remoteStorage.js Documentation

Release 1.2.3

RS Community

Sep 23, 2020

CONTENTS

Welcome to the remoteStorage.js documentation!

remoteStorage.js is a JavaScript library for storing user data locally in the browser, as well as connecting to remoteStorage servers and syncing data across devices and applications. It is also capable of connecting and syncing data with a person's Dropbox or Google Drive account (optional).

Note: For brevity's sake, we will also use the short name rs.js across these docs.

CHAPTER

ONE

WHY USE THIS?

1.1 Offline-first design

rs.js is offline-first by design, meaning data is stored locally first[1], and synced to and from a remote storage account second. This makes it a robust sync solution for mobile applications, where slow and spotty network connections are a normal situation.

It's also useful, when a backend goes down, as users can just keep using their app and have their data automatically synced whenever the server is back online.

1.2 Zero backend

rs.js is built for creating fully unhosted apps. Meaning users are able to connect their own storage account to apps on their devices, without app developers having to store or even see their users' data. Thus, developers don't have to integrate, manage, maintain and secure a storage server or cloud.

A nice side effect of this design is that your app can scale to millions of users with literally *zero* cost for storage.

Also, in case you decide to abandon your app[2], users can continue to use it across devices until they switch to a new one on their own time. You may even reverse your decision at some point and still have a lot of your users right there.

1.3 Data sharing

Different apps can access the same data, so you can build an app that uses and manipulates existing data, without building import/export features or having users start over from scratch.

Even better, you can get advanced capabilities for free by using shared, open-source *data modules*, which you can cooperate on with other developers.

For example: if you want to enable users to share files from their storage account in your app, you can just integrate the shares module within a matter of minutes, giving you client-side thumbnail generation and other features in the process.

[1] Except for apps and use cases that don't require caching, like e.g. with Sharesome
[2] Let's just be honest: nothing lasts forever.

1.4 Reliability

The very first prototype of rs.js has been written in November 2010. Since then, it has been used, tested, stabilized, and improved in more than 4000 commits. The library has been used in commercial apps by hundreds of thousands of users, and in countries around the globe. We have seen pretty much every device, browser, privacy setting and network connection there is, and fixed bugs and issues for most of them.

In short: you can rely on rs.js to do its job. And if you do find a critical bug, there's a team of people who will help with fixing it.

1.5 One JS API for multiple storage options

rs.js optionally supports Dropbox and Google Drive as storage backends which users can connect. Conveniently, as an app developer you don't have to implement anything special in order for these backends to work with your code[3]. Just *configure OAuth app keys*, and your users can choose between 3 different backends to connect.

[3] Except adding UI for it, in case you're not using the *connect widget*, of course.

CHAPTER TWO

GETTING STARTED

This section contains introductory documentation for app developers who are new to *remoteStorage.js*.

2.1 Adding rs.js to an app

remoteStorage.js is distributed as a single UMD (Universal Module Definition) build, which means it should work with all known JavaScript module systems, as well as without one (using a global variable).

We recommend adding the library from a JavaScript package manager, although you may also just download the release build from GitHub.

The package is available on npm as `remotestoragejs` and on Bower as `remotestorage`:

```
$ npm install remotestoragejs
```

```
$ bower install -S rs
```

2.1.1 Examples

ES6 module

```
import RemoteStorage from 'remotestoragejs';
```

CommonJS module

```
var RemoteStorage = require('remotestoragejs');
```

AMD module

For example with RequireJS:

```
requirejs.config({
  paths: {
    RemoteStorage: './lib/remotestorage'
  }
});
```

(continues on next page)

(continued from previous page)

```
requirejs(['RemoteStorage'], function(RemoteStorage) {
  // Here goes my app
});
```

No module system

If you just link the build from HTML, it will add `RemoteStorage` as a global variable to `window`.

```
<script type="text/javascript" src="remotestorage.js"></script>
```

Ember.js

ES6 modules from npm should be supported natively soon, but for now you can use Browserify via ember-browserify, enabling you to import the module from npm like this:

```
import RemoteStorage from 'npm:remotestoragejs';
```

2.1.2 Caveat emptor (no promises)

Please be aware of the fact that although remoteStorage.js is generally compatible with older browsers as well as the latest ones, we do not include a polyfill for JavaScript Promises anymore.

This means that, if you do not add your own polyfill, and no other library in your build comes with one, rs.js will break in browsers, which do not support Promises. A detailed overview of supported browsers is available on caniuse.com. Notable examples would be Android up to 4.4 and Internet Explorer up to 11.

You can find a list of polyfill libraries on the Promises website. A good choice for a small and simple polyfill would be es6-promise-auto for example.

2.2 Initialization & configuration

Now that you've imported the `RemoteStorage` class, here's how you typically set things up.

Note: Where and how you do this exactly will naturally depend on the rest of your code, your JS framework, and personal preferences.

2.2.1 Initializing an instance

First step is to initialize a `remoteStorage` instance:

```
const remoteStorage = new RemoteStorage();
```

The constructor optionally takes a configuration object. Let's say we want to enable debug logging to see in the console what rs.js is doing behind the scenes:

```
const remoteStorage = new RemoteStorage({logging: true});
```

Or perhaps we're building an app that doesn't need local caching, but only operates on the remote server/account:

```
const remoteStorage = new RemoteStorage({cache: false});
```

Also see the *RemoteStorage API doc*.

2.2.2 Claiming access

Next, we need to tell *rs.js* which parts of the user's storage we want to access. Let's say we want to read and write a user's favorite drinks, which they might have added via the My Favorite Drinks demo app:

```
remoteStorage.access.claim('myfavoritedrinks', 'rw');
```

Now, when they connect their storage, users will be asked to give the app read/write access to the `myfavoritedrinks/` folder. And that's also what the OAuth token, which we receive from their storage server, will be valid for, of course.

If you want to build a special app, like for example a backup utility, or a data browser, you can also claim access to the entire storage (which is generally discouraged):

```
remoteStorage.access.claim('*', 'rw');
```

Also see the *Access API doc*.

2.2.3 Configuring caching

Last but not least, we'll usually want to configure caching (and with it automatic sync) for the data we're accessing. The `caching.enable()` method will activate full caching for the given path, meaning all of the items therein will be automatically synced from and to the server:

```
remoteStorage.caching.enable('/myfavoritedrinks/')
```

See the *Caching API doc* for details and options.

2.3 Using the Connect Widget add-on

The easiest option for letting people connect their storage to your app is using the Connect Widget add-on library, which is written and maintained by the rs.js core team.

This is by no means required, and it's easy to integrate all functionality in your own UI as well. However, it's a great way for starting out with RS app development and, if desired, replacing the widget with your own custom code later on.

Hint: If you haven't seen the widget in action yet, you can try it out e.g. with My Favorite Drinks right now.

2.3.1 Adding the library

The Connect Widget library is distributed the same way as *remoteStorage.js* itself: as a UMD build, compatible with all JavaScript module systems, or as a global variable named `Widget`, when linked directly.

You can find the connect widget as `remotestorage-widget` on npm, and its source code and usage instructions on GitHub.

Check out *Adding rs.js to an app* for examples of loading a UMD module in your code.

2.3.2 Adding the widget

With the `Widget` class loaded, just create a new widget instance using the *previously initialized* `remoteStorage` instance, like so:

```
const widget = new Widget(remoteStorage);
```

Then you can attach the widget to the DOM:

```
widget.attach();
```

Or if you want to attach it to a specific parent element, you can also hand it a DOM element ID:

```
widget.attach('my-parent-element-id');
```

That's it! Now your users can use the widget in order to connect their storage, and you can listen to the `remoteStorage` instance's events in order to get informed about connection status, sync progress, errors, and so on.

Tip: If you'd like to implement connect functionality in your own user interface and code, the widget can serve as a useful source code example. For everything it does, the Connect Widget only uses public APIs and events of *rs.js*, which you can also use in your own code.

2.4 Handling events

In order to get informed about users connecting their storage, data being transferred, the library going into offline mode, errors being thrown, and other such things, you can listen to the events emitted by the `RemoteStorage` instance, as well as `BaseClient` instances.

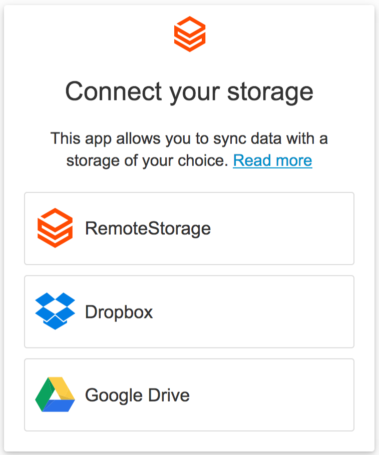
Simply register your event handler functions using the `.on()` method, like so:

```
remoteStorage.on('connected', () => {
  const userAddress = remoteStorage.remote.userAddress;
  console.debug(`${userAddress} connected their remote storage.`);
})

remoteStorage.on('network-offline', () => {
  console.debug(`We're offline now.`);
})

remoteStorage.on('network-online', () => {
  console.debug(`Hooray, we're back online.`);
})
```

Check out the *RemoteStorage API doc* for a complete list of events and when exacty they are emitted.

Also check out *Change events* in the *BaseClient API doc*, which you can use to handle incoming data and changes from the remote storage.

2.5 Offering Dropbox and Google Drive storage options


Connect your storage
This app allows you to sync data with a storage of your choice. Read more
RemoteStorage
Dropbox
Google Drive

rs.js has optional support for syncing data with Dropbox and Google Drive instead of a RemoteStorage server.

There are a few drawbacks, mostly sync performance and the lack of a permission model. So apps can usually access all of a user's storage with these backends (vs. only relevant parts of the storage with RS accounts). However, while RS is not a widely known and deployed protocol, we find it helpful to let users choose something they already know, and potentially migrate to an RS account later on.

For these additional backends to work, you will have to register your app with Dropbox and/or Google first. Then you can configure your OAuth app ID/key like so:

```
remoteStorage.setApiKeys({
  dropbox: 'your-app-key',
  googledrive: 'your-client-id'
});
```

Hint: The *Connect widget* will automatically show only the available storage options, based on the presence of the *dropbox* and *googledrive* API keys. RemoteStorage is always enabled.

2.5.1 Dropbox

An app key can be obtained by registering your app.

- Create a new app for the "Dropbox API", with "Full Dropbox access"
- You need to set one or more OAuth2 redirect URIs for all routes a user can connect from, for example `http://localhost:8000/` for an app you are developing locally.

Known issues

- Storing files larger than 150MB is not yet supported
- Listing and deleting folders with more than 10000 files will cause problems
- Content-Type is not fully supported due to limitations of the Dropbox API
- Dropbox preserves cases but is not case-sensitive
- `getItemURL` is not implemented yet (see issue #1052)

2.5.2 Google Drive

A client ID can be obtained by registering your app in the Google Developers Console.

- Create an API, then add credentials for Google Drive API. Specify you will be calling the API from a "Web browser (Javascript)" project. Select that you want to access "User data".
- On the next screen, fill out the Authorized JavaScript origins and Authorized redirect URIs for your app (for every route a user can connect from, same as with Dropbox)
- Once your app is running in production, you will want to get verified by Google to avoid a security warning when the user first connects their account

Known issues

- Sharing public files is not supported yet (see issue #1051)
- `getItemURL` is not implemented yet (see issue #1054)

2.6 Reading and writing data

As soon as your *RemoteStorage* instance is ready for action (signaled by the `ready` event), we can start reading and writing data.

2.6.1 Anonymous mode

One of the unique features of rs.js is that users are not required to have their storage connected, before using your app[1]. Any data which is written to the local store before connecting an account, is automatically synced to the remote storage, whenever one is being connected[2].

[1] You may intentionally require them to connect their storage first, if it improves the UX, of course.
[2] We call the state before connecting a storage "anonymous mode".

2.6.2 Using BaseClient

A `BaseClient` instance is the main endpoint you will use for interacting with a connected storage: listing, reading, creating, updating and deleting documents, as well as handling incoming changes.

Check out the *BaseClient API docs* in order to learn about all functions available for reading and writing data and how to use them.

There are two options for acquiring a BaseClient instance:

Quick and dirty: creating a client via `scope()`

This should mainly be used for manually exploring client functions and in development. Using *`scope()`*, you can create a new BaseClient scoped to a given path:

```
const client = remoteStorage.scope('/foo/');

// List all items in the "foo/" category/folder
client.getListing('')
  .then(listing => console.log(listing));

// Write some text to "foo/bar.txt"
const content = 'The most simple things can bring the most happiness.'
client.storeFile('text/plain', 'bar.txt', content)
  .then(() => console.log("data has been saved"));
```

The recommended way: using clients in data modules

The recommended way is to use the private and public `BaseClient` instances, which are available in so-called *data modules*. Continue to the next section in order to learn about them.

CHAPTER

THREE

DATA MODULES

One of the core ideas of the remoteStorage protocol, and one of its unique features, is that data shouldn't be locked into a single app. Why not create a to-do in one app, but track time on it in another one for example?

With traditional Web apps, this is only possible by implementing custom, proprietary APIs, which make you entirely dependent on the app provider. Also, for other app developers to access that data via the API, they usually have to register an application with the original provider, who can revoke this access at any point. However, with remoteStorage, and unhosted web apps in general, end users have ultimate control over which apps have access to their data.

In order to make it easy and safe for your app data to be compatible with other apps, we created the concept of data modules for rs.js, which can be shared and collaboratively developed in the open.

Data modules can contain as much or little functionality as necessary or desired, not just defining data formats and types, but also handling for example data validation, formatting, processing, transformation, encryption, indexing, and whatever else you may come up with.

Thus, modules are not only useful for making your data/app compatible with others, but also to encapsulate functionality you might only want to use in a custom module in your own app. And although sharing your data module(s) is certainly encouraged, it is by no means required, of course.

3.1 Defining a module

A data module is just a JavaScript object containing a module name and a builder function.

The builder function receives two *base clients* when loaded: one for private data stored in `/my-module-name/` and one for public data stored in `/public/my-module-name/`. It must return an object, defining the properties and functions to be used in the app as `exports`:

```
var Bookmarks = { name: 'bookmarks', builder: function(privateClient, publicClient) {
  return {
    exports: {
     addBookmark: function() {}
    }
  }
}};
```

You can then load it into your *RemoteStorage* instance either on initialization, or later using the `addModule()` function:

```
const remoteStorage = new RemoteStorage({ modules: [ Bookmarks ] });

// or later:

remoteStorage.addModule(Bookmarks);
```

It will then be available on the instance as its module name, allowing you to call the functions and properties that the module exports:

```
remoteStorage.bookmarks.addBookmark();
```

3.2 Defining data types

Data types can be defined using the `declareType()` method. It expects a name (which you can later use with `storeObject()`), as well as a JSON Schema object defining the actual structure and formatting of your data.

Consider this simplified example of an archive bookmark:

```
var Bookmarks = { name: 'bookmarks', builder: function(privateClient, publicClient) {

  privateClient.declareType('archive-bookmark', {
    "type": "object",
    "properties": {
      "id": {
        "type": "string"
      },
      "title": {
        "type": "string"
      },
      "url": {
        "type": "string",
        "format": "uri"
      },
      "tags": {
        "type": "array",
        "default": []
      },
    },
    "required": [ "title", "url" ]
  });

  // ...
}};
```

Now that we have a basic data type in place for storing bookmarks, we can add a function for storing them. This will actually validate the incoming data against the type's schema, and reject the promise with detailed validation errors in case the data format doesn't match:

```
var Bookmarks = { name: 'bookmarks', builder: function(privateClient, publicClient) {
  // ...

  return {
    exports: {

      add: function (bookmark) {
        bookmark.id = md5Hash(bookmark.url); // hash URL for nice ID
        var path = "archive/" + bookmark.id; // use hashed URL as filename as well

        return privateClient.storeObject("archive-bookmark", path, bookmark).
          then(function() {
            return bookmark; // return bookmark with added ID property
```

(continues on next page)

(continued from previous page)

```
          });
        }

      }
    }
}};

// and in your app:

remoteStorage.bookmarks.add({
  title: 'Unhosted Web Apps',
  url: 'https://unhosted.org',
  tags: ['unhosted', 'remotestorage', 'offline-first']
})
.then(() => {
  console.log('stored bookmark successfully');
})
.catch((err) => {
  console.error('validation error:', err);
});
```

Hint: JSON Schema is very powerful and flexible. If you want to learn more about it, check out the free e-book Understanding JSON Schema for example. The complete official specs can be found at http://json-schema.org/documentation.html

3.3 Publishing and finding data modules

3.3.1 npm

The recommended way for publishing data modules is as npm packages.

Our naming convention for rs.js modules is `remotestorage-module-mymodulename`. Thus, you can also find them by searching npm for "remotestorage-module".

You can also add "remotestorage-module" and "remotestorage" to the `keywords` property of your `package.json`.

3.3.2 GitHub & Co.

If you use GitHub – or any other code hosting/collaboration platform for that matter – for publishing your module's source code, please use the same naming convention as for the npm module for the repo name. And it's a good idea to add the topic/tag/label "remotestorage-module" there as well, of course.

https://github.com/topics/remotestorage-module

Hint: With npm, you can also install modules directly from a Git repo or GitHub, pointing to just the repo or a branch name, tag, or commit: https://docs.npmjs.com/files/package.json#github-urls

3.3.3 Examples

- For a real-world example of a data module package, see e.g. the shares module on GitHub and on npm. Check out `webpack.config.js` and the source code in `index.js` to see how it is built and exported.

Note: Unfortunately, we didn't have any package management for data modules before rs.js 1.0. To be fair, JavaScript package managers weren't actually a thing yet, when this functionality was added to the library. However, it means we're still in the process of porting and publishing modules and you won't find very many existing data modules on npm right now. You can check the old modules repo for source code of legacy modules.

CHAPTER

FOUR

JAVASCRIPT API

This section contains documentation of the public JavaScript functions available to developers using remoteStorage.js in their apps and programs.

4.1 RemoteStorage

4.1.1 Constructor

Create a `remoteStorage` instance like so:

```
var remoteStorage = new RemoteStorage();
```

The constructor can optionally be called with a configuration object. This example shows all default values:

```
var remoteStorage = new RemoteStorage({
  cache: true,
  changeEvents: {
    local:    true,
    window:   false,
    remote:   true,
    conflict: true
  },
  cordovaRedirectUri: undefined,
  logging: false,
  modules: []
});
```

Note: In the current version, it is only possible to use a single `RemoteStorage` instance. You cannot connect to two different remotes yet. We intend to support this soon (see issue #991)

Warning: For the change events configuration, you currently have to set all events explicitly. Otherwise it would disable the unspecified ones. (see issue #1025)

4.1.2 Events

You can handle events from your `remoteStorage` instance by using the `.on()` function. For example:

```
remoteStorage.on('connected', function() {
  // Storage account has been connected, let's roll!
});
```

List of events

`ready`

Emitted when all features are loaded and the RS instance is ready

`not-connected`

Emitted when ready, but no storage connected ("anonymous mode")

`connected`

Emitted when a remote storage has been connected

`disconnected`

Emitted after disconnect

`error`

Emitted when an error occurs; receives an error object as argument

There are a handful of known errors, which are identified by the `name` property of the error object:

Name	Description
`Unauthorized`	Emitted when a network request resulted in a 401 or 403 response. You can use this event to handle invalid OAuth tokens in custom UI (i.e. when a stored token has been revoked or expired by the RS server).
`DiscoveryError`	A variety of storage discovery errors, e.g. from user address input validation, or user address lookup issues

Example:

```
remoteStorage.on('error', err => console.log(err));

// {
//   name: "Unauthorized",
//   message: "App authorization expired or revoked.",
//   stack: "Error  at new a.Unauthorized (vendor.js:65710:41870)"
// }
```

`features-loaded`

Emitted when all features are loaded

`connecting`

Emitted before webfinger lookup

`authing`

Emitted before redirecting to the authing server

`wire-busy`

Emitted when a network request starts

`wire-done`

Emitted when a network request completes

`sync-req-done`

Emitted when a single sync request has finished

`sync-done`

Emitted when all tasks of a sync have been completed and a new sync is scheduled

`network-offline`

Emitted once when a wire request fails for the first time, and `remote.online` is set to false

`network-online`

Emitted once when a wire request succeeds for the first time after a failed one, and `remote.online` is set back to true

sync-interval-change

Emitted when the sync interval changes

4.1.3 Prototype functions

The following functions can be called on your `remoteStorage` instance:

authorize (*options*)

Initiate the OAuth authorization flow.

This function is called by custom storage backend implementations (e.g. Dropbox or Google Drive).

Arguments

- **options** (*object*) –
- **options.authURL** (*string*) – URL of the authorization endpoint
- **options.scope** (*string*) – access scope
- **options.clientId** (*string*) – client identifier (defaults to the origin of the redirectUri)

connect (*userAddress*[, *token*])

Connect to a remoteStorage server.

Discovers the WebFinger profile of the given user address and initiates the OAuth dance.

This method must be called *after* all required access has been claimed. When using the connect widget, it will call this method itself.

Special cases:

1. If a bearer token is supplied as second argument, the OAuth dance will be skipped and the supplied token be used instead. This is useful outside of browser environments, where the token has been acquired in a different way.
2. If the Webfinger profile for the given user address doesn't contain an auth URL, the library will assume that client and server have established authorization among themselves, which will omit bearer tokens in all requests later on. This is useful for example when using Kerberos and similar protocols.

Arguments

- **userAddress** (*string*) – The user address (user@host) to connect to.
- **token** (*string*) – (optional) A bearer token acquired beforehand

Example:

```
remoteStorage.connect('user@example.com');
```

disconnect ()

"Disconnect" from remote server to terminate current session.

This method clears all stored settings and deletes the entire local cache.

Example:

```
remoteStorage.disconnect();
```

enableLog()

TODO: do we still need this, now that we always instantiate the prototype?

Enable remoteStorage logging.

Example:

```
remoteStorage.enableLog();
```

disableLog()

TODO: do we still need this, now that we always instantiate the prototype?

Disable remoteStorage logging

Example:

```
remoteStorage.disableLog();
```

getSyncInterval()

Get the value of the sync interval when application is in the foreground

Returns **number** – A number of milliseconds

Example:

```
remoteStorage.getSyncInterval();
// 10000
```

setSyncInterval(*interval*)

Set the value of the sync interval when application is in the foreground

Arguments

- **interval** (*number*) – Sync interval in milliseconds (between 1000 and 3600000)

Example:

```
remoteStorage.setSyncInterval(10000);
```

getBackgroundSyncInterval()

Get the value of the sync interval when application is in the background

Returns **number** – A number of milliseconds

Example:

```
remoteStorage.getBackgroundSyncInterval();
// 60000
```

setBackgroundSyncInterval(*interval*)

Set the value of the sync interval when the application is in the background

Arguments

- **interval** – Sync interval in milliseconds (between 1000 and 3600000)

Example:

```
remoteStorage.setBackgroundSyncInterval(60000);
```

getCurrentSyncInterval()

Get the value of the current sync interval. Can be background or foreground, custom or default.

Returns **number** – A number of milliseconds

Example:

```
remoteStorage.getCurrentSyncInterval();
// 15000
```

getRequestTimeout()

Get the value of the current network request timeout

Returns **number** – A number of milliseconds

Example:

```
remoteStorage.getRequestTimeout();
// 30000
```

setRequestTimeout(*timeout*)

Set the timeout for network requests.

Arguments

- **timeout** – Timeout in milliseconds

Example:

```
remoteStorage.setRequestTimeout(30000);
```

scope(*path*)

This method enables you to quickly instantiate a BaseClient, which you can use to directly read and manipulate data in the connected storage account.

Please use this method only for debugging and development, and choose or create a *data module* for your app to use.

Arguments

- **path** (*string*) – The base directory of the BaseClient that will be returned (with a leading and a trailing slash)

Returns **BaseClient** – A client with the specified scope (category/base directory)

Example:

```
remoteStorage.scope('/pictures/').getListing('');
remoteStorage.scope('/public/pictures/').getListing('');
```

setApiKeys(*apiKeys*)

Set the OAuth key/ID for either GoogleDrive or Dropbox backend support.

Arguments

- **apiKeys** (*Object*) – A config object with these properties:
- **apiKeys.type** (*string*) – Backend type: 'googledrive' or 'dropbox'
- **apiKeys.key** (*string*) – Client ID for GoogleDrive, or app key for Dropbox

Example:

```
remoteStorage.setApiKeys({
  dropbox: 'your-app-key',
  googledrive: 'your-client-id'
});
```

setCordovaRedirectUri (*uri*)

Set redirect URI to be used for the OAuth redirect within the in-app-browser window in Cordova apps.

Arguments

- **uri** (*string*) – A valid HTTP(S) URI

Example:

```
remoteStorage.setCordovaRedirectUri('https://app.wow-much-app.com');
```

startSync ()

Start synchronization with remote storage, downloading and uploading any changes within the cached paths.

Please consider: local changes will attempt sync immediately, and remote changes should also be synced timely when using library defaults. So this is mostly useful for letting users sync manually, when pressing a sync button for example. This might feel safer to them sometimes, esp. when shifting between offline and online a lot.

Returns **Promise** – A Promise which resolves when the sync has finished

Example:

```
remoteStorage.startSync();
```

stopSync ()

Stop the periodic synchronization.

Example:

```
remoteStorage.stopSync();
```

on (*eventName*, *handler*)

Register an event handler. See *List of events* for available event names.

Arguments

- **eventName** (*string*) – Name of the event
- **handler** (*function*) – Event handler

Example:

```
remoteStorage.on('connected', function() {
  console.log('user connected their storage');
});
```

onChange (*path*, *handler*)

Add a "change" event handler to the given path. Whenever a "change" happens (as determined by the backend, such as e.g. <RemoteStorage.IndexedDB>) and the affected path is equal to or below the given 'path', the given handler is called.

You should usually not use this method directly, but instead use the "change" events provided by *BaseClient*

Arguments

- **path** (*string*) – Absolute path to attach handler to
- **handler** (*function*) – Handler function

Example:

```
remoteStorage.onChange('/bookmarks/', function() {
  // your code here
})
```

4.2 BaseClient

A `BaseClient` instance is the main endpoint you will use for interacting with a connected storage: listing, reading, creating, updating and deleting documents, as well as handling incoming changes.

Base clients are usually used in *data modules*, which are loaded with two `BaseClient` instances by default: one for private and one for public documents.

However, you can also instantiate a BaseClient outside of a data module using the `remoteStorage.scope()` function. Similarly, you can create a new scoped client within another client, using the `BaseClient`'s own `scope()`.

Contents

4.2.1 Data read/write operations

A `BaseClient` deals with three types of data: folders, objects and files:

- `getListing()` returns a mapping of all items within a folder.
- `getObject()` and `storeObject()` operate on JSON objects. Each object has a type.
- `getFile()` and `storeFile()` operates on files. Each file has a MIME type.
- `getAll()` returns all objects or files for the given folder path.
- `remove()` operates on either objects or files (but not folders; folders are created and removed implictly).

Caching logic for read operations

All functions requesting/reading data will immediately return data from the local store, *as long as it is reasonably up-to-date*. The default maximum age of requested data is two times the periodic sync interval (10 seconds by default).

However, you can adjust this behavior by using the `maxAge` argument with any of these functions, thereby changing the maximum age or removing the requirement entirely.

- If the `maxAge` requirement is set, and the last sync request for the path is further in the past than the maximum age given, the folder will first be checked for changes on the remote, and then the promise will be fulfilled with the up-to-date document or listing.
- If the `maxAge` requirement is set, and cannot be met because of network problems, the promise will be rejected.
- If the `maxAge` requirement is set to `false`, or the library is in offline mode, or no remote storage is connected (a.k.a. "anonymous mode"), the promise will always be fulfilled with data from the local store.

Hint: If *caching* for the folder is turned off, none of this applies and data will always be requested from the remote store directly.

List of functions

`getListing`(*path*, *maxAge*)

Get a list of child nodes below a given path.

Arguments

- **`path`** (*`string`*) – The path to query. It MUST end with a forward slash.
- **`maxAge`** (*`number`*) – (optional) Either `false` or the maximum age of cached listing in milliseconds. See *Caching logic for read operations*.

Returns **Promise** – A promise for an object representing child nodes

Example usage:

```
client.getListing('')
      .then(listing => console.log(listing));
```

The folder listing is returned as a JSON object, with the root keys representing the pathnames of child nodes. Keys ending in a forward slash represent *folder nodes* (subdirectories), while all other keys represent *data nodes* (files/objects).

Data node information contains the item's ETag, content type and -length.

Example of a listing object:

```
{
  "@context": "http://remotestorage.io/spec/folder-description",
  "items": {
    "thumbnails/": true,
    "screenshot-20170902-1913.png": {
      "ETag": "6749fcb9eef3f9e46bb537ed020aeece",
      "Content-Length": 53698,
      "Content-Type": "image/png;charset=binary"
    },
    "screenshot-20170823-0142.png": {
      "ETag": "92ab84792ef3f9e46bb537edac9bc3a1",
```

(continues on next page)

(continued from previous page)

```
      "Content-Length": 412401,
      "Content-Type": "image/png;charset=binary"
    }
  }
}
```

Warning: At the moment, this function only returns detailed metadata, when caching is turned off (`new RemoteStorage({cache: false})`). With caching turned on, it will only contain the item names as properties with `true` as value. See issues #721 and #1108 — contributions welcome!

getObject (*path*, *maxAge*)

Get a JSON object from the given path.

Arguments

- **path** (*string*) – Relative path from the module root (without leading slash).
- **maxAge** (*number*) – (optional) Either `false` or the maximum age of cached object in milliseconds. See *Caching logic for read operations*.

Returns Promise – A promise, which resolves with the requested object (or `null` if non-existent)

Example:

```
client.getObject('/path/to/object')
      .then(obj => console.log(obj));
```

storeObject (*typeAlias*, *path*, *object*)

Store object at given path. Triggers synchronization.

See `declareType()` and *data types* for an explanation of types

Arguments

- **type** (*string*) – Unique type of this object within this module.
- **path** (*string*) – Path relative to the module root.
- **object** (*object*) – A JavaScript object to be stored at the given path. Must be serializable as JSON.

Returns Promise – Resolves with revision on success. Rejects with a ValidationError, if validations fail.

Example:

```
var bookmark = {
  url: 'http://unhosted.org',
  description: 'Unhosted Adventures',
  tags: ['unhosted', 'remotestorage', 'no-backend']
}
var path = MD5Hash(bookmark.url);

client.storeObject('bookmark', path, bookmark)
      .then(() => console.log('bookmark saved'))
      .catch((err) => console.log(err));
```

getAll (*path*, *maxAge*)

Get all objects directly below a given path.

Arguments

- **path** (*string*) – Path to the folder. Must end in a forward slash.
- **maxAge** (*number*) – (optional) Either `false` or the maximum age of cached objects in milliseconds. See *Caching logic for read operations*.

Returns **Promise** – A promise for an object

Example response object:

```
// TODO
```

For items that are not JSON-stringified objects (e.g. stored using *storeFile* instead of *storeObject*), the object's value is filled in with *true*.

Example usage:

```
client.getAll('').then(objects => {
  for (var path in objects) {
    console.log(path, objects[path]);
  }
});
```

getFile (*path*, *maxAge*)
Get the file at the given path. A file is raw data, as opposed to a JSON object (use `getObject()` for that).

Arguments

- **path** (*string*) – Relative path from the module root (without leading slash).
- **maxAge** (*number*) – (optional) Either `false` or the maximum age of the cached file in milliseconds. See *Caching logic for read operations*.

Returns **Promise** – A promise for an object

The response object contains two properties:

`mimeType`	String representing the MIME Type of the document.
`data`	Raw data of the document (either a string or an ArrayBuffer)

Example usage (displaying an image):

```
client.getFile('path/to/some/image').then(file => {
  var blob = new Blob([file.data], { type: file.mimeType });
  var targetElement = document.findElementById('my-image-element');
  targetElement.src = window.URL.createObjectURL(blob);
});
```

storeFile (*mimeType*, *path*, *body*)
Store raw data at a given path.

Arguments

- **mimeType** (*string*) – MIME media type of the data being stored
- **path** (*string*) – Path relative to the module root
- **body** (*string|ArrayBuffer|ArrayBufferView*) – Raw data to store

Returns **Promise** – A promise for the created/updated revision (ETag)

Example (UTF-8 data):

```
client.storeFile('text/html', 'index.html', '<h1>Hello World!</h1>')
      .then(() => { console.log("Upload done") });
```

Example (Binary data):

```
var input = document.querySelector('form#upload input[type=file]');
var file = input.files[0];
var fileReader = new FileReader();

fileReader.onload = function () {
  client.storeFile(file.type, file.name, fileReader.result)
        .then(() => { console.log("Upload done") });
};

fileReader.readAsArrayBuffer(file);
```

remove (*path*)

Remove node at given path from storage. Triggers synchronization.

Arguments

- **path** (*string*) – Path relative to the module root.

Returns Promise –

Example:

```
client.remove('path/to/object')
      .then(() => console.log('item successfully deleted'));
```

4.2.2 Change events

`BaseClient` offers a single event, named `change`, which you can add a handler for using the `.on()` function (same as in `RemoteStorage`):

```
client.on('change', function (evt) {
  console.log('data was added, updated, or removed:', evt)
});
```

Using this event, you can stay informed about data changes, both remote (from other devices or browsers), as well as locally (e.g. other browser tabs).

In order to determine where a change originated from, look at the `origin` property of the incoming event. Possible values are `window`, `local`, `remote`, and `conflict`, explained in detail below.

Example:

```
{
  // Absolute path of the changed node, from the storage root
  path: path,
  // Path of the changed node, relative to this baseclient's scope root
  relativePath: relativePath,
  // See origin descriptions below
  origin: 'window|local|remote|conflict',
  // Old body of the changed node (local version in conflicts; undefined if creation)
  oldValue: oldBody,
  // New body of the changed node (remote version in conflicts; undefined if deletion)
```

(continues on next page)

(continued from previous page)

```
  newValue: newBody,
  // Old contentType of the changed node (local version for conflicts; undefined if␣
↪creation)
  oldContentType: oldContentType,
  // New contentType of the changed node (remote version for conflicts; undefined if␣
↪deletion)
  newContentType: newContentType
}
```

local

Events with origin `local` are fired conveniently during the page load, so that you can fill your views when the page loads.

Example:

```
{
  path: '/public/design/color.txt',
  relativePath: 'color.txt',
  origin: 'local',
  oldValue: undefined,
  newValue: 'white',
  oldContentType: undefined,
  newContentType: 'text/plain'
}
```

Hint: You may also use for example *`getAll()`* instead, and choose to deactivate these.

remote

Events with origin `remote` are fired when remote changes are discovered during sync.

Note: Automatically receiving remote changes depends on the *caching* settings for your module/paths.

window

Events with origin *window* are fired whenever you change a value by calling a method on the `BaseClient`; these are disabled by default.

Hint: You can enable them by configuring `changeEvents` for your *RemoteStorage* instance.

`conflict`

Events with origin `conflict` are fired when a conflict occurs while pushing out your local changes to the remote store.

Say you changed 'color.txt' from 'white' to 'blue'; if you have set `config.changeEvents.window` to `true` (by passing it to the `RemoteStorage` constructor, see the Constructor section of the *RemoteStorage API doc*), then you will receive:

```
{
   path: '/public/design/color.txt',
   relativePath: 'color.txt',
   origin: 'window',
   oldValue: 'white',
   newValue: 'blue',
   oldContentType: 'text/plain',
   newContentType: 'text/plain'
 }
```

But when this change is pushed out by asynchronous synchronization, this change may be rejected by the server, if the remote version has in the meantime changed from 'white' to for instance 'red'; this will then lead to a change event with origin 'conflict' (usually a few seconds after the event with origin 'window', if you have those activated). Note that since you already changed it from 'white' to 'blue' in the local version a few seconds ago, `oldValue` is now your local value of 'blue':

```
{
   path: '/public/design/color.txt',
   relativePath: 'color.txt',
   origin: 'conflict',
   oldValue: 'blue',
   newValue: 'red',
   oldContentType: 'text/plain',
   newContentType: 'text/plain',
   // Most recent known common ancestor body of local and remote
   lastCommonValue: 'white',
   // Most recent known common ancestor contentType of local and remote
   lastCommonContentType: 'text/plain'
 }
```

4.2.3 Data types

`declareType` (*alias*, *uri*, *schema*)

Declare a remoteStorage object type using a JSON schema.

See *Defining data types* for more info.

Arguments

- **`alias`** (*`string`*) – A type alias/shortname
- **`uri`** (*`uri`*) – (optional) JSON-LD URI of the schema. Automatically generated if none given
- **`schema`** (*`object`*) – A JSON Schema object describing the object type

Example:

```
client.declareType('todo-item', {
  "type": "object",
  "properties": {
    "id": {
      "type": "string"
    },
    "title": {
      "type": "string"
    },
    "finished": {
      "type": "boolean"
      "default": false
    },
    "createdAt": {
      "type": "date"
    }
  },
  "required": ["id", "title"]
})
```

Visit http://json-schema.org for details on how to use JSON Schema.

`validate` (*object*)

Validate an object against the associated schema.

Arguments

- **`object`** (`Object`) – JS object to validate. Must have a `@context` property.

Returns **Object** – An object containing information about validation errors

Example:

```
var result = client.validate(document);

// result:
// {
//   error: null,
//   missing: [],
//   valid: true
// }
```

4.2.4 Caching

`cache` (*path*, *strategy*)

Set caching strategy for a given path and its children.

See *Caching strategies* for a detailed description of the available strategies.

Arguments

- **`path`** (`string`) – Path to cache
- **`strategy`** (`string`) – Caching strategy. One of 'ALL', 'SEEN', or 'FLUSH'. Defaults to 'ALL'.

Returns **BaseClient** – The same instance this is called on to allow for method chaining

Example:

```
client.cache('documents/', 'ALL');
```

flush(*path*)
TODO: document

Arguments

- **path** (*string*) –

Example:

```
client.flush('documents/');
```

4.2.5 Other functions

getItemURL(*path*)
Retrieve full URL of a document. Useful for example for sharing the public URL of an item in the `/public` folder.

Arguments

- **path** (*string*) – Path relative to the module root.

Returns **string** – The full URL of the item, including the storage origin

Warning: This method currently only works for remoteStorage backends. The issues for implementing it for Dropbox and Google Drive can be found at #1052 and #1054.

scope(*path*)
Instantiate a new client, scoped to a subpath of the current client's path.

Arguments

- **path** (*string*) – The path to scope the new client to.

Returns **BaseClient** – A new client operating on a subpath of the current base path.

4.3 Access

This class is for requesting and managing access to modules/folders on the remote. It gets initialized as `remoteStorage.access`.

4.3.1 List of functions

claim(*scope*, *mode*)
Claim access on a given scope with given mode.

Arguments

- **scope** (*string*) – An access scope, such as "contacts" or "calendar"
- **mode** (*string*) – Access mode. Either "r" for read-only or "rw" for read/write

Example:

```
remoteStorage.access.claim('contacts', 'r');
remoteStorage.access.claim('pictures', 'rw');
```

Claiming root access, meaning complete access to all files and folders of a storage, can be done using an asterisk for the scope:

```
remoteStorage.access.claim('*', 'rw');
```

4.4 Caching

The caching class gets initialized as `remoteStorage.caching`, unless the *RemoteStorage* instance is created with the option `cache: false`, disabling caching entirely.

In case your app hasn't explictly configured caching, the default setting is to cache any documents that have been either created or requested since your app loaded. For offline-capable apps, it usually makes sense to enable full, automatic caching of all documents, which is what `enable()` will do.

Enabling full caching has several benefits:

- Speed of access: locally cached data is available to the app a lot faster.
- Offline mode: when all data is cached, it can also be read when your app starts while being offline.
- Initial synchronization time: the amount of data your app caches can have a significant impact on its startup time.

Caching can be configured on a per-path basis. When caching is enabled for a folder, it causes all subdirectories to be cached as well.

4.4.1 Caching strategies

For each subtree, you can set the caching strategy to `ALL`, `SEEN` (default), and `FLUSH`.

- `ALL` means that once all outgoing changes have been pushed, sync will start retrieving nodes to cache pro-actively. If a local copy exists of everything, it will check on each sync whether the ETag of the root folder changed, and retrieve remote changes if they exist.
- `SEEN` does this only for documents and folders that have been either read from or written to at least once since connecting to the current remote backend, plus their parent/ancestor folders up to the root (to make tree-based sync possible).
- `FLUSH` will only cache outgoing changes, and forget them as soon as they have been saved to remote successfully.

4.4.2 List of functions

`enable` (*path*)

Enable caching for a given path.

Uses caching strategy `ALL`.

Arguments

- **`path`** (*`string`*) – Path to enable caching for

Example:

```
remoteStorage.caching.enable('/bookmarks/');
```

`disable` (*path*)

Disable caching for a given path.

Uses caching strategy `FLUSH` (meaning items are only cached until successfully pushed to the remote).

Arguments

- **`path`** (*`string`*) – Path to disable caching for

Example:

```
remoteStorage.caching.disable('/bookmarks/');
```

`set` (*path*, *strategy*)

Configure caching for a given path explicitly.

Not needed when using `enable`/`disable`.

Arguments

- **`path`** (*`string`*) – Path to cache
- **`strategy`** (*`string`*) – Caching strategy. One of 'ALL', 'SEEN', or 'FLUSH'.

Example:

```
remoteStorage.caching.set('/bookmarks/archive/', 'SEEN');
```

`checkPath` (*path*)

Retrieve caching setting for a given path, or its next parent with a caching strategy set.

Arguments

- **`path`** (*`string`*) – Path to retrieve setting for

Returns **string** – caching strategy for the path

Example:

```
remoteStorage.caching.checkPath('documents/').then(strategy => {
  console.log(`caching strategy for 'documents/': ${strategy}`));
  // "caching strategy for 'documents/': SEEN"
});
```

`reset` ()

Reset the state of caching by deleting all caching information.

Example:

```
remoteStorage.caching.reset();
```

CHAPTER

FIVE

USAGE WITH NODE.JS

Although remoteStorage.js was initially written for being used in browsers, we do support using it in a Node.js environment as well. See *this section* for getting started.

The main difference between rs.js in a browser and using it on a server or in a CLI program is how to connect a storage. The RS protocol uses the OAuth Implicit Grant flow for clients to receive a bearer token, which they can use in HTTP requests. This works by redirecting back to the Web application with the token attached to the redirect URI as a URI fragment.

Now, with rs.js in a browser, calling `remoteStorage.connect('user@example.com')` will take care of the entire OAuth process, including the parsing of the URI after the redirect, saving the token to localStorage and changing the library's state to connected. But in a node.js program, that's obviously not possible, because there's no browser that will open the OAuth dialog and receive the redirect with the token attached to the redirect URI.

5.1 connect() with a token

For this reason, among others, you can call the connect function with a token that you acquired beforehand:

```
remoteStorage.connect('user@example.com', 'abcdefghijklmnopqrstuvwxyz')
```

This will skip the entire OAuth process, because you did that before in some other way, of course.

5.2 Obtaining a token

For some programs, like e.g. a server daemon, you can usually acquire the token from your server manually, and then just configure it for example as environment variable, when running your program.

For CLI programs, and if you actually want to integrate the OAuth flow in your program, one possible solution is the following:

1. Set up a simple Web site/app, which you publish under a fitting domain/URI that you can use as the OAuth redirect URI.
2. Have the user enter their user address and do a Webfinger lookup for auth URL etc., e.g. using webfinger.js.
3. Create the OAuth request URI with the correct scope etc., and open a browser window with that URI from your program (or prompt the user to open it).
4. Have the Web app, which the user is being redirected to, show the token to the user, in order for them to copy and enter in your program
5. Connect with that token.

You can find a complete example for this process in rs-backup, a remoteStorage backup CLI program. In particular its code for connecting a storage and the simple Web page its using for the redirect.

Hint: rs-backup is not using remoteStorage.js at all, which you might also want to consider as an option when writing non-browser applications.

5.3 Caveats

- IndexedDB and localStorage are not supported by default in Node.js, so the library will fall back to in-memory storage for caching data locally. This means that unsynchronized data will be lost between sessions and program executions.

5.4 Examples

- hubot-remotestorage-logger, a Hubot script that logs chat messages to remoteStorage-enabled accounts using the chat-messages module

CHAPTER

SIX

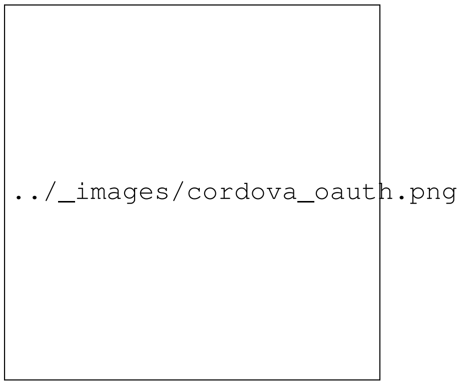
USAGE IN CORDOVA APPS

Apache Cordova is a mobile development framework. It allows you to use standard web technologies - HTML5, CSS3, and JavaScript for cross-platform development. Applications execute within wrappers targeted to each platform, and rely on standards-compliant API bindings to access each device's capabilities such as sensors, data, network status, etc.[1]

To use remoteStorage.js in a Cordova app, you need to have the InAppBrowser plugin installed.

Cordova apps are packaged for the different platforms and installed on the device. The app doesn't need to be hosted as a web app (although it can be as well). But for the remoteStorage connection to work, you need to provide a page that is accessible via a public URL. This will be used as the redirect URL during the OAuth flow.

When a user connects their storage, the OAuth dialog will open in an in-app browser window, set to show the address to prevent phishing attacks.



After the user authorizes the app, the server will redirect to the configured redirect URL with the authorization token added as a parameter. remoteStorage.js will intercept this redirect, extract the token from the URL and close the window.

So the user doesn't actually see the page of the redirect URL and it does't need to have the remoteStorage.js library included or have any special logic at all. But you should still make sure that it can be identified as belonging to your app. Storage providers will usually show the URL in the OAuth dialog, and they may also link to it (e.g. from the list of connected apps).

You can configure the redirect URL for your app, either by calling

```
remoteStorage.setCordovaRedirectUri('https://myapp.example.com');
```

or as config when creating your rs instance:

[1] Taken from https://cordova.apache.org/docs/en/latest/guide/overview/index.html

```
const remoteStorage = new RemoteStorage({
  cordovaRedirectUri: 'https://myapp.example.com'
});
```

No further action is needed and you can now use remoteStorage.js as with any other web app.

6.1 Google Drive config

If you wish to use the optional Google Drive adapter, you need to configure a different user agent for your app. Otherwise the authorization page will show an error to the user.

In case you haven't set your own UA string already, here's how you can do it:

```
<preference name="OverrideUserAgent" value="Mozilla/5.0 remoteStorage" />
```

CHAPTER

SEVEN

CONTRIBUTING

This section contains information and help for people wanting to contribute to remoteStorage.js development.

7.1 Code overview

The code of remoteStorage.js consists of files in the `src/` folder of this repo. These are built into a single file in the `release/` folder using webpack. Unit tests live in the `test/` folder and are based on Jaribu.

The structure of the code is based around feature loading. Most files in `src/` correspond to a feature, e.g. `discover.js` to `RemoteStorage.Discover` or `caching.js` to `RemoteStorage.Caching`.

The feature loading happens synchronously during the page load in `src/remotestorage.js` (just including this script in your app will lead to executing the code that loads the features).

Most features load under their own name, but for `remoteStorage.local` a choice is made between `RemoteStorage.IndexedDB`, `RemoteStorage.LocalStorage` and `RemoteStorage.InMemoryCaching`, depending on what the environment (browser, node.js, Electron, WebView, or other) supports.

For `remoteStorage.local` we then also have a special mixin called `src/cachinglayer.js`, which mixes in some common functions into the object.

The `remoteStorage.remote` feature is not loaded immediately, but only when `RemoteStorage.Discover` calls `remoteStorage.setBackend()`, at which point a choice is made between `RemoteStorage.WireClient`, `RemoteStorage.GoogleDrive`, `RemoteStorage.Dropbox` (or any other future backend) to become the `remote`.

7.2 Building

Hint: We're using npm scripts for all common tasks, so check out the `scripts` section in `package.json` to learn about what they're doing exactly and what else is available.

7.2.1 Development

```
$ npm run dev
```

This will watch `src/` for changes and build `remotestorage.js` in the `release/` directory every time you save a source file. Useful for testing rs.js changes with an app, for example by creating a symlink to `release/remotestorage.js`.

This build includes [source maps](https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/) directly, so you can easily place `debugger` statements in the code and step through the actual source code in your browser's debugger tool.

7.2.2 Production

```
$ npm run build
```

This creates the minified production build in `release/`.

It also creates a seperate source maps file, which you can link to in case you want to (e.g. to improve exception tracking/debugging in production).

7.3 Testing

Before contributing to remoteStorage.js, make sure your patch passes the test suite, and your code style passes the code linting suite.

We use the Jaribu framework for our test suites and JSHint for linting. Both are set as dev dependencies in `package.json`, so after installing those via `npm install`, you can use the following command to run everything at once:

```
$ npm run test
```

Or you can use the Jaribu executable directly in order to run the suite for a single file:

```
$ ./node_modules/.bin/jaribu test/unit/cachinglayer-suite.js
```

Tip: If you add `./node_modules/.bin` to your `PATH`, you can call executables in any npm project directly. For example in `~/.bashrc`, add the line `export PATH=$PATH:./node_modules/.bin` (and run `source ~/.bashrc` to load that change in open terminal sessions). Then you can just run `jaribu test/unit/foo_suite.js`.

7.3.1 Continous integration

The rs.js test suite is run by Travis CI on every push to our repo on GitHub. When you open a pull request, your code will be tested there, too. You can check out the build status and history at https://travis-ci.org/remotestorage/remotestorage.js/, and the CI settings in `.travis.yml`.

7.4 Documentation

The documentation for remoteStorage.js is generated from reStructuredText files in the `doc/` folder, as well as JSDoc code comments, which are being pulled in via special declarations in those files.

We use Sphinx to generate the documentation website, and the sphinx-js extension for handling the JSDoc part.

7.4.1 How to write reStructuredText and JSDoc

For learning both the basics and advances features of reStructuredText, we highly recommend the reStructuredText Primer on the Sphinx website.

For JSDoc, you can find an intro as well as a detailed directive reference on their official website.

7.4.2 Automatic builds and publishing

The documentation is published via Read the Docs. Whenever the Git repository's `master` branch is pushed to GitHub, RTD will automatically build a new version of the site and publish it to remotestoragejs.readthedocs.io.

This means that if you want to contribute to the documentation[1], you don't necessarily have to set up Sphinx and sphinx-js locally (especially for small changes). However, if you want to preview what your local changes look like when they are rendered as HTML, you will have to set up local builds first.

7.4.3 How to build the docs on your machine

Setup

1. Install Python and PIP (likely already installed)
2. Install sphinx:

```
$ pip install sphinx
```

3. Install required extensions (from repository root):

```
$ pip install -r doc/requirements.txt
```

4. Install JSDoc globally via npm:

```
$ npm install -g jsdoc
```

Build

Run the following command to automatically watch and build the documentation:

```
$ npm run doc
```

This will start a web server, serving the HTML docs on http://localhost:8000.

[1] Every single bit helps other people! Even fixing a typo is worth a pull request.

Hint: The autobuild cannot watch for changes in JSDoc comments as of now, so you will need to re-run the command, or change something in a `.rst` file in order for code documentation changes to be re-built.

7.5 GitHub workflow

7.5.1 General guidelines

- When you start working on an existing GitHub issue (or you plan on doing that in the immediate future), assign it to yourself, so that others can see it and don't start working on it in parallel.
- When you create a branch to work on something, use the naming scheme described further down in this document.
- Never push directly to the `master` branch for any changes to the source code itself.
- As soon as you want others to review your changes, or even just discuss them, create a pull request. Don't forget to explain roughly what it is you're doing in that branch, i.e. what the problem/idea is and what the result is supposed to be, when merging the changes. If necessary or helpful, mention related discussions from other issues.
- A pull request can be merged as soon as at least two people with commit access to the repo have given a +1, meaning they reviewed and tested the changes and have no further improvements to suggest.

7.5.2 Branch names

Using common branch names, that include topics and issue IDs, makes everyone's lives much easier, and keep the repo clean. Branches on our organization repositories should be created using the following scheme:

`[bugfix|feature|docs|refactor]/[issue id]-[description_with_underscores]`

So for example, if you want to work on fixing a bug with let's say initial sync, that is described in issue #423, the branch should look something like:

`bugfix/423-race_condition_on_initial_sync`

And if it's an enhancement to the widget it could look like this e.g.:

`feature/321-customizable_widget_content`

If there's no issue yet, create one first!

7.5.3 Pulling changes

Always use `--rebase` when pulling code from the org repo. That way your local changes are added on top of the current history, avoiding merge commits and the mixing up the commit history. You can set up Git to use rebase by default by running `git config --global branch.autosetuprebase always` once.

The easiest way to update your local repository with all remote changes, including all branches – old and new – is a tool called git-up. With that installed you can just run `git up` instead of `git pull [options]`, which will fetch all branches from all remotes and rebase your commits on top of them (as well as stash and unstash uncommitted code, if necessary).

7.5.4 Commit messages

- The first line of the message (aptly called "subject line" in Git terminology) should not be longer than 72 characters.
- If the subject line is not enough to describe the changes properly, add a blank line after the subject line and then as much text as you want, using normal language with capitalization, punctuation, etc.
- Always use messages that describe roughly *what* the change does and, if not obvious, *why* this change leads to the desired result.
- Leave out any text that isn't directly associated with the changes, that the commit introduces. Examples: "as suggested by @chucknorris", "lol wtf was that", "not sure if this fixes it".
- Commit as much and often as possible locally (and with any message that helps you during your work), and then clean up the history and merge commits that belong together before pushing to the org repo. You can do that with `git rebase -i [ref]` (learn more).
- You can reference issues from commit messages by adding keywords with issue numbers. Certain keywords will even close the issue automatically, once a branch is merged into master. For example `Fix widget flickering when opening bubble (fixes #423)` will close issue #423 when appearing on the master branch at GitHub.

7.5.5 Reviewing pull requests

- Check if it works, if it has unit tests, if the tests pass, and if JSHint and CodeClimate are happy.
- Check if the code is understandable, with clear and unambiguous names for functions and variables, and that it has JSDoc comments and a changelog entry.
- If you use `git up`, like recommended above, it will automatically create tracked branches for all remote branches. So in order to review/test a branch on the org repo, just do `git checkout [branchname]`. You can then also add new commits to that branch and push them in order to add your changes to the pull request.
- If the pull request was issued from a user's own repository, you'll have to fetch the code from there, of course. If you haven't pulled from that user yet, you can add a new remote for the user with `git remote add [username] [repo-url]`. After that, `git up` will fetch code from that remote as well, so you can then check it out using `git checkout [username]/branchname`.

(This will put you in a so-called 'detached HEAD' state, but don't worry, everything is fine! If you want to work on that code, just create a new branch from there with the command Git shows you then, or just go back to your code with e.g. `git checkout master` later.)

7.5.6 Merging pull requests

- Once a pull request has two +1s for the latest changes from collaborators, you can either merge it yourself or wait for somebody to do it for you (which will happen very soon).
- If the new commits and their commit messages in that branch all make sense on their own, you can use the merge button on GitHub directly.
- If there are a lot of small commits, which might not make sense on their own, or pollute the main project history (often the case with long running pull requests with a lot of additions during their lifetime), fetch the latest changes to your local machine, and either do an interactive rebase to clean up branch and merge normally, or use `git merge --squash` to squash them all into one commit during the merge.

- Whenever you squash multiple commits with either `git rebase -i` or `git merge --squash`, make sure to follow the commit message guidelines above. Don't just leave all old commit messages in there (which is the default), but delete them and create a new meaningful message for the whole changeset.
- When squashing/editing/amending other peoples' commits, use `--author` to set them as the original author. You don't need full names for that, but just something that Git can find in the history. It'll tell you if it can't find an author and let you do it again.

7.6 Release checklist

- Build library and manually test all browsers you have access to, including mobile devices and private browsing mode
- Create changelog since last release
  - Collect and summarize changes using e.g.:

    ```
    git log --no-merges <LAST RELEASE TAG>..HEAD
    ```

  - Add changes to *CHANGELOG.md*
  - Commit to Git
- Run `npm version patch|minor|major|x.x.x-rc1`. This will automatically:
  - run the test suite
  - update the version in package.json
  - update the version in bower.json
  - create a release build
  - commit everything using version as commit description
  - create a Git tag for the version
  - push the release commit and tag to GitHub
- Publish release notes on GitHub
  - Go to https://github.com/remotestorage/remotestorage.js/tags and click "Add release notes"
  - Use version string as title and changelog items as description
  - For RCs and betas, tick the "This is a pre-release" option on the bottom
  - These notes will automatically be posted *to the community forums <https://community.remotestorage.io/t/release-updates-for-rs-libraries/433>* after a while
- Publish to npm (https://www.npmjs.org/package/remotestoragejs):

  ```
  npm publish
  ```

- Update https://github.com/remotestorage/myfavoritedrinks to use new release
  - Replace `remotestorage.js` file with new release build
  - Check if everything is still working
  - Commit
  - `git push origin`

  - `git push 5apps master`

- Link release announcement on Mastodon (remoteStorage@kosmos.social). This will automatically cross-post to Twitter and IRC.
- If it's an important release, also notify the Unhosted mailing list

7.7 Libary internals

This section contains information about some of the internals and concepts of the remoteStorage.js library.

7.7.1 Discovery bootstrap

This section describes how connecting to a storage works internally.

When the RemoteStorage instance is instantiated, it checks the fragment of the URL to see if it contains an `access_token` or `remotestorage` parameter. In the first case, the access token is given to the remote using `remoteStorage.remote.configure()`. In the second case, WebFinger discovery is triggered for the user address given (see storage-first section of the remoteStorage spec).

The user can also set the user address through the widget, or the app can call `remoteStorage.remote.configure({userAddress: 'user@host.com'})` to set the user address.

When a user address is set, but no other remote parameters are known yet, WebFinger discovery will be triggered. From the WebFinger response, the library extract the storage base URL, the storage API, and the OAuth dialog URL.

If no OAuth URL is given, Implied Auth is triggered: https://github.com/remotestorage/remotestorage.js/issues/782

If an OAuth URL is known, but no token yet, the OAuth dance will be started by setting the `location.href` of the window, redirecting the user to that URL. When the dance comes back, the library will detect the `access_token` from the window location during the page load, and from that point onwards, the remote is connected.

7.7.2 Caching

The caching strategies are stored in `remoteStorage.caching._rootPaths`. For instance, on https://myfavoritedrinks.remotestorage.io/, it has the value `{ /myfavoritedrinks/: "ALL" }`.

These rootPaths are not stored in localStorage. If you refresh the page, it is up to the app to set all caching strategies again during the page load.

The effect of the caching strategy is basically achieved through three paths:

1. Setting caching strategy 'ALL' for a path creates an empty node for that path, unless it already exists.
2. The sync process will then do a GET request, and create new nodes under any folder with an 'ALL' strategy, when that folder is fetched.
3. The sync process will create a new task for any node under an 'ALL' strategy, unless a task already exists for one of its ancestors.

The result is all paths with an explicit 'ALL' strategy will get fetched, and if they are folders, then in the next round, all its children will also be fetched, etcetera.

7.7.3 Data format of the local cache

This section describes the structure and concepts of the local cache.

Storing up to 4 revisions of each node

Each cache node represents the versioning state of either one document or one folder. The versioning state is represented by one or more of the `common`, `local`, `remote`, and `push` revisions. Local changes are stored in `local`, and in `push` while an outgoing request is active. Remote changes that have either not been fetched yet, or have not been merged with local changes yet, are stored in `remote`.

autoMerge

The `sync.autoMerge` function will try to merge local and remote changes into the common revision of a node. It may emit change events with a 'conflict' origin to indicate that an unpushed local change was overruled by a remote change.

When consulting the base client about the current value of a node, you will get either its 'local' revision if it exists, or its 'common' revision otherwise. The following are versioning tree diagrams of how local and remote revisions of a node can interact:

```
//in sync:
1)   . . . . [common]

//dirty:
2)   . . . . [common]
                 \
                  \ . . . . [remote]

//local change:
3)   . . . . [common] . . . . [local]

//conflict (should autoMerge):
4) . . . . . [common] . . . . [local]
                \
                 \ . . . . [remote]

//pushing:
5)   . . . . [common] . . . . [push] . . . . [local]

//pushing, and known dirty (should abort the push, or just wait for the conflict to␣
↪occur):
6)   . . . . [common] . . . . [push] . . . . [local]
                 \
                  \ . . . . [remote]
```

Each of `local`, `push`, `remote`, and `common` can have have following properties:

- for documents:
  - body
  - contentType
  - contentLength
  - revision

    - timestamp

- for folders:
    - itemsMap (itemName -> true, or itemName -> false to indicate an unmerged deletion)
    - revision
    - timestamp

NB: The timestamp represents the last sync time, not the last modified time. It is used by the `isOutdated` function in `src/cachinglayer.js` to determine if the data needs to be fetched from remote again, or can be served from cache.

“keep/revert” conflict resolution

RemoteStorage implements a hub-and-spokes versioning system, with one central remoteStorage server (the hub) and any number of clients (the spokes). The clients will typically be unhosted web apps based on this JS lib (remotestorage.js), but they don’t have to be; they can also be based on other client implementations, they can be hosted web apps, desktop apps, native smartphone apps, etcetera. New versions of subtrees always start at one of these clients. They are then sent to the server, and from there to all the other clients. The server assigns the revision numbers and sends them to the initiating client using HTTP ETag response headers in response to PUT requests. remotestorage.js is a library that attempts to make it easy to build remoteStorage applications, by hiding both the push/pull synchronization and the version merging from the app developer. Versioning conflicts between the local client and the remote server are initially resolved as a ‘remote wins’, to which the client code may respond with an explicit revert (putting the old, local version back), any type of custom merge (putting the result of the merge in place), or by doing nothing (“keep”), and leaving the remote result in place. This system is called “keep/revert”, since the library takes a pro-active action (‘remote wins’), which the app can then either keep, or revert.

Sync is tree-based: syncing a node is equivalent to syncing all its children. There are two parts at play, that interact: transporting the diffs to and from the remote server, and merging the local and remote versions into one common version. Each document starts out as non-existing in both its local and remote versions. From there on, it can be created, updated, and deleted any number of times throughout its history, both locally and remotely. If at some point in time it either does not exist neither locally nor remotely, or its body and content-type are the same byte-for-byte on both sides, then the two stores are in agreement. If the document exists in only one of the stores, or the document’s body or its content-type differs between the two stores, then the document is in conflict.

The library is always aware of the latest local version, but it may or may not be aware of the latest remote version, and therefore of whether a conflict or agreement exists for the document. Likewise, the server is not necessarily aware of the latest local version, if there are changes that haven’t been pushed out yet; nor does it care, though, since the server does not get involved in conflict resolution. It only serializes conditional updates from all clients into one canonical versioning history.

The lack of sync between client and server can be fixed by doing a GET, PUT, or DELETE. A GET will return the current remote version; a conditional PUT or DELETE will push out the change, while at the same time checking if any unfetched remote changes exist. If they do, then the push will fail, and the library will fetch instead. After this, the library has a latest known common revision of the document, possibly a local version if it was changed since then, and possibly a remote version if it was changed since then, but the newer version has yet to be retrieved.

Before resolving a conflict, both revision histories are squashed. This means that creating+deleting a document becomes a noop, and deleting+creating, or updating it multiple times, becomes one single update. Then, if the document was changed in different ways locally and remotely, it goes into conflict state; if it was changed only locally or only remotely, then the change is automatically accepted by the other store (whether client to server or server to client). Note that in the case of a successful conditional push request, this will already have happened.

Conflicts that are discovered by a document fetch, fire their ‘keep/revert’ event immediately. Conflicts that are discovered through a parent folder fetch, or through a conditional push, fire their ‘keep/revert’ event after the new remote version is fetched.

The library's conflict resolution strategy is 'remote wins'. This means that the module will receive them in the form of change events with origin 'conflict'. When receiving such a change event, the module can still decide to revert it explicitly.

As noted before, merging a subtree is done by merging each document that exists within that subtree, in either or both stores. When the library fetches a folder listing, it can detect a remote child change, which then may or may not result in a conflict. When a folder listing comes in, which has changed since the last time it was retrieved, four types of information may be discovered:

- which of the documents directly within the folder changed their remote revision since the last check (new ETag on a document item)
- in which previously empty subtrees at least one document was created (new folder item)
- in which subtrees all previously existing documents were deleted (folder item disappeared)
- in which subtrees at least one document was either created, updated, or deleted (new ETag on a folder item)

All of these can occur in a folder that was at the same time either unchanged, updated, or deleted locally. When updated, it might be that different items were changed locally and remotely, or that the same item was changed on both sides, either in the same way, or in different ways.

The library handles all these cases so the module developer does not need to worry about them.

Implications for module design

There are a number of important implications for module design:

- First of all, this sync process follows the 'asynchronous synchronization' design principle (https://github.com/offlinefirst/research/issues/9). Don't wait for it to finish. The module should work with the local copy of the data, and handle incoming updates through evented programming. The only exception to this is where a body of data is too big to cache locally, and the module needs to expose on-demand access of remote data to the app. In all other cases, the module should expose the local version as 'the truth'.
- Even then, IndexedDB is not fast enough to access from a button click. Make sure to put an in-memory caching layer in the module, and return control to the app immediately. An example of this approach is the SyncedMap data structure used in https://github.com/michielbdejong/meute.
- Use folders and subfolders. This allows the tree-based sync algorithm to shine and efficiently detect changes in any of potentially thousands of documents by checking the ETag from one single HTTP request to the root folder of the tree.
- Use meaningful collections. Multiple clients can each edit a different document without ever entering in conflict with each other. But editing the same document is interpreted as a conflict. For instance, when two calendar apps both schedule an event on a certain date, this would be a conflict if the module stores one document per day. However, if the module stores one document per event, and instead uses one /folder/ for each day, then the two events can co-exist on the same day without generating a conflict. Documents are a unit of conflict, but folders are not. Another example is storing todo-list items with long UUID hashes instead of their list index numbers as document names. Editing item "5" would conflict with inserting a new item "5". But if both items have a long unique name, then they don't clash with each other. So make sure to choose unique item names for items that should not conflict.

CHAPTER EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX