

Storage means a space for putting things for future use. From a web application perspective, storage can be broadly classified into two types: server-side storage and client-side storage. Server-side storage points to the many different types of database storage like SQL, MYSQL, MongoDB, etc. Client-side storage means storing data in the browser itself so that it can be used when required without requesting the server-side database.

In this article, we'll discuss local storage and its usage from an Angular application's perspective. Let's start by trying to understand what local storage is.

## What is Local Storage?

Local storage is client-side storage for web applications. It stays there as long as it's not cleared—unlike session storage, which lasts for the current browser tab session. Local storage provides up to 5MB of storage, allocated separately for each domain. For example, local storage for the website www.abc.com will be different from www.xyz.com.

## Working with Local Storage

The local storage API provides a couple of methods to store, retrieve, remove and clear the storage. Let's see how to use these methods.

### localStorage.setItem

The setItem method allows you to add a value to the local storage. It stores data using a key value pair. For example:

localStorage.setItem('token','xhja787')

The above code stores the data xhja787 in a key called token inside the browser local storage.

***Note: localStorage only stores string values. To store objects or arrays, you need to use JSON.stringify to convert them into a string first.***

Now, let's see how to access the stored data using getItem

### localStorage.getItem

getItem allows you to read back the data saved in local storage using the same key. For example, to fetch the data stored using the key token, you can use the following method:

localStorage.getItem('token')

### localStorage.removeItem

Once you are done using the data in the local storage, you can remove it using the removeItem method. For example:

localStorage.removeItem('token')

### localStorage.clear

You can clear all of the data out of the local storage using the clear method to wipe the slate clean.

localStorage.clear()

Now that's all plain JavaScript code to access the local storage. Using local storage in Angular is no different. First, let’s create an Angular application and then we'll create a service wrapper on top of the localStorage API.

## Creating the Angular App

You can create the Angular app using the Angular CLI.

ng new local-app

Once you have the Angular app, you can create an Angular service using the following command:

ng g service local

Let's add four wrapper methods to add, get, remove and clear storage. Here is how the service looks:

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class LocalService {

constructor() { }

public saveData(key: string, value: string) {

localStorage.setItem(key, value);

}

public getData(key: string) {

return localStorage.getItem(key)

}

public removeData(key: string) {

localStorage.removeItem(key);

}

public clearData() {

localStorage.clear();

}

}

You can use the local service by injecting the service into your component. In this Angular example, to use the LocalService in the app.component.ts file, you need to inject it by importing it in AppComponent and creating an instance in the constructor method.

import { Component } from '@angular/core';

import { LocalService } from './local.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

constructor(private localStore: LocalService) {

}

}

## Local Storage Uses

Local storage can be used to store pretty much anything you like. It is most commonly used for storing JSON web tokens, user preferences and application settings such as themes.

## How Secure is Local Storage?

When using local storage in Angular, the rule of thumb is to not store anything sensitive. Anything you store in local storage can be accessed by code in your browser It's ok to store JSON web tokens since they’re already encrypted.

# Managing Local Storage in Angular

by

[Hudson Baker](https://blog.briebug.com/blog/author/Hudson%20Baker)

in [Angular](https://blog.briebug.com/blog/category/Angular) / [Server-Side Rendering](https://blog.briebug.com/blog/category/Server-Side-Rendering) / [Dependency Injection](https://blog.briebug.com/blog/category/Dependency-Injection) / [Frontend](https://blog.briebug.com/blog/category/Frontend)

Local Storage can be used in Angular in the same way as any other JavaScript application. However, in this article, I will show you how to leverage Angular and Observables to manage local storage using reactive patterns. I will cover the following topics:

* Defining Local Storage
* What is Safe to Put in Local Storage, and What is Not
* Common Use Cases
* Local Storage Methods
* Improving Architecture with Service as a Subject
* When there is No Local Storage Object (Server Side Rendering)

## Defining Local Storage

Local Storage is in-browser storage that has allocated space for each domain. The Local Storage for www.briebug.com will be separate from the Local Storage for www.homestarrunner.com. Up to **5MB** of data can be stored in Local Storage and all of the transactions with local storage are **synchronous**.

Data can be stored in the client side browser using HTML5 and is supported in all modern browsers. It is similar to session storage but unlike session storage it persists even if the browser is closed and reopened. If you want storage persistence only during the open tab session, use session storage.

Local storage can be cleared manually by the user as well as through the Javascript LocalStorage API or the Developer Tools Application tab.

You can find out more about Local Storage and it's compatibility at [Mozilla](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage).

## What is Safe to Put in Local Storage, and What is Not

You can store any kind of information you want in [Local Storage](https://blog.briebug.com/blog/how-to-add-ngrx-store-slices-into-localstorage) as long as it's formatted as JSON and is within the memory limits. This could be any data you want to cache to improve the user's experience.

**WARNING** Never put any sensitive data in local storage, ever! **All** browser storage (Local, Session, Cookies) is **insecure**. It can be accessed by any code in the browser. Even when using common authentication methods, like JWT's, make sure you are aware of the security risks and protect your users accordingly.

## Common Use Cases

What should you even use local storage for? This is the most important question when it comes to any utility. Good architecture relies on preparedness. Make sure you know a utilities capabilities, where it shines and where it doesn't shine, before using it.

### Storing Cookies

Cookies are normally stored as Cookies in the browser, not in local storage. Cookies must be generated by a web server and with SPA's (Single Page Applications) and PWA's (Progressive Web Apps) the web app may be running independent of a server. If you want to use cookies in that kind of app, you can create them and store them in Local Storage. Just encode them using a client-side library and store them in local storage. You could also roll your own but why re-invent the wheel?

When using Cookies in Local Storage, make sure you expire them! Using the browser's Cookie API, Cookies are automatically expired based on the set time. Local Storage does not expire like Cookies do.

Another advantage is that Cookie storage maxes out at 4 KB while Local Storage goes up to 5 MB. If you are worried about compatibility, you can check for Local Storage and default to Cookies if the browser does not support it.

### Storing JWTs

JSON Web Tokens, known as JWTs, are a common alternative to Cookies. They are a way to store user session data in Local Storage for API call authentications. This has become more popular than storing cookies in Local Storage because of many supporting libraries for the browser and server-side handling of JWTs.

JWTs are encrypted "tamper-proof" tokens. So, while they may be publicly accessible in local storage, their encrypted signature makes them safe to store. JWTs are verified on every request and must be reverified when the user's session expires.

### Storing Global Settings Info

In some cases, you may just want to store information on how the user likes their experience. This could mean user theming, custom navigation patterns aided by deep learning, or what types of things the user likes on your website.

### The Trade Offs of Other Types of Browser Storage

Which browser storage is right for you? While this article is not going to go into depth on comparing all of them, [wpreset](https://wpreset.com/localstorage-sessionstorage-cookies-detailed-comparison/) has a great chart with the basic differences of each type of storage.

### https://content.app-sources.com/s/799697976087936221/uploads/Images/Browser_Storage_Types-1348960.png

**Local Storage Methods**

Local Storage is an Immutable API, meaning every interaction with the object does not mutate it. This is a good strategy and best practice when dealing with data in most situations. Luckily, the API only has four methods. It is very simple.

The Local Storage API has 4 methods:

1. setItem(key: string, data: string | JSON): void

* Takes a key parameter and a value parameter. The key allows you to retrieve the value later using lookups. The value is stored as a JSON string. This method does not return anything.

2. getItem(key: string): string | JSON | null

* Takes a key parameter for looking up data in storage. If this lookup fails it will return a null value.

3. removeItem(key: string): undefined

* Takes a key parameter for looking up data in storage. This method always returns undefined.

4. clear(): undefined

* Takes no parameters. Clears all data in local storage. This method always returns undefined.

**Using These Methods**

A common way of setting data in local storage, getting it, and removing it:



setData(data) {

const jsonData = JSON.stringify(data)

localStorage.setItem('myData', jsonData)

}

​

getData() {

return localStorage.getItem('myData')

}

​

removeData(key) {

localStorage.removeItem(key)

}

*\*Even though localStorage exists on the Window object, localStorage is a global as well. We can all use these functions without accessing Window first*

**Improving Architecture with Service as a Subject**

Service as a Subject is a very common way to aid in data management and hydration by leveraging Observables. This architecture uses the principles of immutability as well as single responsibility. Because of that, we can create a much more stable, testable and usable feature.

**Creating an Injectable Local Storage Reference**

Creating a localStorage reference will be the first goal. This creates single responsibility for each feature and creates a reusable reference. This will also streamline the process when we need to substitute the API in the case of testing or server side rendering.

For this service, I will include a getter that returns a function containing the localStorage object:



function getLocalStorage(): Storage {

return localStorage;

}

​

@Injectable({ providedIn: "root" })

export class LocalStorageRefService {

get localStorage(): Storage {

return getLocalStorage();

}

}

*local-storage-ref.service.ts*

**Injecting the Local Storage Reference**

The next step is to inject our local-storage-ref.service.ts into our local-storage.service.



@Injectable({ providedIn: 'root' })

export class LocalStorageService {

private \_localStorage: Storage;

​

constructor(private \_localStorageRefService: LocalStorageRefService) {

this.\_localStorage = \_localStorageRefService.localStorage

}

}

*local-storage.service.ts*

**Behavior Subject and Exposing the Observable**

To lay the groundwork for data hydration, I'm going to create a Behavior Subject that uses the data type for the information I want to put in localStorage. The Behavior Subject will be private because we don't want random components altering the data flow: we want to ensure the data is changed only through the proper methods on the service. For reading the data, I will expose a public Observable.



interface MyData {

name: string

age: number

}

​

@Injectable({ providedIn: "root" })

export class LocalStorageService {

...

private \_myData$ = new BehaviorSubject<MyData>(null)

public myData$ = this.\_myData$.asObservable()

}

*exposing the data stream*

**Hydrating the Observable with Local Storage**

Now each method from the localStorage API can be used in tandem with our Behavior Subject to create a data stream from our Observable by utilizing the next() method:



@Injectable({ providedIn: "root" })

export class LocalStorageService {

...

setInfo(data: MyData) {

const jsonData = JSON.stringify(data)

this.\_localStorage.setItem('myData', jsonData)

this.\_myData$.next(data)

}

​

loadInfo() {

const data = JSON.parse(this.\_localStorage.getItem('myData'))

this.\_myData$.next(data)

}

​

clearInfo() {

this.\_localStorage.removeItem('myData')

this.\_myData$.next(null)

}

​

clearAllLocalStorage() {

this.\_localStorage.clear()

this.myData$.next(null)

}

}

*Local Storage and Hydration*

We can manage browser data in any component or other service. You could improve this service with any necessary side effects; like notifying the user that they have saved a new color theme.

**Using the Service and Showing the Data**

Now that all the boilerplate is set up, I can now use the exposed observable in the template to get the data as well as call these service functions to manage the data stream.

Starting with the Typescript file, I'm going to create a property as the service's data observable and add methods to call the service. I will include a form so new data can be added or removed from local storage by the user:



myInfo$ = this.\_localStorageService.myData$

form: FormGroup

​

constructor(

private \_localStorageService: LocalStorageService,

private \_fb: FormBuilder

) {}

​

ngOnInit() {

this.\_localStorageService.clearAllLocalStorage()

this.\_initForm()

}

​

private \_initForm() {

this.form = this.\_fb.group({

name: '',

age: ''

})

}

​

setInfo() {

const { name, age } = this.form.value

this.\_localStorageService.setInfo({

name,

age

})

}

​

clearInfo() {

this.\_localStorageService.clearInfo()

}

​

clearAll() {

this.\_localStorage.clearAllLocalStorage()

}

myInfo$ = this.\_localStorageService.myData%content%lt;/span> is key here

Using the Async Pipe I can stream the local storage data into the template and call these methods to modify the data.



<div \*ngIf="myInfo$ | async as myInfo">

​

name: {{ myInfo?.name || "no name set" }}

age: {{ myInfo?.age || "no age set" }}

​

</div>

<div>

<form \*ngIf="form" [formGroup]="form">

<input formControlName="name" type="text" placeHolder="name" />

<input formControlName="age" type="text" placeHolder="age" />

<button (click)="setInfo()" type="button">

set Info

</button>

​

<button (click)="clearInfo()" type="button">

clear Info

</button>

​

<button (click)="clearAll()" type="button">

clear all local storage

</button>

</form>

</div>

*easy and clean solution to stream local storage data*

This is a good architecture to help scale the use of local storage in your app. It can be easily reused and the functionality can be expanded from here.

**When there is No Local Storage Object (Server Side Rendering)**

With server-side rendering (like with Angular Universal) there is no localStorage object because there is no browser. In fact, there are no browser API's at all. It's all running on the server. There are several ways to solve this problem and hundreds of libraries that patch this as well. I'm just going to go over a couple options that will assume the use of Angular Universal which is generally run with a Node.js server.

**Defensive coding**

The first option is to just code defensively. Maybe you don't have an alternative to localStorage yet but you want to make sure your app is error-free in the event that it was rendered on a server.

In this case, we can use the platform token provided by angular which tells us if the app is running on a server or a browser. Let's re-visit the LocalStorageRefService but inject this token and check the browser:



import { Injectable, PLATFORM\_ID, Inject } from '@angular/core'

import { isPlatformBrowser } from '@angular/common'

​

function getLocalStorage() {

return localStorage

}

​

@Injectable({ providedIn: 'root' })

export class LocalStorageRefService {

​

constructor(@Inject(PLATFORM\_ID) private platformId) {}

​

get localStorage() {

if (isPlatformBrowser(this.platformId)) {

return getLocalStorage()

} else {

*// use alternative or throw error*

}

}

}

*check for the browser!*

This doesn't get us the full functionality of localStorage but it's the first step. Defensive coding is a good pattern for creating a positive user experience. An app with lots of errors can quickly drive users away.

**Injecting the Local Storage Reference**

You can simulate local storage???

Yes! Well, kind of. It's not technically localStorage at all because localStorage only exists on the browser. We can use an API that has the same functions and stores data in a Javascript Object. This way, the app does not know or care if it is using localStorage in a browser or on a server. It will function either way.

You can roll your own and, for a simple API like localStorage, it is actually not that difficult. You just need an object that has the same functions as localStorage and use that instead on the server. In this case, I'm going to use a library for the example. If you want to see a manually mocked example, check out [this article](https://medium.com/@santibarbat/how-to-use-localstorage-with-angular-9-universal-ssr-517578615637) by Santi Barbat on Medium. The localStorageRefService would be a good place to use his API mock and would look something like this:



constructor(

private \_simLocalStorageService: SimLocalStorageService,

@Inject(PLATFORM\_ID) private platformId

) {}

​

get localStorage() {

if (isPlatformBrowser(this.platformId)) {

return getLocalStorage()

} else {

return this.\_simLocalStorageService.getSimLocalStorage()

}

}

*return your own object of Local Storage*

That way you can mock localStorage in its own service, making sure to separate responsibility, and return it if the browser doesn't exist.

**Using a Library to Mock on the Server**

If you know your app is going to run server-side, you may opt to mock it out completely by using a server library and never use the localStorage object. I want my app to run on the server and the browser, though, so I'm still going to use this browser check and pass the library I want to use in place of local storage.

**Injecting the Local Storage Reference**

[Node Local Storage](https://www.npmjs.com/package/node-localstorage) is a great library that you can install in your angular app using npm install node-localstorage. This library specifically works with node, it will not work with other types of servers, but usually libraries exist for other types of servers as well.

After install, we can import this library and use it if the native localStorage API does not exist:



import { LocalStorage } from 'node-localstorage'

...

private localStorage: Storage

​

constructor(@Inject(PLATFORM\_ID) private platformId) {

if (isPlatformBrowser(this.platformId)) {

this.localStorage = getLocalStorage()

} else {

*// maintain only one instance of this file and API*

this.localStorage = LocalStorage('./scratch')

}

}

​

get localStorage() {

return this.localStorage

}

*use the storage library as an alternative*

The service has been altered to set a property with the designated localStorage API. This is done to make sure the instance of the API is not re-initialized during the use of the application. If this functionality was not in place, a new instance of the library's API would be created every time this function was called and the user would never be able to retrieve data because it would overwrite the previous instance. For the same reason, it is **very important** that you manage this service as a singleton; meaning there is only one instance of it.

'./scratch' is just a file name. This file will be created in the same directory as your service on the server. This is the file where the "local storage" data will be written.

This is a pretty good solution but we can improve this architecture as well.

**Dependency Injection**

Instead of having the consumer (the service) decide what API to use, we can use Angular's dependency Injector and move that decision to the provider. This is a better architecture because we are obscuring the responsibility of which API to use to a more relevant place in the app. The Service doesn't really care which API it uses for Local Storage and really it shouldn't be in charge of choosing the API. It's only there to retrieve the API for the app.

At the Module level we can use this function and provide it as a factory:



export function getLocalStorage(@Inject(PLATFORM\_ID) platformId) {

*// sim local storage service could also be the node storage library*

return isPlatformBrowser(platformId) ? new BrowserLocalStorageService(...) : new SimLocalStorageService(...);

}

​

{ provide: LocalStorageService, useFactory: getLocalStorage, deps: { PLATFORM\_ID } }

This is much simpler and contains the solution at the root of the module.

**Unit Testing**

Unit testing is often done on a headless server to be more efficient. This is the case when using Jest to test an app. Tests on headless servers do not have any Browser API's either so if the app isn't running server side, but headless unit testing is part of your CI, you will need to mock Local Storage for testing. This can also be written in a similar way as above by using Angular's dependency injector:



export interface LocalStorageService {

*// Describe standard local storage service interface with subject here*

*// setItem, getItem, etc...*

}

​

*// Standard provider for app that runs in a browser:*

*// { provide: LocalStorageService, useClass: BrowserLocalStorageService }*

​

*// Test provider when unit testing:*

TestBed.configureTestingModule({

providers: [

{ provide: LocalStorageService, useClass: SimLocalStorageService }

]

})

If you wanted to use a simulated service, and never actually differentiate if Local Storage exists on the browser, you can just inject the interface. Angular handles this for us so the interface is a usable code construct as well as the Injection Token:



@Injectable()

export class MyConsumerService {

constructor(private localStorage: LocalStorageService) {}

}

**That's a wrap**

Starting with a solid base, we learned about the bare-bones localStorage API. Its methods setItem getItem removeItem and clear were pretty simple and easy to implement. After grasping the basics, we moved on to a better architecture. By leveraging RxJs we created a service as a subject and hydrated our view with fresh data every time changes to localStorage were made. This became our clean and scalable solution up until we decided we wanted to run the app on the server. After running into the issue that localStorage just does not exist on the server, we figured out how to create an alternative by either rolling our own API or utilizing a server-side compatible library.

This skill went from "zero to hero" and I hope it helped you broaden your understanding of both Angular and Browsers by providing context to real-world issues. If you want to play with the example built here you can check it out [here](https://stackblitz.com/edit/local-storage-subject-as-service-j9qe28?file=src%2Fapp%2Flocal-storage.service.ts) on stackblitz.com.