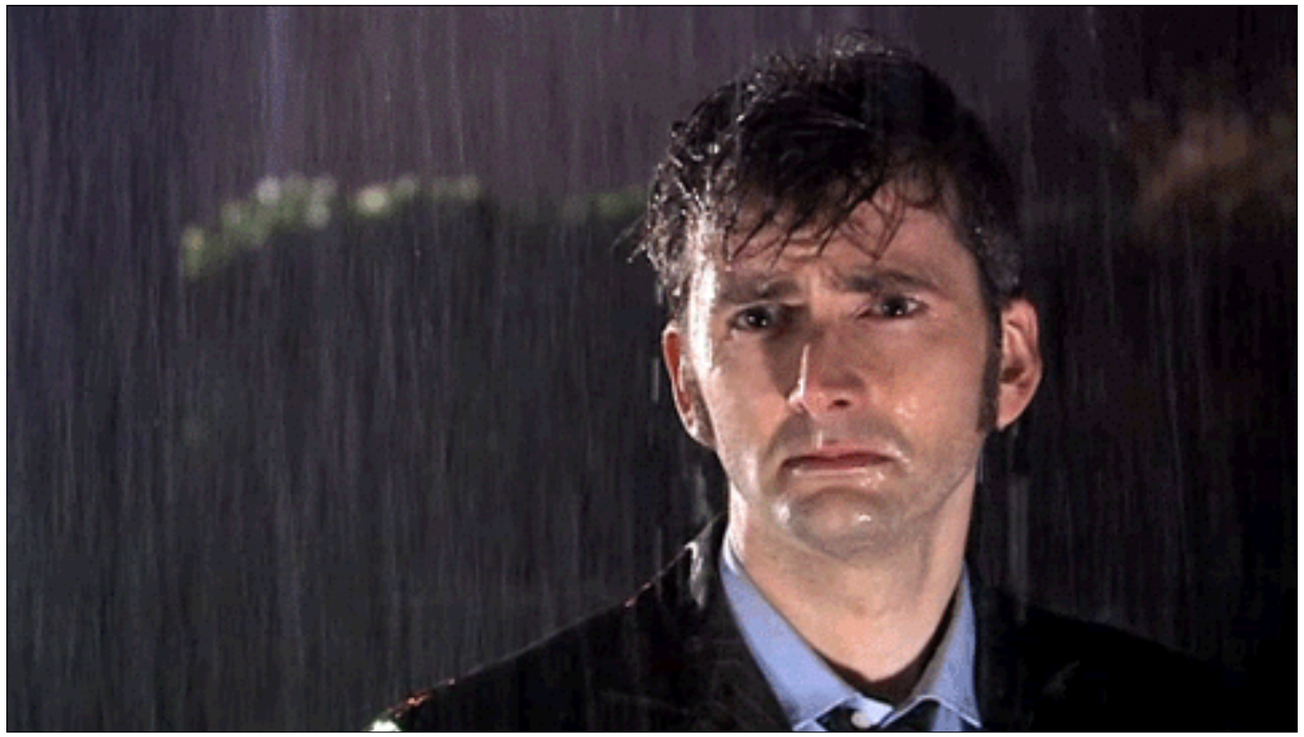


The offline first **concept** became one of the most **important** ones to follow.

Jake Archibald **got** it **right** the **other** day.

-> read!

This **means**, **every** web app should to **work without** a connection **after** the very **first** visit and just use the **network** for **get / set** new **data**. Everything **else** is online-first.



And we **already** know, that **AppCache** is a **douche bag** and has a **LOT** of **issues**, we need to **handle**.

The **major** issue with App Cache is not just the number of **gotcha's.**

The design is working really well for **single page web apps**, but **not** for **multi-page sites**.

Service Worker

@misprintedtype

Service Worker is a **new** browser **feature** that provide **event-driven** scripts that run **independently** of web pages.

It's a **programmable** network **proxy**, allowing you to **control** how network **requests** from your page are **handled** and help **developers** to build **URL-friendly**, always-**available** applications in a sane and layered way.

But **why** can't we **use** web workers for **that**?

Service Worker

Why?

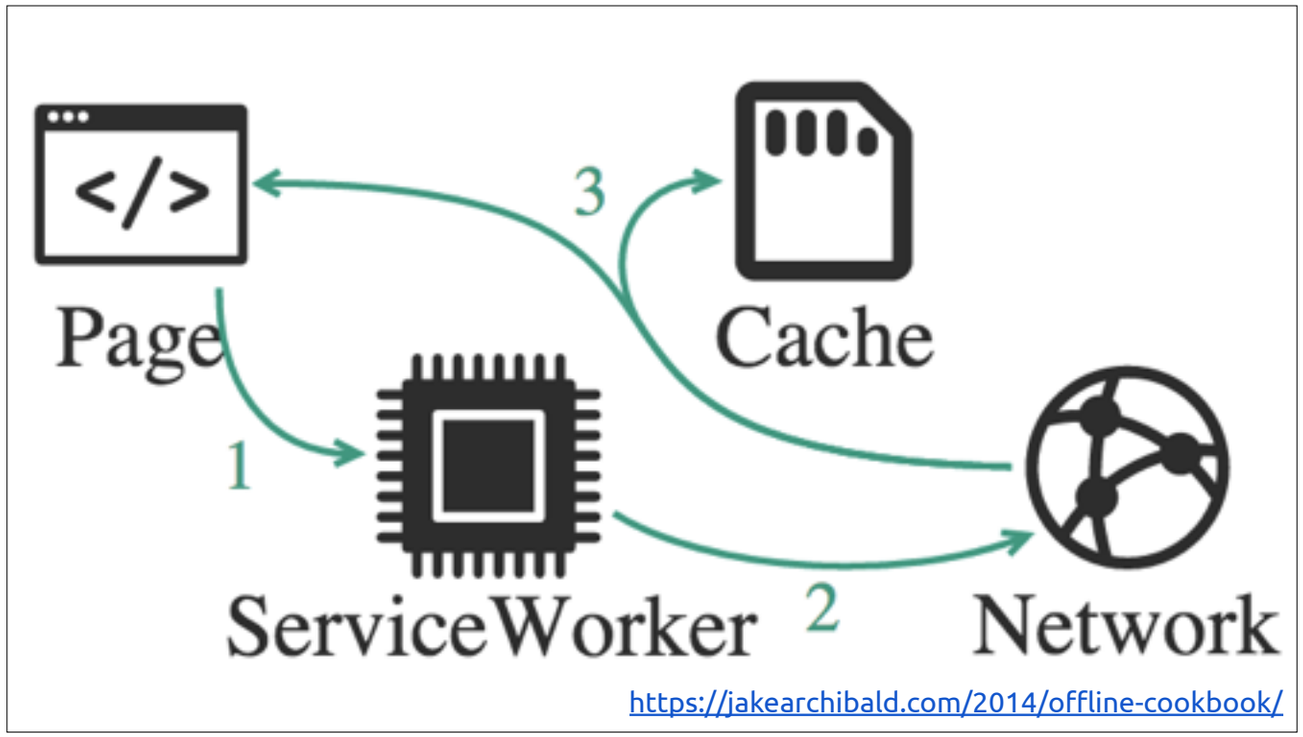
@misprintedtype

The **difference** is…

With **service workers** you can **fetch** your **assets** and **serve** them to your **caches**. So we can **replace** the AppCache.

On **top** of that, we can **outsource** the heavy **operations**, like **pulling** the huge **pile** of **JSON**.

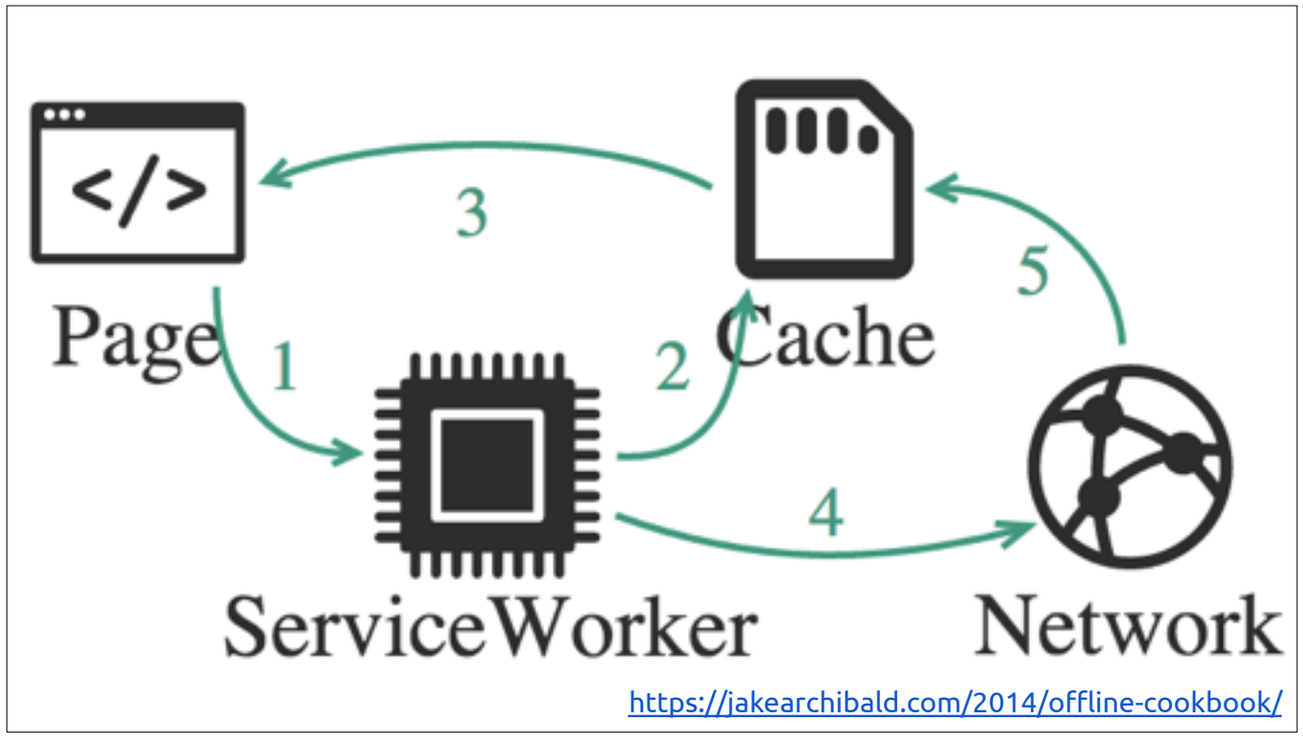
Imagine…



https://jakearchibald.com/2014/offline-cookbook/

0. your page **requests** something **new**
1. the **request** gets **through** the **service** worker
2. then to the **network**, because it's **new**
3. the network **serves** back the **response** to the page **AND** the cache

(because the service worker **told** the **cache** to **fetch** the **response**)
Great for **assets**…



Here,
we have the **same** use **case**, **but** the service worker **knows,** we already **stored** the requested **asset** or **data** in the **cache** already… So it **serves** straight from the **cache**.

You can also **add** the **behavior**, it should go to the **network** and **check**, if there is an **update** available.

Awesome right? This **saves** a lot of **time** and **requests**, because you are able to **control** the whole **behavior**.

And one more **feature**, how it **saves** resources…

Unlike **other** workers, Service Workers can be **shut down** at the **end** of **events**.

Service Worker

shut down at end of events

@misprintedtype

It will be **terminated** when not in use, and **restarted** when it's next **needed.**

You **can’t** rely on **global** state within a service worker's **onfetch** and **onmessage** handlers.

If there is information that you need to **persist** and **reuse** across **restarts**, service workers do have **access** to the **IndexedDB** API.

Service Worker

scriptable caches

@misprintedtype

ServiceWorkers also have **scriptable caches**. Along with the ability to **respond** to network **requests** from certain web pages via script, this **provides** a way for applications to "**go offline**".

The **main** advantage here is, that Service Worker doesn't **just cache** the **files.** It also supports **background sync** && increases your **performance** enormously, it also gives you **support** for **push notifications**

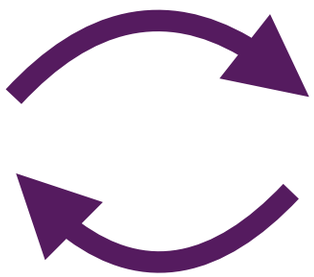
Service Worker

promises

@misprintedtype

Service workers make **extensive** use of **promises**, so you should be **familiar** with the concept, **before** you start using Service Worker.

SW lifecycle

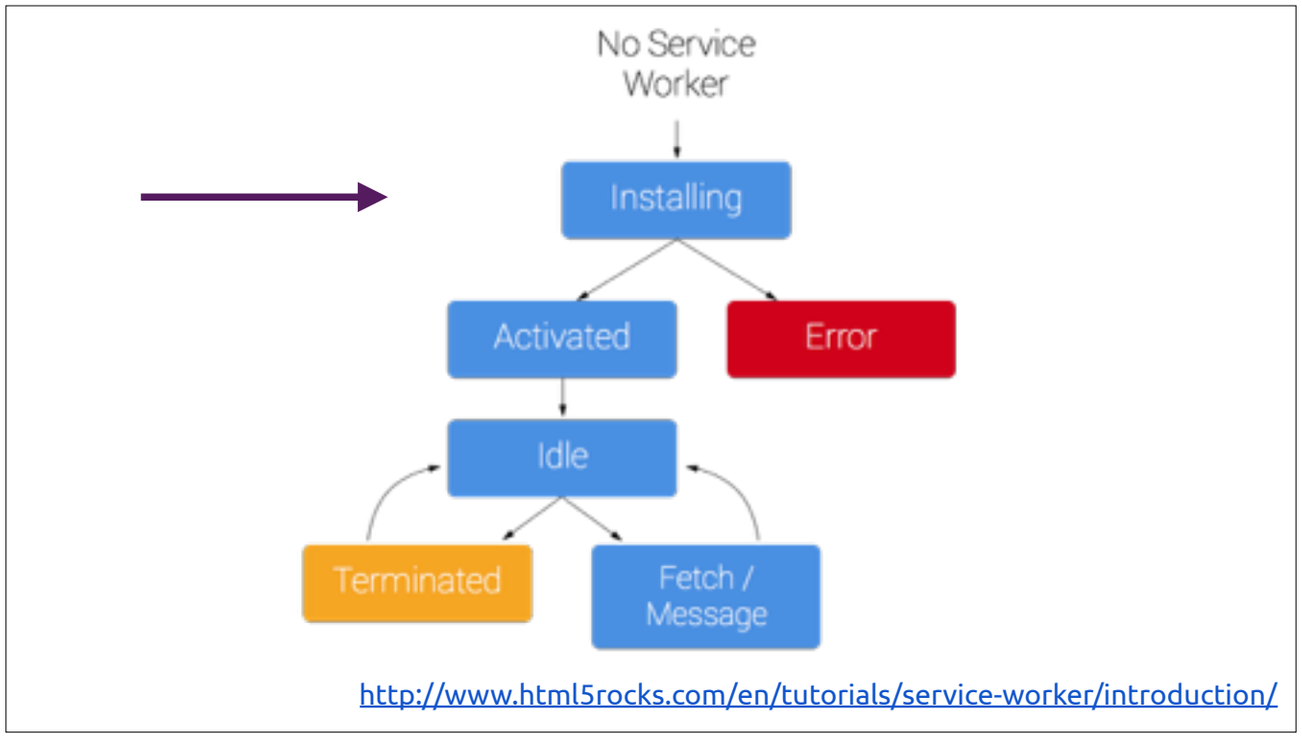


@misprintedtype

A service worker has a **lifecycle** that is completely **separate** from your web **page**.

To **install** a service worker for your site, you need to **register** it, like with we've **seen before** with the workers.

Registering a service worker will cause the browser to **start** the service worker **install step** in the background.

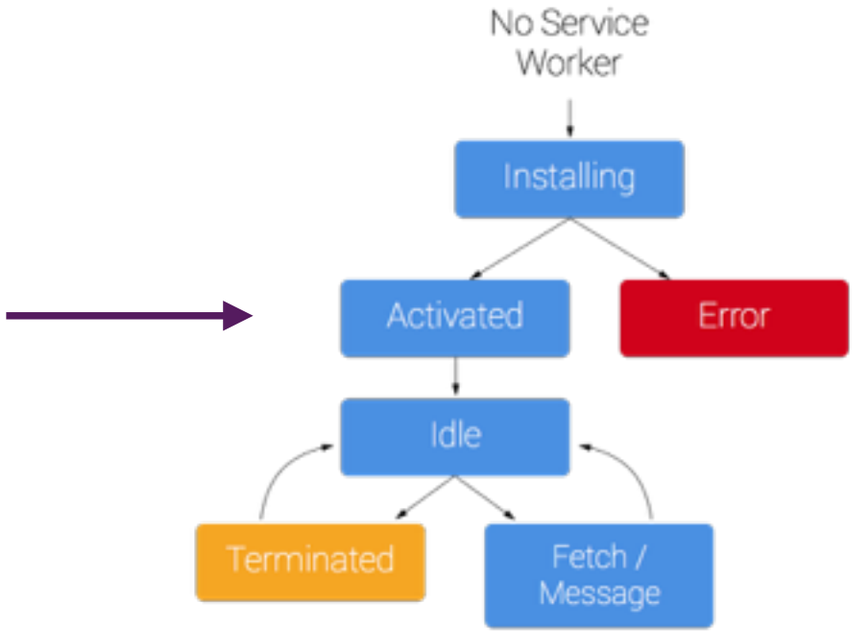


http://www.html5rocks.com/en/tutorials/service-worker/introduction/

Typically during the install step, you'll want to **cache** some **static assets**.

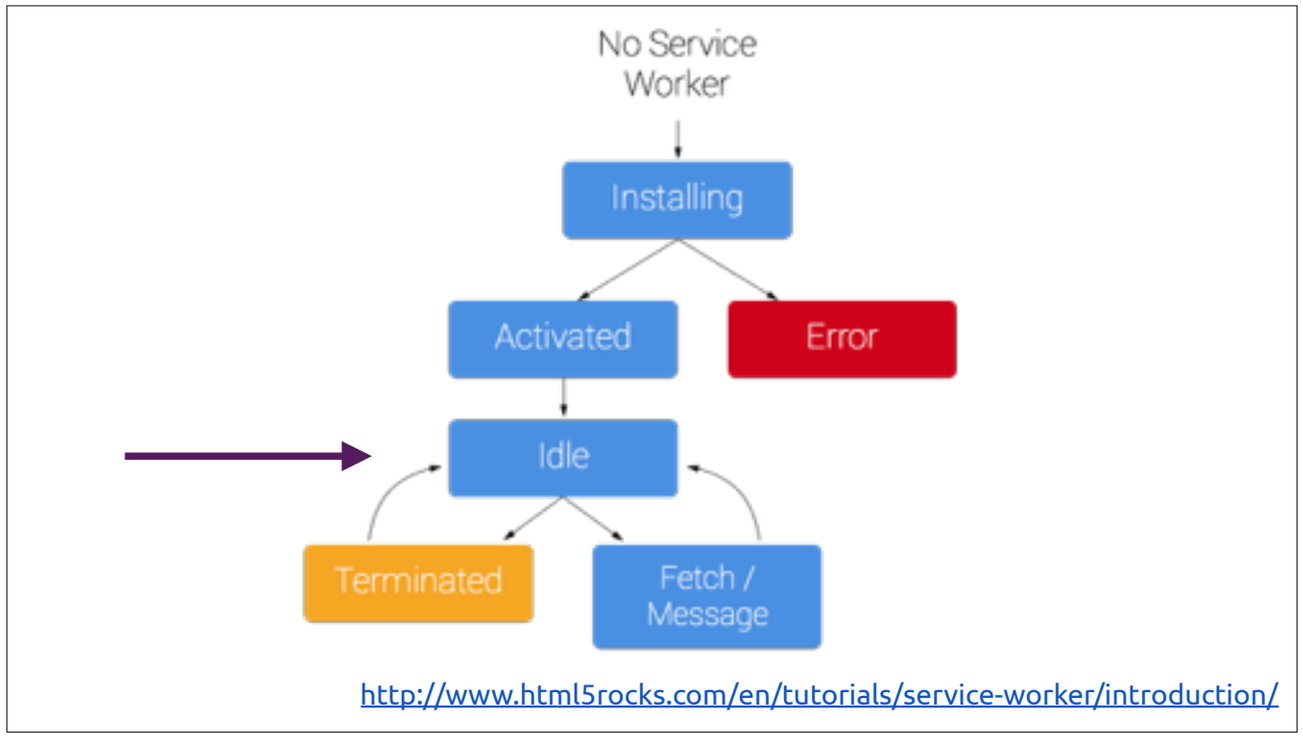
If all the files are cached **successfully**, then the service worker becomes **installed**. If any of the files **fail** to download and cache, then the install **step** will **fail** and the service worker **won't** activate.

No worries about that. it'll **try again** next time.



http://www.html5rocks.com/en/tutorials/service-worker/introduction/

When we're installed, the **activation** step will **follow** and this is a **great** opportunity for **handling** any management of **old caches**



http://www.html5rocks.com/en/tutorials/service-worker/introduction/

After the activation step, the service worker will **control** all pages that fall **under** its **scope.** The page that **registered** the service worker for the **first** time won't be **controlled** until it's **loaded again**.

Once a service worker is in **control**, it will be in one of **two states**: either the service worker will be **terminated** to save memory, or it will **handle fetch** and **message events** which occur when a network **request** or **message** is made from your page.

https://jakearchibald.github.io/isserviceworkerready/

@misprintedtype

Service worker is **still** in **development**, but pretty **far** in **progress** already.

It’s partially shipped for Chrome, Firefox already. Edge and Safari are working on implementing it.

If you want to dive deeper into Service Worker, watch the talk by **Jake Archibald** from JSConfEU **last** year. And **Jake** is really **fun** and an **amazing** guy, his **talks** are **legendary**.

There is the **link**.

offline cookbook

http://jakearchibald.com/2014/offline-cookbook/

@misprintedtype

Also there is the offline cookbook written by Jake you can check out.

Awesome! Now we **have** all the **files**…

…let's finally **take care** the **important** stuff… and just take a specific look at the **data**!

Do not harm humans!

(first law of robotics)

@misprintedtype

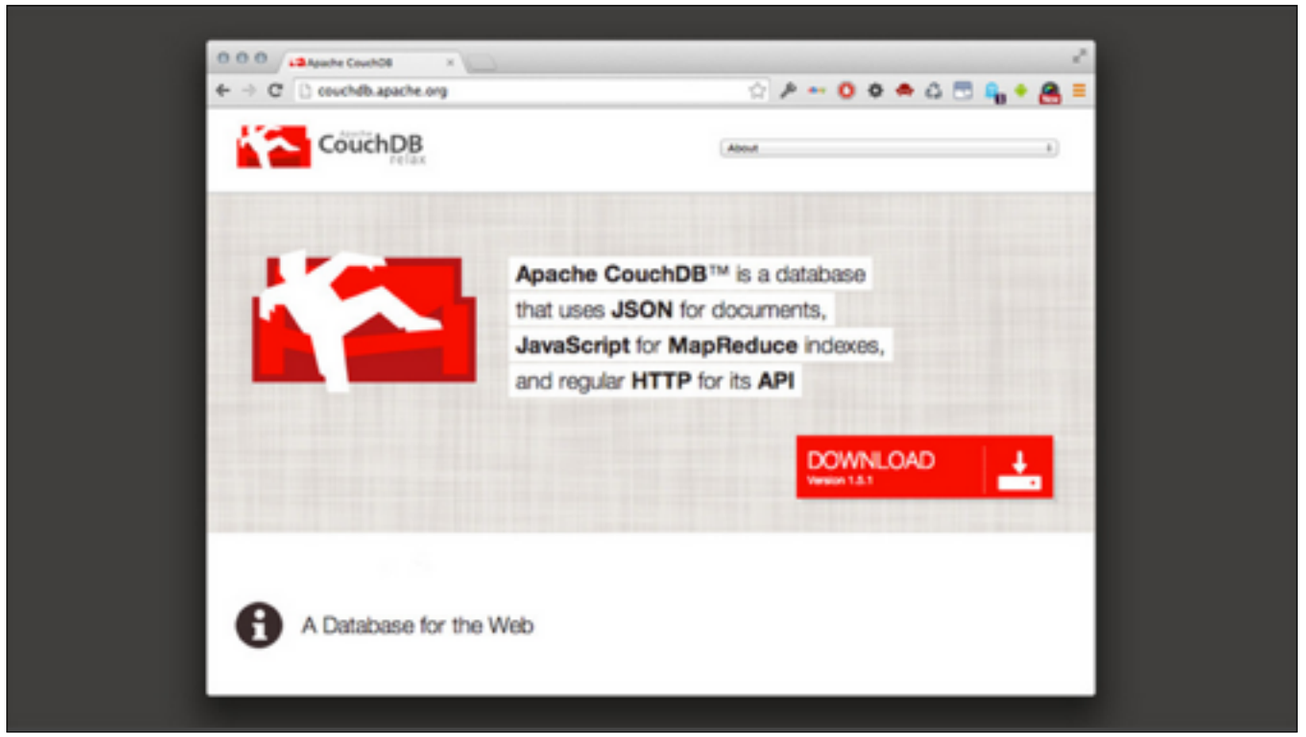
The first **law** of **robotics** is… do not **harm** any **human**!

Do you **know** the first **law** of offline **first**?

First **law** of Offline First is do **not** lose **data**!

While **decoupled** the client and server side, we really need to **store** it on **both sides**.

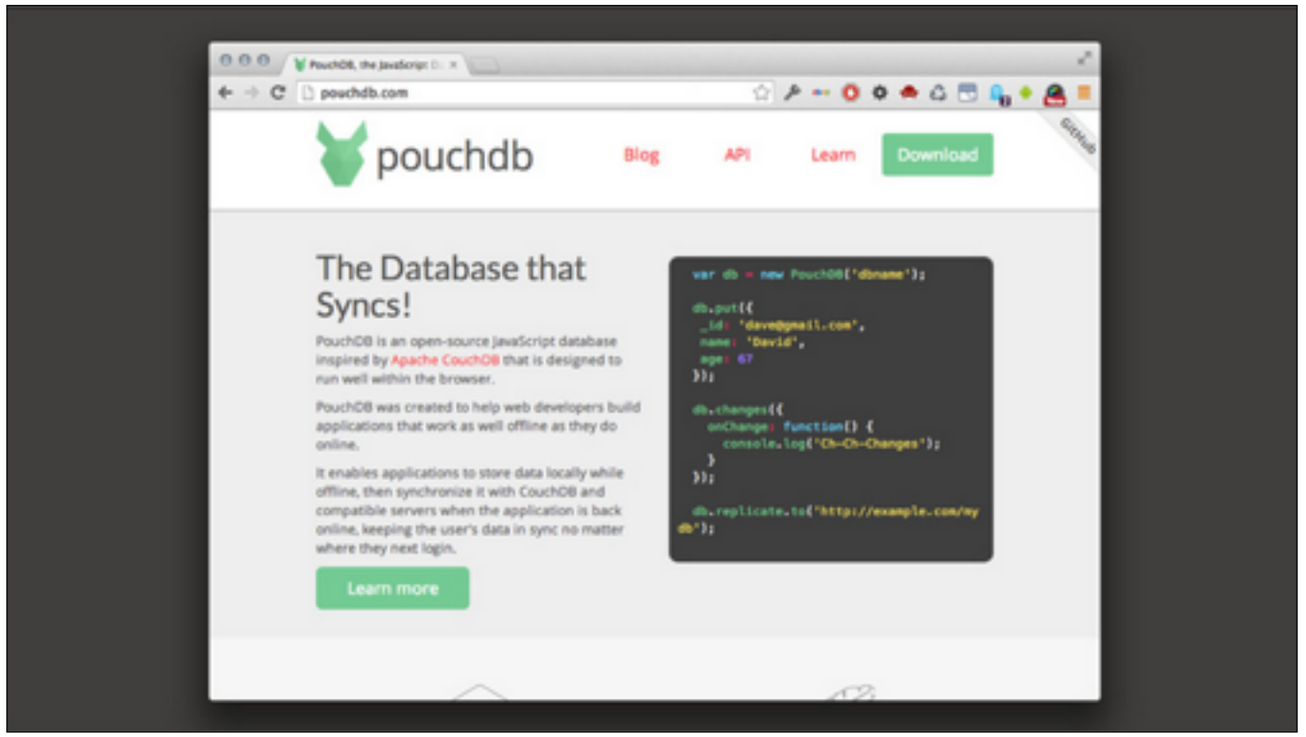
There for we **trust PouchDB** for the **client** and **CouchDB** on the **server** side. They are the **perfect team** to make **offline** first quick and **easy**.



CouchDB a NoSQL database that **replicates**, which is **exactly** what we **want** for sync.

CouchDB comes with a suite of **features**, such as on-the-fly **document transformation** and real-time change **notifications**, we use for our **message** and **task** system.

It also has a very basic **version control system** which helps you to **manage** your data.
So when you data is out of **sync**, it's making automatic **intelligent** decisions and helps you to **merge** the data.

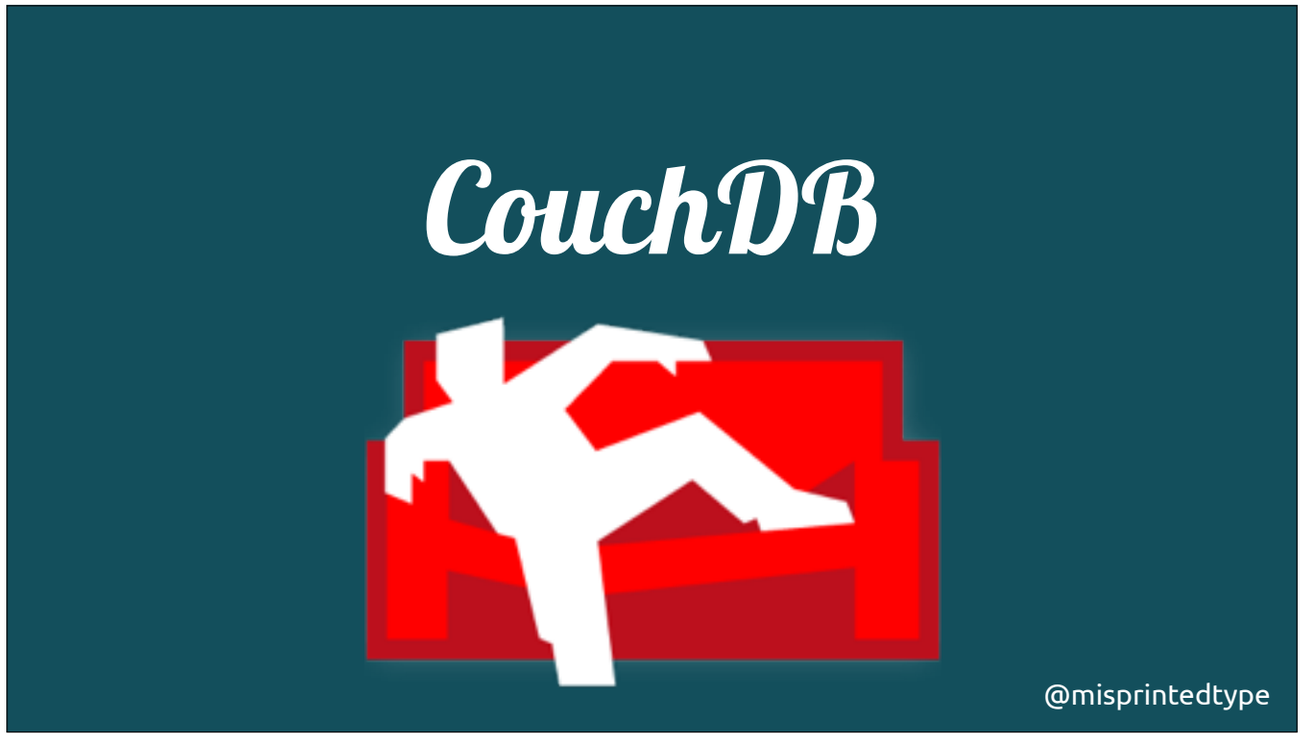


PouchDB is an **open-source JavaScript database** inspired by CouchDB that is **designed** to run well **within** the browser and **store** all your data **locally** while offline.

Per **default** it uses **IndexedDB** in Chrome and Firefox and **WebSQL** in Safari to **store** the data. You also can **add adapters** for **local** and **session** storage.

It sync's with **CouchDB** and **compatible servers**, which speak the CouchDB **replication protocol,** like **Cloudant** and the Couchbase Sync **Gateaway**… when the **application** is back **online**.

… It also **works** on **mobile devices** and for native apps e.g. with **phonegap**, **cordova** and many more…

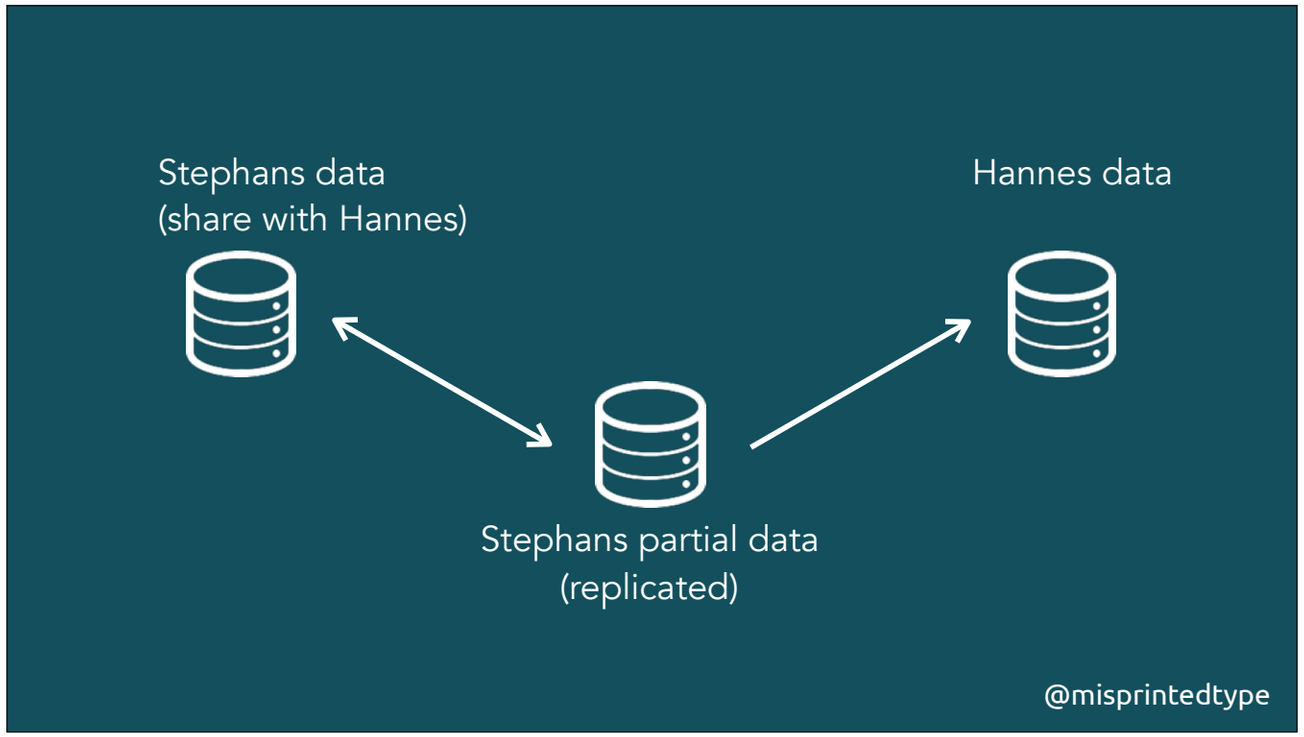


The **special** thing about **hoodie** is, that **every user** has his **own database** and it's **private** by **default**. So we **solve** a lot of **problems** from **the get go.**

When we have **one database** per **user**, we always know **where** to **sync** to.
When the **version** of the **document** isn't **up-to-date**, we can **check** by **opaque document revision id** which to sync **first** and how to **merge** the data without **real** conflicts.

When we receive **partial** data, we **store** the **version** of the **document** and can **grab** that before **merging**.

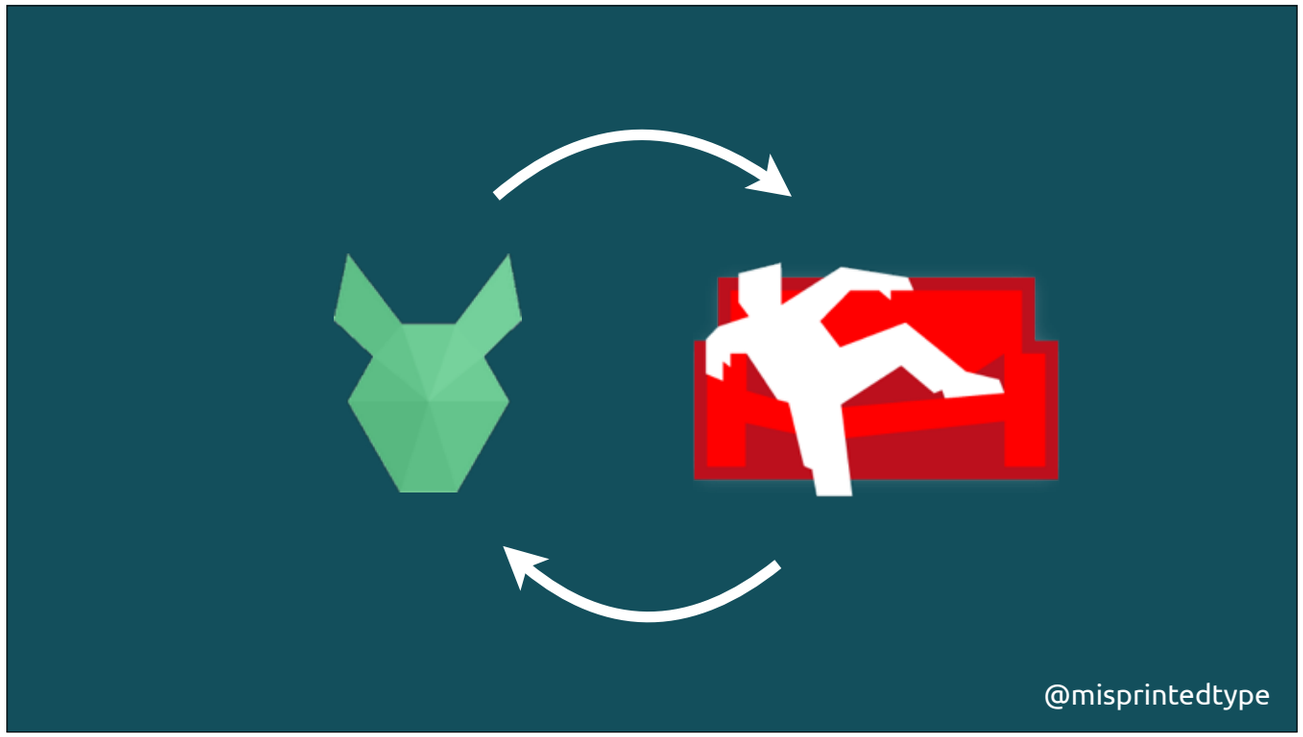
Thanks to the **event listeners** and **task**, we can **synchronize** the data **across** devices… this means **multi** device **usage** on-the-fly.



You want to **share** your **data**? Sure!

When we **want** to **share** some of their **private** data, we can „**publish**“ it.
So it get’s **replicated** and others are able to **read** / **read** + **manipulate** the data.

Then we are **able** to **sync** it again in our **application logic** into their **private** storage.



PouchDB is also a CouchDB **client**, and you should be able to **switch** between a **local** database or an **online** CouchDB **instance** without **changing** any of your **application's code**.

A **question** I get asked a **lot** is… **how much** data can PouchDB **store**?

browser	storage	limitation	confirm
Firefox	IndexedDB	unlimited	y
Chrome / Opera / Android 4.4+	IndexedDB	% of storage	y
IE 10+	SQLite	250MB	n
Mobile Safari	WebSQL	50MB	n
Safari	WebSQL	5MB -> 500MB	y
Android 4.3 and lower	IndexedDB	200MB	n

A **question** I get asked a **lot** is… **how much** data can PouchDB **store**?

In **Firefox**, PouchDB uses **IndexedDB**, which will **ask** the user if data can be **stored** the **first** time it is attempted, **then** every **50MB** after. The amount that can be stored is **unlimited**.

Chrome calculates the amount of **available** storage on the user's **hard drive** and uses that to calculate a **limit**.

Opera should work the **same** as **Chrome**, since it's based on Blink.

Internet Explorer 10+ has a **hard 250MB** limit.

Mobile Safari on iOS has a **hard 50MB** limit, while **desktop Safari** will **prompt** users wanting to store **more** than **5MB** up to a limit of **500MB**.

Android works the same as **Chrome** as of **4.4**+, while **older** version can store up to **200MB**.

In **PhoneGap**/**Cordova**, you can have **unlimited** data on **both** iOS and Android by using the **SQLite Plugin**.

I hope, i could give you a **small insight** in the **concept** of offline first and how to build **applications** that handle offline first **not** as a **bug**, **but** as a **feature**!

If you’d like to **say hi**, have any **questions** or want to **grab** some **hoodie** stickers, I’ll be **around**.

hoodie <3 you

@misprintedtype / @hoodiehq

Thank you <3