The localStorage API is simple to use but synchronous, so if you use it too often, your app will soon begin to freeze.

The indexedDB API is asynchronous and efficient, but it's a mess to use: you'll soon be caught by the callback hell, as it does not support Promises yet.

Mozilla has done a very great job with the localForage library: a simple API based on native localStorage, but internally stored via the asynchronous indexedDB for performance. But it's built in ES5 old school way and then it's a mess to include into Angular.

This module is based on the same idea as localForage, but built in ES6+ and additionally wrapped into RxJS Observables to be homogeneous with other Angular modules.

angular-web-storage

@ngx-pwa/local-storage uses indexedDB internally to make the operations asynchronous.

Application -> IndexedDB -> ngStorage -> localStorage

**@ngx-pwa/local-storage**

**Async local storage for Angular**

Efficient client-side storage module for Angular apps and Progressive Wep Apps (PWA):

* **simplicity**: based on native localStorage API,
* **perfomance**: internally stored via the asynchronous indexedDB API,
* **Angular-like**: wrapped in RxJS Observables,
* **security**: validate data with a JSON Schema,
* **compatibility**: works around some browsers issues,
* **documentation**: API fully explained, and a changelog!
* **reference**: 1st Angular library for client-side storage according to [ngx.tools](https://ngx.tools/" \l "/search?q=local%20storage).

**By the same author**

* [Angular schematics extension for VS Code](https://marketplace.visualstudio.com/items?itemName=cyrilletuzi.angular-schematics) (GUI for Angular CLI commands)
* Other Angular library: [@ngx-pwa/offline](https://github.com/cyrilletuzi/ngx-pwa-offline)
* Popular [Angular posts on Medium](https://medium.com/@cyrilletuzi)
* Follow updates of this lib on [Twitter](https://twitter.com/cyrilletuzi)
* [Angular onsite trainings](https://formationjavascript.com/formation-angular/) (based in Paris, so the website is in French, but [my English bio is here](https://www.cyrilletuzi.com/en/web/) and I'm open to travel)

**Why this module?**

For now, Angular does not provide a client-side storage module, and almost every app needs some client-side storage. There are 2 native JavaScript APIs available:

* [localStorage](https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage)
* [indexedDB](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API)

The localStorage API is simple to use but synchronous, so if you use it too often, your app will soon begin to freeze.

The indexedDB API is asynchronous and efficient, but it's a mess to use: you'll soon be caught by the callback hell, as it does not support Promises yet.

Mozilla has done a very great job with the **[localForage library](http://localforage.github.io/localForage/)**: a simple API based on native localStorage, but internally stored via the asynchronous indexedDB for performance. But it's built in ES5 old school way and then it's a mess to include into Angular.

This module is based on the same idea as localForage, but built in ES6+ and additionally wrapped into [**RxJS Observables**](http://reactivex.io/rxjs/) to be homogeneous with other Angular modules.

**Getting started**

Install the right version according to your Angular one via **[npm](http://npmjs.com/)**:

*# For Angular 8:*

npm install @ngx-pwa/local-storage

*# For Angular 6 & 7:*

npm install @ngx-pwa/local-storage@6

The following second setup step is:

* **only for version >= 8**,
* not required for the lib to work,
* **strongly recommended for all new applications**, as it allows interoperability and is future-proof, as it should become the default in a future version,
* **prohibited in applications already using this lib and already deployed in production**, as it would break with previously stored data.

**import** { StorageModule } **from** '@ngx-pwa/local-storage';

@NgModule({

  imports: [

    StorageModule.forRoot({

      IDBNoWrap: true,

    })

  ]

})

**export** class AppModule {}

**Must be done at initialization, ie. in AppModule, and must not be loaded again in another module.**

**Upgrading**

If you still use the old angular-async-local-storage package, or to update to new versions, see the [**migration guides**](https://github.com/cyrilletuzi/angular-async-local-storage/blob/HEAD/MIGRATION.md)**.**

Versions 4 & 5, which are *not* supported anymore, needed an additional setup step explained in [**the old module guide**](https://github.com/cyrilletuzi/angular-async-local-storage/blob/HEAD/docs/OLD_MODULE.md).

**API**

2 services are available for client-side storage, you just have to inject one of them were you need it.

**LocalStorage**

**import** { LocalStorage } **from** '@ngx-pwa/local-storage';

@Injectable()

**export** class YourService {

  constructor(private localStorage**:** LocalStorage) {}

}

This service API follows the [**native localStorage API**](https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage), except it's asynchronous via [**RxJS Observables**](http://reactivex.io/rxjs/):

class LocalStorage {

  length**:** Observable<number>;

  getItem(index**:** string, schema**?:** JSONSchema)**:** Observable<unknown> {}

  setItem(index**:** string, value**:** any)**:** Observable<true> {}

  removeItem(index**:** string)**:** Observable<true> {}

  clear()**:** Observable<true> {}

}

**StorageMap**

**import** { StorageMap } **from** '@ngx-pwa/local-storage';

@Injectable()

**export** class YourService {

  constructor(private storageMap**:** StorageMap) {}

}

New *since version 8* of this lib, this service API follows the [**native Map API**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) and the new upcoming standard **[kv-storage API](https://github.com/WICG/kv-storage)**, except it's asynchronous via [**RxJS Observables**](http://reactivex.io/rxjs/).

It does the same thing as the LocalStorage service, but also allows more advanced operations. If you are familiar to Map, we recommend to use only this service.

class StorageMap {

  size**:** Observable<number>;

  get(index**:** string, schema**?:** JSONSchema)**:** Observable<unknown> {}

  set(index**:** string, value**:** any)**:** Observable<undefined> {}

  delete(index**:** string)**:** Observable<undefined> {}

  clear()**:** Observable<undefined> {}

  has(index**:** string)**:** Observable<boolean> {}

  keys()**:** Observable<string> {}

}

**How to**

The following examples will show the 2 services for basic operations, then stick to the StorageMap API. But except for methods which are specific to StorageMap, you can always do the same with the LocalStorage API.

**Writing data**

let user**:** User **=** { firstName: 'Henri', lastName: 'Bergson' };

this.storageMap.set('user', user).subscribe(() => {});

*// or*

this.localStorage.setItem('user', user).subscribe(() => {});

You can store any value, without worrying about serializing. But note that:

* storing null or undefined makes no sense and can cause issues in some browsers, so the item will be removed instead,
* you should stick to JSON data, ie. primitive types, arrays and *literal* objects. Map, Set, Blob and other special structures can cause issues in some scenarios. See the [serialization guide](https://github.com/cyrilletuzi/angular-async-local-storage/blob/HEAD/docs/SERIALIZATION.md) for more details.

**Deleting data**

To delete one item:

this.storageMap.delete('user').subscribe(() => {});

*// or*

this.localStorage.removeItem('user').subscribe(() => {});

To delete all items:

this.storageMap.clear().subscribe(() => {});

*// or*

this.localStorage.clear().subscribe(() => {});

**Reading data**

this.storageMap.get('user').subscribe((user) => {

  console.log(user);

});

*// or*

this.localStorage.getItem('user').subscribe((user) => {

  console.log(user);

});

Not finding an item is not an error, it succeeds but returns:

* undefined with StorageMap

this.storageMap.get('notexisting').subscribe((data) => {

  data; *// undefined*

});

* null with LocalStorage

this.localStorage.getItem('notexisting').subscribe((data) => {

  data; *// null*

});

Note you'll only get *one* value: the Observable is here for asynchrony but is not meant to emit again when the stored data is changed. And it's normal: if app data change, it's the role of your app to keep track of it, not of this lib. See [**#16**](https://github.com/cyrilletuzi/angular-async-local-storage/issues/16) for more context and [**#4**](https://github.com/cyrilletuzi/angular-async-local-storage/issues/4) for an example. A watch() method may come soon (see [**#108**](https://github.com/cyrilletuzi/angular-async-local-storage/pull/108))

**Checking data**

Don't forget it's client-side storage: **always check the data**, as it could have been forged.

You can use a [**JSON Schema**](http://json-schema.org/) to validate the data.

this.storageMap.get('test', { type: 'string' }).subscribe({

  next: (user) => { */\* Called if data is valid or `undefined` or `null` \*/* },

  error: (error) => { */\* Called if data is invalid \*/* },

});

**See the**[**full validation guide**](https://github.com/cyrilletuzi/angular-async-local-storage/blob/HEAD/docs/VALIDATION.md)**to see how to validate all common scenarios.**

**Subscription**

You *DO NOT* need to unsubscribe: the Observable autocompletes (like in the Angular HttpClient service).

But **you *DO* need to subscribe**, even if you don't have something specific to do after writing in storage (because it's how RxJS Observables work).

**Errors**

As usual, it's better to catch any potential error:

this.storageMap.set('color', 'red').subscribe({

  next: () => {},

  error: (error) => {},

});

For read operations, you can also manage errors by providing a default value:

**import** { of } **from** 'rxjs';

**import** { catchError } **from** 'rxjs/operators';

this.storageMap.get('color').pipe(

  catchError(() => **of**('red')),

).subscribe((result) => {});

See the [**errors guide**](https://github.com/cyrilletuzi/angular-async-local-storage/blob/HEAD/docs/ERRORS.md) for some details about what errors can happen.

**Map-like operations**

Starting *with version >= 8* of this lib, in addition to the classic localStorage-like API, this lib also provides a Map-like API for advanced operations:

* .keys()
* .has(key)
* .size

See the [**documentation**](https://github.com/cyrilletuzi/angular-async-local-storage/blob/HEAD/docs/MAP_OPERATIONS.md) for more info and some recipes. For example, it allows to implement a multiple databases scenario.

**Quick Tip: Make IndexedDB a Breeze With LocalForage**

May 31st 2017

[**JavaScript**](https://tutorialzine.com/tag/javascript) | [**Quick Tip**](https://tutorialzine.com/tag/quick-tip)

IndexedDB is a local NoSQL database that allows developers to safely store data in the browser. It has great cross-platform support, works with any type of data, and is powerful enough for building apps that work offline.

Although it is probably the best solution for client side storage, IndexedDB has one critical flaw - it's low-level API. Things like transactions, cursors, and a lack of support for promises over-complicate IndexedDB and make it exhausting to work with.

Thankfully, there is a more dev-friendly way!

**LocalForage to the Rescue**

LocalForage is an open-source JavaScript library that makes working with in-browser databases much more enjoyable. On the outside its API looks very similar to [localStorage](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage" \t "_blank), while under the hood it hides the entire arsenal of IndexedDB features.

Compared to the 15 lines of code required to do anything with IndexedDB, with localForage creating a database and accessing its entries comes down to using a simple method. It also adds much needed support for promises plus other helpful utilities.

**Installation**

Adding localForage to a project is quite simple. Either drop it directly into the HTML:

<script src="assets/js/localforage.min.js"></script>

Or install using a package manager of your choice:

npm install localforage --save

The library is browserify-friendly and can be used with bundlers like Webpack. The localForage interface doesn't require any additional initialization or loading so we can use it as soon as it becomes available.

import localforage from "localforage";

localforage.setItem('key', 'value');

**Writing to the Store**

Since we don't have to setup or create new databases, we can go right in and add some data to our store. This is done via the setItem method, taking two parameters - **key** and **value**.

* key - Unique, case-sensitive identifier that will be used whenever we want to access that item later on. Using setItem again on the same key will overwrite it.
* value - The data we want to store. It can be any valid string, number, object, array or file blob.

The process is asynchronous so if we want to do something else with the data and handle errors we have to use a promise or callback.

var hexColors = {

red: 'ff0000',

green: '00ff00',

yellow: 'ffff00'

};

localforage.setItem('colors', hexColors).then(function (value) {

console.log(value.red);

}).catch(function(err) {

console.error(err);

});

**Reading from the Store**

Fetching items from the database works in pretty much the same way. We simply use getItem, pass the name of the key, and use a promise to work with the data.

localforage.getItem('colors').then(function (value) {

console.log(value.red);

}).catch(function(err) {

console.error(err);

});

If we try to get a key that doesn't exist the promise will resolve successfully but the value inside will be null.

**Other Methods**

LocalForage has some other useful methods for working with the database. They are all just as easy to use as setItem and getItem, also supporting promises or callbacks.

* removeItem(key) - Removes the key/value pair from the store.
* keys() - Returns an array of all the keys' names (only the names).
* iterate(callback) - Works like forEach, expecting a callback function and going over all the key/value pairs.
* length() - Returns the number of items in the store.
* clear() - Wipes out the store.

**Multiple Databases**

So far the examples in this article used the localforage interface directly resulting in a single global store. If we need more then one store we can create as many instances as we want using createInstance:

var dogStore = localforage.createInstance({

name: "Dogs"

});

var catStore = localforage.createInstance({

name: "Cats"

});

Each store is completely independant and has access only to it's own data (NoSQL databases are mostly non-relational).

dogStore.setItem('Jake', 'Good boy');

catStore.getItem('Jake').then(function (value) {

console.log(value); // Will result in null

});

**Conclusion**

If you are looking for a simple way to manage client-side databases, localForage is one of the best tools available right now. It's API provides all the needed utilities, giving you enough freedom to organize your storage however you see fit.

* [The official docs](https://localforage.github.io/localForage/) - The project's documentation isn't very detailed but covers most of what you need to know.
* [localForage on GitHub](https://github.com/localForage/localForage) - The repo is very active, if you have any problems with the library make sure to check the issues here first.
* [angular-localForage](https://github.com/ocombe/angular-localForage) - Plugin for working with localForage in Angular.