

MANDELROT APPS

FRONTEND DEVELOPER GUIDE

Contents

- Intro
- Features
- The APPS folder
- Minimum required app structure
 - Required folder/files
 - Auto-generated stuff
- The app-data file
- Your backend functions files
- Requesting internal resources
 - How to format the requests
 - Config for internal routing
- Working with modules
 - Requiring your own modules
 - package.json
 - The utils/public folder
- Communications
 - The message standard format
 - Communications with your own backend functions
 - With other apps backend functions
 - With the backend utils/public modules
 - Broadcasting

Intro

This document is for developers who already know what's the MApps backend about (if not please go to see the "overview guide" available) and want to use it to make their own frontend apps. If this is the case, with this document you will understand all you need to get to work.

All the updated info, including the guides and some basic example apps to see how the system works (don't miss those!), in the project GitHub repo:

https://github.com/mandelrot/mapps_backend

This software has been made by Jose Alemán / Mandelrot. You can download the code and use it freely, including changing whatever you want. In case you want to know more or contact me:

My blog: <https://mandelrot.com> (spanish)

My professional webpage: <http://josealeman.info> (english)

Features

In the MApps system the backend is just a basic engine that recognizes the frontend apps, sets up a server where the admin can enable them for the final users, gives the users a main page where they can see in real time the available (enabled) frontend apps, and then route the frontend petitions among them. It provides also some public common utilities in case the front apps want to use them (example: an encryption module), although they don't have to.

So when you develop your app you need to know that there's no actual backend functions waiting to handle your requests, nor a backend database to store anything (unless there's another frontend app doing that, and you should point to it then). You need to create both parts and include them in your package. If this is not clear now don't worry, you will understand everything later.

In the same frontend package, there will be a sub-folder where you will place two functions modules (more on this later) that will be the ones the backend engine will require. You will have to create an "internal.js" module where your own frontend will find private functions to invoke, and another "external.js" where the public functions will be available. When the backend engine gets a message from a frontend app asking for a function of yours, if the sender is the same than the target app it will look for "internal.js" and if it's a different app asking it will look for "external.js". The backend will take care of the routing and will know which module to require dependind on who is calling.

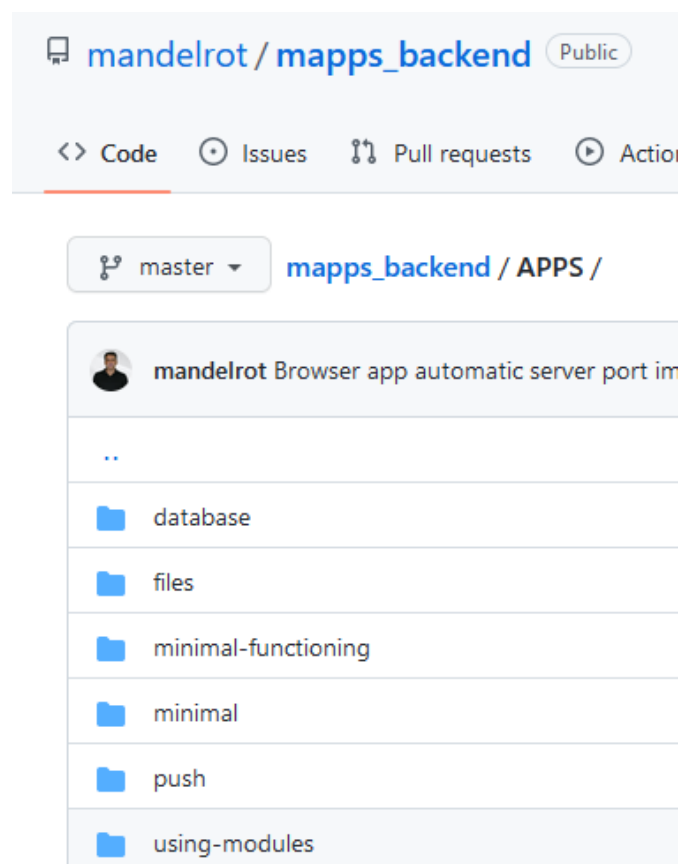
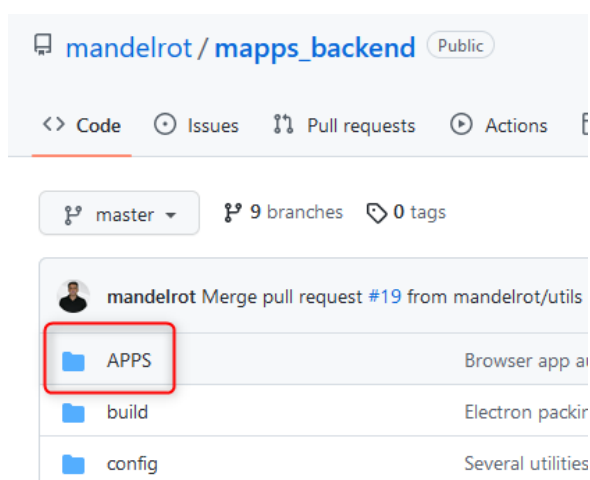
And, since everything is connected via socket, you can trigger backend events when you want and send them to other apps that will be connected and listening (typical use case: refreshing data).

Please note: your dev OS doesn't matter for the production environment. You can program in Windows with exactly the same code the admin will copy to a Linux server for example. The content of your frontend app folder will be exactly the same everywhere.

The APPS folder

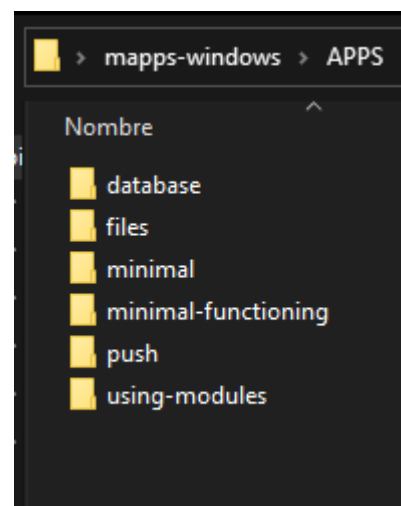
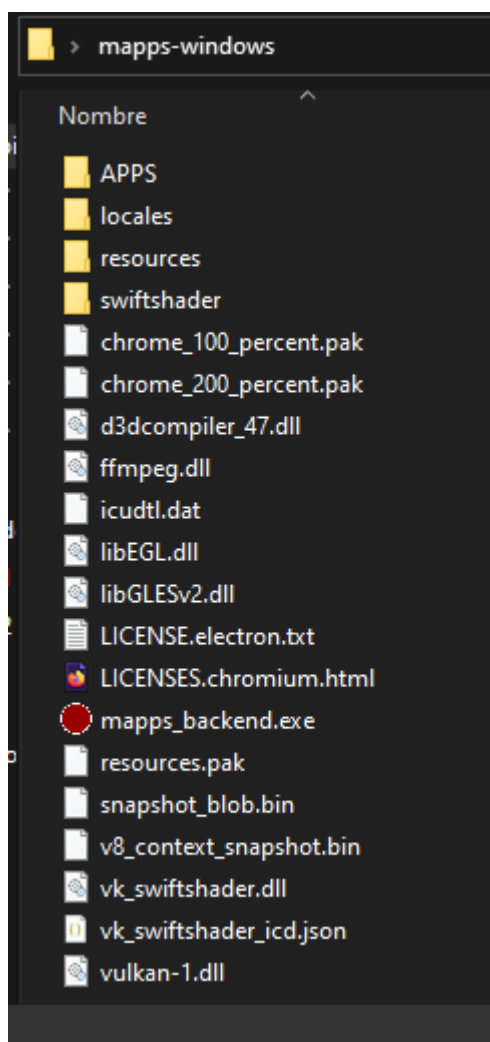
This chapter contains sysadmin information that in theory is not for developers; but you will need to know all this to set a dev environment for your front app. If you need to compile a fresh engine copy please check the admin guide available and there you will find all the steps detailed at max; here we will suppose you already have a compiled copy of the software, and we will take the main engine folder as root.

In a fresh copy you wouldn't have any app installed yet and you should install yours at first (more on this in a moment). However, if you want to test some apps already working, in the GitHub repo you will find some demo apps I made as an example and we will use them for this tutorial:



When you compile a fresh copy of the package there's an APPS folder not present yet (it should be in the executable package root folder), but it's where the frontend apps should be. You could create it yourself, but it's not necessary if you don't want: just running the engine once (the executable file) it will be auto-generated. If so, you will see the APPS folder contains a file named "apps.state"; just don't mind it, it's an internal registry file that gets generated when needed. You can delete it and it will appear again by itself, simply ignore it.

So, to be all in the same page, we will start with a folder with the portable package of the engine, and inside it an APPS folder with the example apps you can find in Github. This is Windows, Linux would be similar:



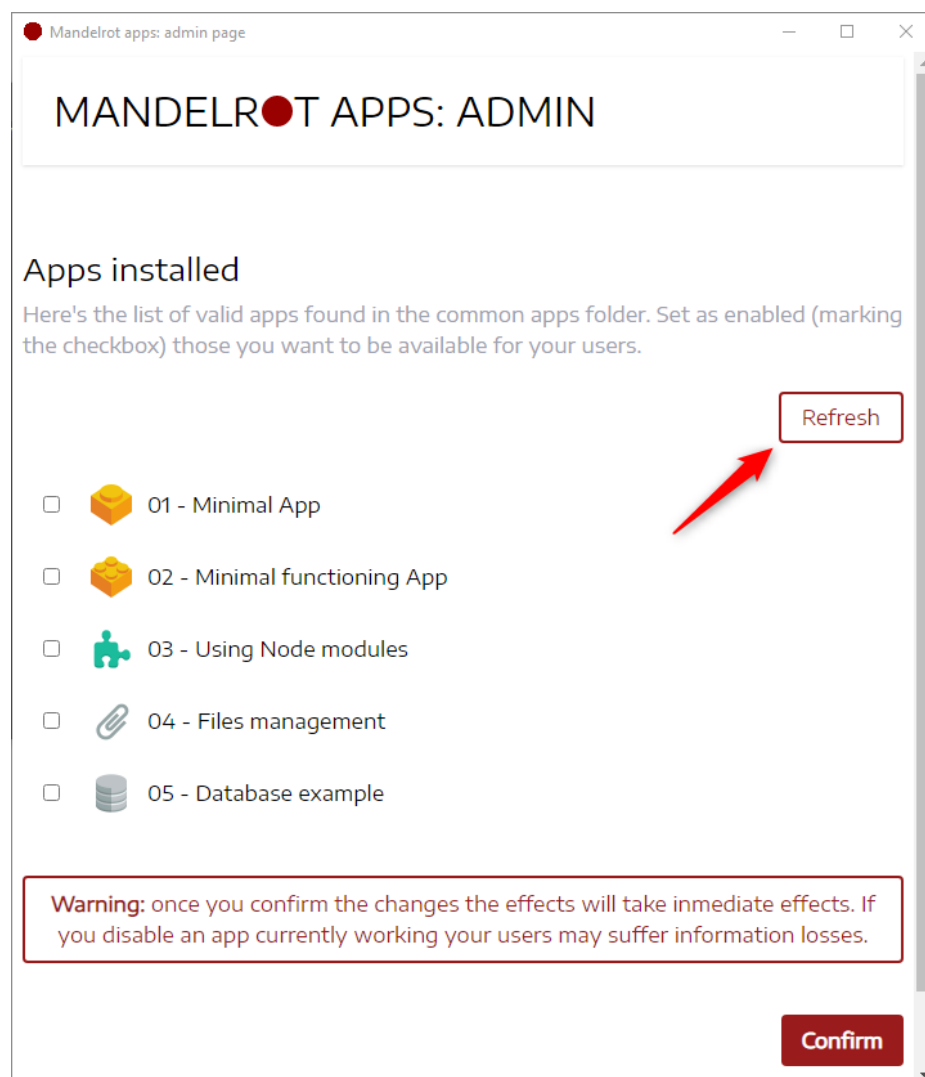
(Please note: the name "APPS" could be changed in the configuration file before compilation. This is the default name and there's no reason to change it, but it's just to let you know in case your admin has set something different).

Inside the APPS folder it you could place anything, but only the folders with a valid internal structure will be recognized by the backend as a part of the suite and the rest will be ignored. More on this later.

Very important: the front app folder name will be used in the internal routes and should NOT be changed in production (it works as an unique ID for your app). The apps folders named “admin” or “main” will be ignored.

Having the APPS folder empty or using the example apps I left in GiHub, the fact is that you will need to “install” your frontend app connecting it to the backend engine to see it working. How do we do that?

When we run the executable file and start the backend engine, the system checks the APPS folder and shows all the valid apps present in it. At start you will get a (maximized) admin window showing something like this:

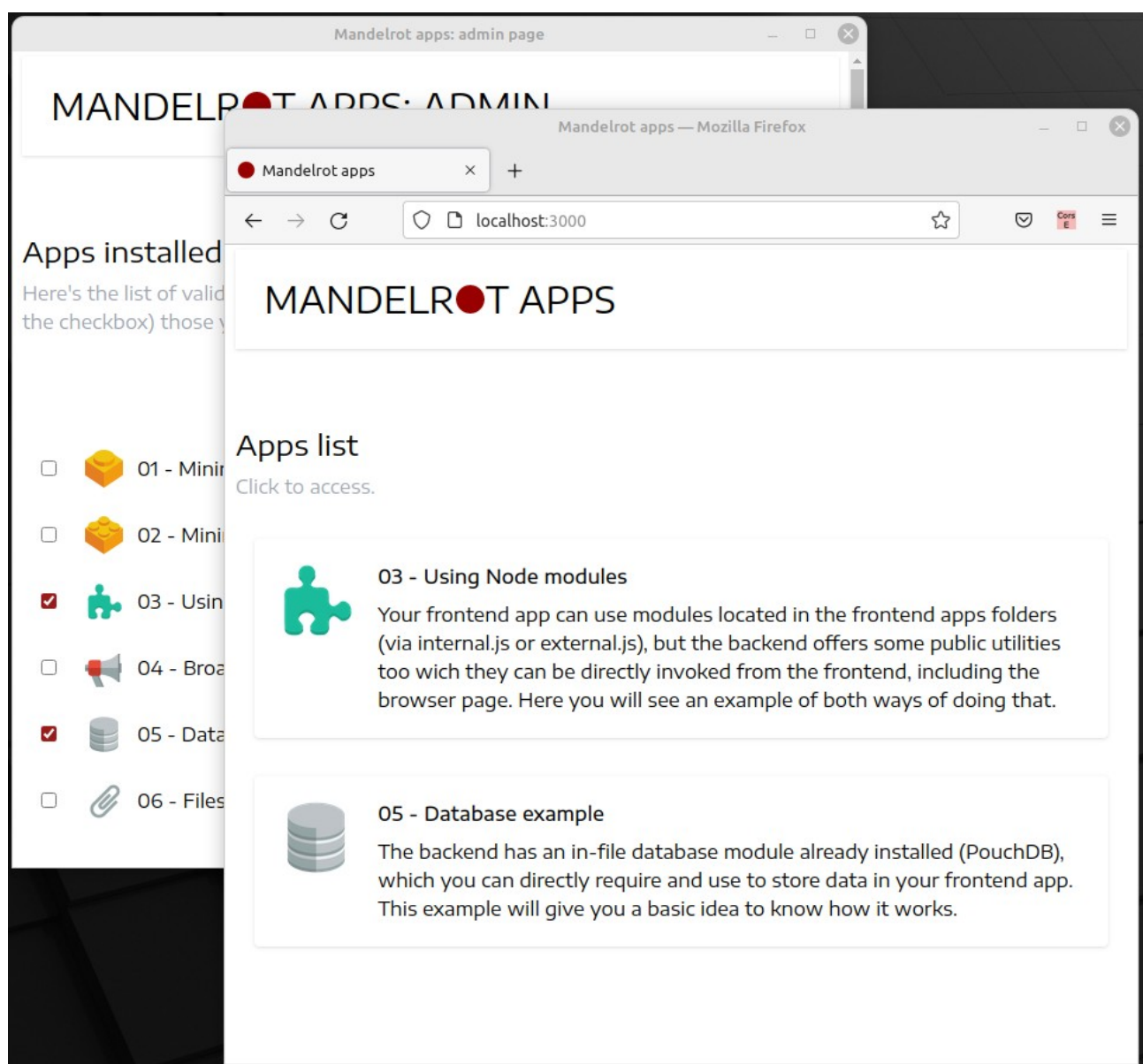


(If your APPS folder is empty, ot at least there's no valid app folder found there, you will obviously see an empty list in the admin window).

So, if you want to connect your frontend app to the system or create a new one, you should have it in the APPS folder. You don't need to close the engine and open it again: just clicking on the “**Refresh**” button the system will check the files structure again.

(Note: it depends on the system, but in the production portable package closing the admin window doesn't stop the engine. You'll have a taskbar icon for that).

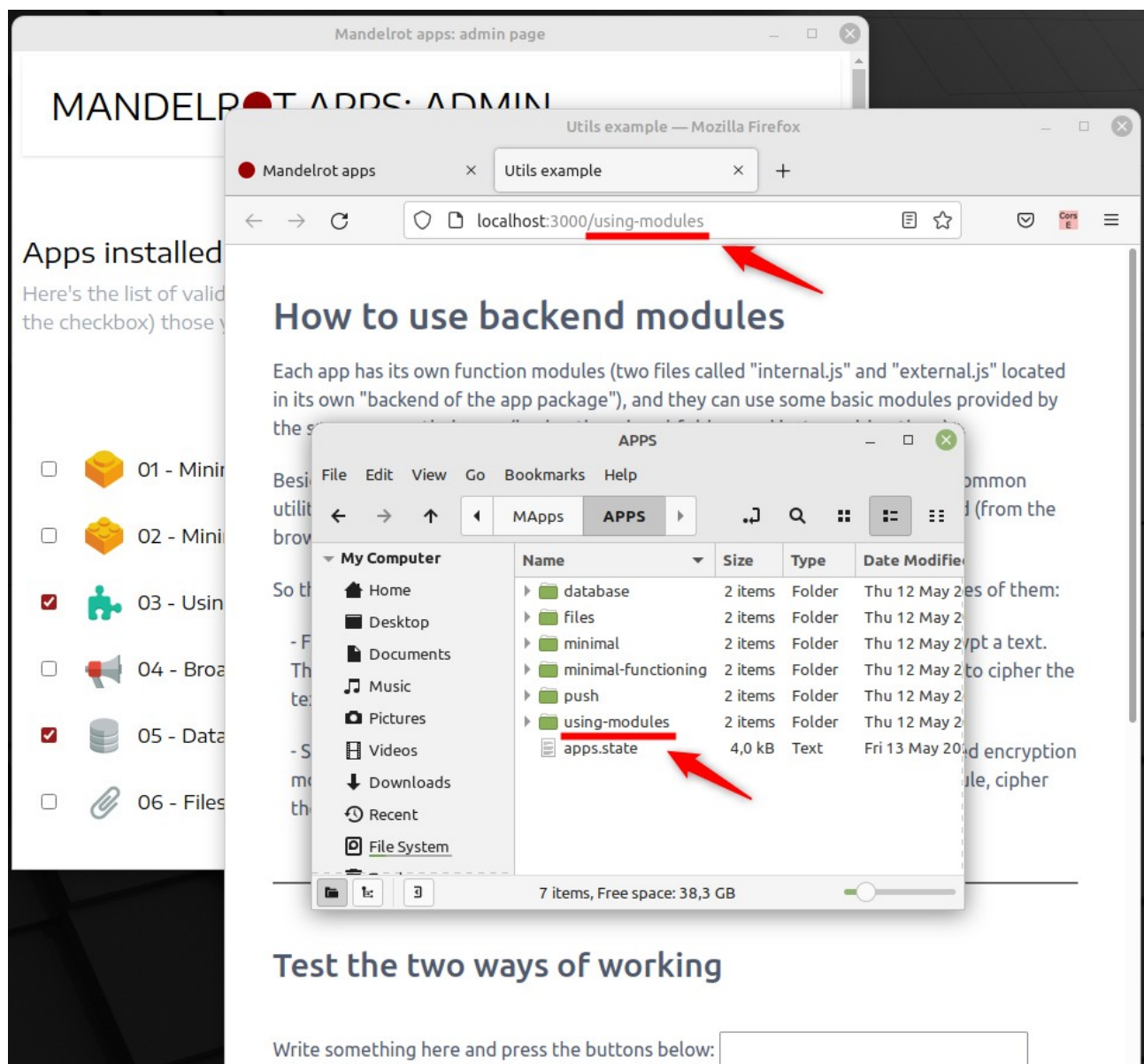
Once your admin manager recognizes the apps in the APPS folder, the next step is make them available for the users. It's as simple as selecting the apps you want to publish, click the “**Confirm**” button, and the users will access the main apps page with a list of the enabled apps in it:



(Note: the url port at the users main webpage is the same as the PORT variable set at the config.js file when compiling. See the admin guide for more info).

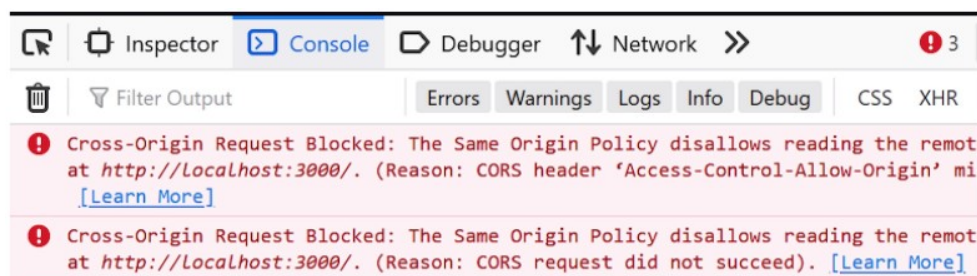
So each time you want to install/uninstall something, just move the copy in or outside the APPS folder and click the Refresh button. The users browser window is connected via socket (it will be push-notified), so the updates will instantly show in the page without refreshing.

When the user wants to access one of the enabled apps they will just click on a list item and the frontend app will be open in a new tab. Please notice that the url of the app is the main page url plus the folder name.



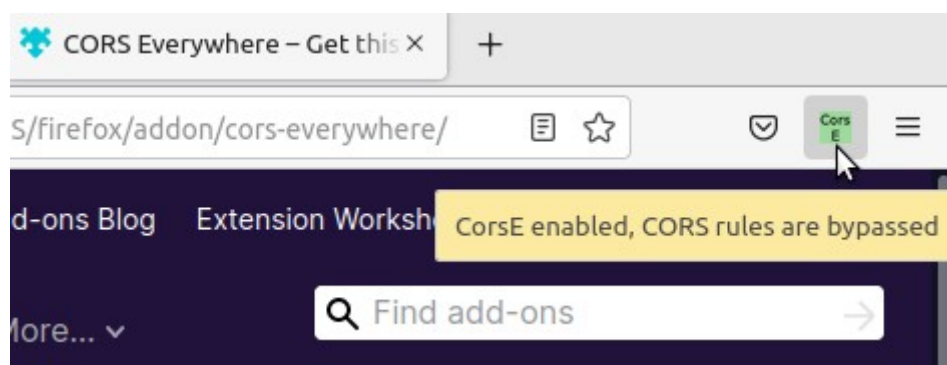
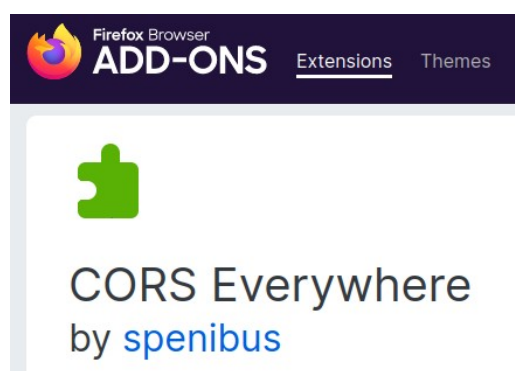
And as we will see later, the folder name will be used in your app routes. Be careful with what you do, when you decide to rename the folder.

CORS conflicts: although you will be able to access the frontend apps main page locally (imitating the user's behavior), if you try to actually use the app logic it will likely not work. If you open the console in the browser app page you will see something like this:



This has nothing to do with this software and it won't happen if you are accessing the page remotely (only if you run the server in your pc). It's a browser security system to prevent hackers to locally access resources that should be remote (if you want to learn more about that here you have [an easy explanation of the topic](#)); every developer hate it but it's something necessary.

Some browsers have extensions that bypass the CORS control, although those are actually security breaches (a safe browser should not allow this at all) and of course the browsers companies don't like it and try to fight it. In the moment of creating this guide I've found a Firefox extension that works in Windows and Linux:

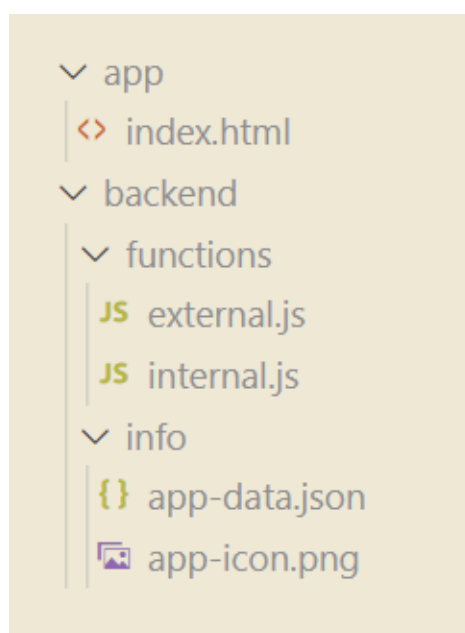


(You can enable/disable the CORS bypass by clicking on the extension button)

But, since the war between the browsers companies and the users is dynamic, maybe it won't work for you or its functionality will be blocked when you read this. In that case you will need to google other current CORS solutions that don't imply having a remote server to avoid local conflicts (which would be the actual solution).

Minimum required app structure

This is the basic file structure the backend will check to validate your app and recognize it as a part of the suite:

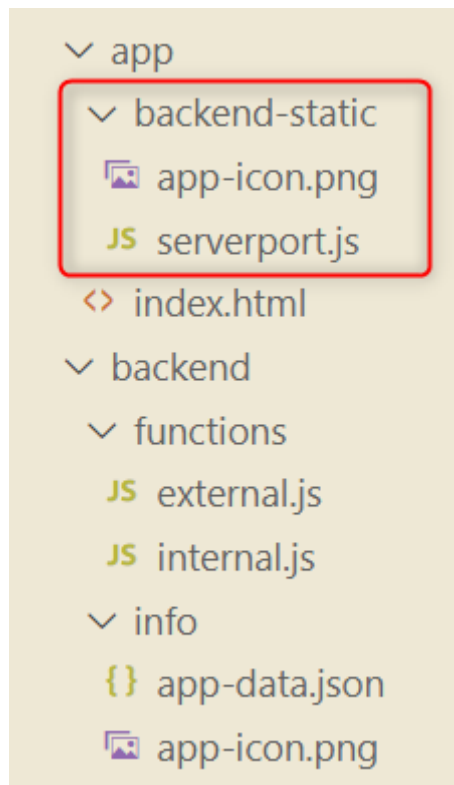


So, if you want to start developing a new front app, the starting point would be creating a folder with your app name and putting these folders and files in it (even if they are empty) just to be detected and accepted by the backend engine.

We have two main subfolders here: an “**app**” folder where your actual frontend app will be, and a “**backend**” folder with the backend functions (the internal module available for your own frontend app and the functions you want to offer other apps) and an info folder we'll get into in a moment.

The “app-icon” is here a .png but it can be anything (.jpg, .svg) as long as its file name to be exactly “app-icon”. If the backend doesn't detect a valid app-icon file a generic icon will be used.

Once the engine detects the app (at starting or clicking on the admin window **Refresh** button) a new folder will be self-generated in your app main subfolder:



This **backend-static** folder will initially contain two elements: a copy of your info/app-icon file (or the generic app icon otherwise) that will be used as your default app favicon, and a **serverport.js** automatic file. We will get into this last one in a moment.

This folder is meant to be used only by the backend, so you shouldn't change anything here. In the future this will be the place where the error log files for this app will be stored, if any.

What is this serverport.js? Here you can see its content:

```
JS serverport.js X
minimal > app > backend-static > JS serverport.js > ...
1  const serverPort = 3000;
```

In the dev time you maybe won't know which port will the admin set for the backend engine, or maybe they will decide to change it and re-compile the backend engine. But when they migrate your app folder to a new engine with

the new port, the new engine will re-generate this file with the current port in use. So if for example you are in the “push” frontend app, in the index.html you can do something like this:

```
<script src="/push/socket.io.min.js"></script>
<script src="/push/backend-static/serverport.js"></script> <!-- file self-generated by the backend -->
<script>
  const socket = io(`http://127.0.0.1:${serverPort}`); // Coming from /backend-static/serverport.js

  // ... etc
</script>
```

You can use this pattern even if you still don’t have the “serverport.js” file, because you know that the backend engine will always create it when needed and take care of setting the current correct value of the `serverPort` variable.

(Note: we’ll later get into how to set the app src links).

As long as you have the basic file structure in your app you can organize your software as you want, creating more elements at will: html-related files or folders, Node modules, database or uploaded files folders and so on. You can even create a full frontend app with any technology or language, and use a full window iframe in the index.html to point to the file or resource where your actual app is. As long as index.html exists, the rest is up to you.

The app-data file

It’s a json with some basic info the backend needs in order to work with your app. This is what it needs to have:

```
{ } app-data.json ×
APPS > minimal > backend > info > { } app-data.json > ...
1 {
2   "appFullName": "01 - Minimal App",
3   "appDescription": "Example of the file structure needed in order to be recognized
4   by the backend as a part of the suite. This app doesn't do anything."
```

And these two fields (plus the icon) will be used in the users main apps page:

MANDELROT APPS

Apps list

[Click to access.](#)



01 - Minimal App

Example of the file structure needed in order to be recognized by the backend as a part of the suite. This app doesn't do anything.

The backend will sort the enabled apps by `appFullName`, and your users will access each app by clicking on the app card (a new tab will be open).

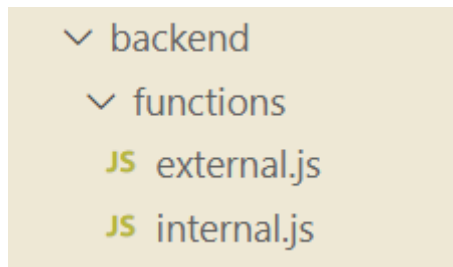
In addition to this, the `app-data` file has an optional field available: `"appRoutingType"`.

```
example-app > backend > info > {} app-data.json > ...
1  {
2    "appFullName": "The example app",
3    "appDescription": "This is just a basic empty app to have a first basic view on how
4    it is structured. See the docs to have all the detailed info.",
5    "appRoutingType": "managedByFrontendApp"
  }
```

It is what it looks like: the server behavior upon http requests may be modified here. More on this coming next.

Your backend function files

In your backend/functions subfolder you will need at least two files: `internal.js` and `external.js`.



These should be regular Node modules, exporting functions that may be invoked from other elements.

```
JS internal.js  ×
minimal > backend > functions > JS internal.js > ...
1  /* These are the functions available to be invoked by the same frontend app */
2
3  const internalFunctions = {}; // You could name the object as you want
4
5  internalFunctions.myExampleFunction = (whateverParamsYouWant) => {
6    // Your Logic here
7    return 'You could return anything and it would be received by the frontend that called
   this function';
8  }
9
10 module.exports = internalFunctions;
11
```

So how does this system work? The design of this system allows any frontend (directly from the browser) to invoke any function in these modules passing it params, and gets returned the function return content. Later on we will see how this works.

If a browser app invokes a function of its own package, the engine server will redirect this petition to the `internal.js` module. If a browser app invokes a function of another package, the server will require the `external.js` module. So the logic of this system implies that you will establish a separation between the functions you want to be “private”, and the functions you will open to other apps (your app “public services”).

Requesting internal resources

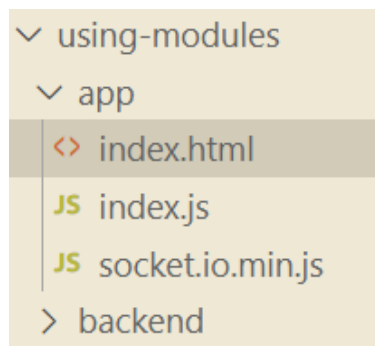
Your app may link any internal resource needed, but it needs to include its own app folder name (the name of the folder in the APPS folder) for it to work.



```
<> index.html X
APPS > using-modules > app > <> index.html > ...
44
45 <script src="/using-modules/socket.io.min.js"></script>
46 <script src="/using-modules/index.js"></script>
47
```

A red arrow points to the `/using-modules/` path in the first script tag.

The server will redirect the request using the “app” subfolder as root. Please have in mind that the route can NOT have two dots together (..) or it will be ignored.



```

  v using-modules
    v app
      <> index.html
      JS index.js
      JS socket.io.min.js
      > backend
```

Regarding routing, as discussed before the `app-data.json` file has an option (`appRoutingType`) that can be included as seen:



```
{ } app-data.json X
example-app > backend > info > { } app-data.json > ...
1 {
2   "appFullName": "The example app",
3   "appDescription": "This is just a basic empty app t
  it is structured. See the docs to have all the deta
4   "appRoutingType": "managedByFrontendApp"
5 }
```

A red box highlights the `"appRoutingType": "managedByFrontendApp"` line, with a red arrow pointing to it.

If your app has this option set, the router will redirect all your requests to `index.html` (but keeping the request url, so your app can handle it internally). This is useful in cases like web-apps made with tools like React or Angular for example.

But you may want to handle some of the routes this way and some others differently (requiring static assets would be the logical use case). Having the “appRoutingType” enabled as shown, you still can directly point to any internal resource by including the substring “static” in it.

Then, with the internal routing enabled, this route would point to index.html:

```
<a href="/my_app_folder/path/to/my/resource.png">click here</a>
```

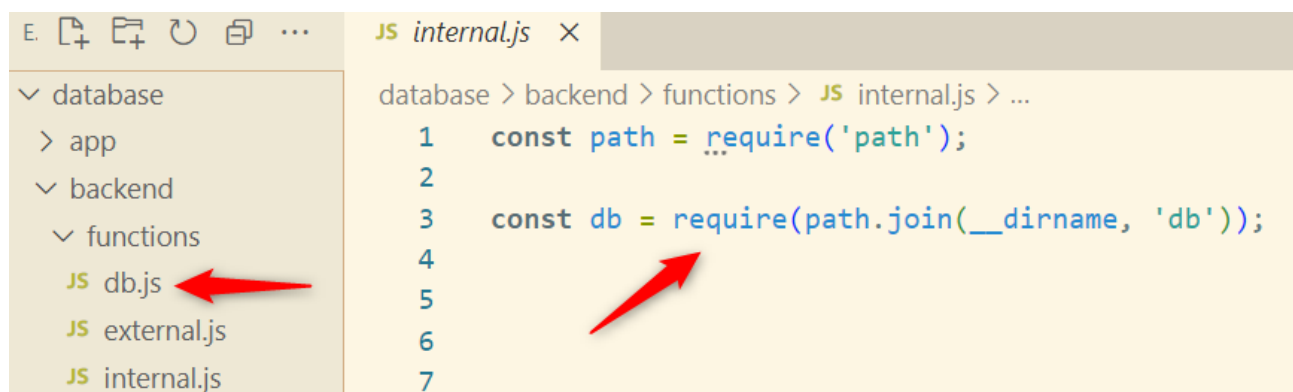
And this would point directly to the resource:

```
<a href="/my_app_folder/path/to/mystatic/resource.png">click here</a>
```

Working with modules

As discussed, these function modules and its children are actually being required by the backend engine and therefore this is Node environment. You can require the basic Node tools like “fs”, “path” and so on (see screenshot below).

On the other hand, your backend functions (within your frontend app folder) may require other files or modules at will so you can implement your own tools and organize things as you want. For example,



So if you want to use any module of your own (as in this example), or some package from Internet, you can download it and store it in your subfolders, and “require” it to use it at will.

What about the packages installed and present at package.json? If these modules are being called from the backend engine, and this backend engine does have a package.json, you should be able to require all the dependencies you see listed there. Right?

In theory yes, and it worked in backend dev mode; but the fact is that once the app is compiled by Electron this stops working. If you go to GitHub and see the package.json file of the backend you will see that there are some dependencies installed that can be required normally by the backend, but they won't work here. But it's ok, I have implemented a way to bypass this.

Before requiring a frontend functions module (internal or external), the backend will see if it contains an import function. And if it's there, it will invoke it to pass an object with instances of some required modules. So the backend requires this Node packages and sends an object to you with this components so you can use them directly.

This is the object from the backend (file: /control/control.js):

```
backendInfo = {
  socketPort: config.server.port,
  modules: {
    // Node packages present in package.json
    PouchDB: require('pouchdb-node'), // In-file database
    fs: require('fs-extra'), // Like the standard Node fs, but with more functions
    cron: require('node-cron'), // Scheduled tasks
    nodemailer: require('nodemailer'), // Email client
    // Other elements
    encryption: require(path.join(__dirname, '..', 'utils', 'public', 'encryption.js')),
    // Important! read the readme in utils/public about the encryption backend module
    socket: io(`http://127.0.0.1:${config.server.PORT}`) // To use in push notifications
  }
};
```

Then, in your frontend app, you can have module variables and (since the import function will be invoked before the actual function required by the frontend, assign values imported by this function to these module variables:

```
let backendInfo;
let fs;
internalFunctions.importBackendInfo = (bInfo) => {
  if (!backendInfo) { backendInfo = bInfo; }
  fs = backendInfo.modules.fs;
  fs.ensureDirSync(path.join(__dirname, 'my-uploaded-files'));
};
```

So in this example you won't use the standard "fs" Node module, but the extended "fs-extra" imported from the backend. As you see this way you can

use some useful dependencies installed in package.json: email management, encryption or socket communication (more on this last one later).

So, in short, we have three ways of importing modules from your backend functions: 1) requiring your local modules (created or downloaded), 2) requiring the standard Node utilities (path, fs), or 3) using the import function that will give you some basic elements you might use too.

And, in addition to all this, there's another extra feature of the MApps system: your frontend, I mean the app living in the browser, can invoke some utilities from the backend too and get back their answer. This is done via socket (we will cover this later), but for now we will say that there's a way for the frontend apps to send "utility messages" to the back for some public functions.

There's a socket event named "utils" that can be triggered by the frontends (more on this next). This event is listened by the backend, and the backend will route the incoming petition to a specific folder: `/utils/public` (check the code on GitHub to see it). This folder contains a few modules by default, but any other module you want to place here (and after compiling) will be available too.



So you can include more public utilities modules here, compile again the production package, and without changing any backend code the "utils" socket channel will automatically give you the possibility to invoke the module you want and within it the function you want, and get its result back.

Please note a special case about this: there's an "encryption.js" module in the utils/public folder that uses the backend `/config/config.js` PASSPHRASE as a base encryption seed. As discussed, you can create your own encryption module next to the "internal/external" modules with just the same functions, and both would work fine.

But watch out! If you encrypt something with a module that uses a seed, you won't be able to decrypt it with another module if it uses a different seed. Be sure you use the same module (backend or frontend) to encrypt things and decrypt them later, or at least that the two different modules use the same seed.

Communications

When you use a socket in your frontend app (browser environment) the information sent must have an specific format. It may be an actual JSON object or a string (as long as it can be later converted to a valid JSON by the backend), with the following structure:

```
const objectToBeSent = {
  app: "yourAppFolderName",
  to: "theTargetedAppFolderName",
  action: "theFunctionYouWantToInvoke",
  data: {
    params: ["the params for", "the function invoked"]
  }
}
```

If “app” and “to” are the same, the backend will require the **internal.js** file within your app folder in APPS to look for the function you want. If the two fields are different the required file will be **external.js** in the “to” folder app.

(Note: calls to non-enabled apps in execution time will be ignored. If the admin disables an app its functions will stop being accesible by the rest of the environment).

And this object (or a string with a JSON valid structure and containing these fields) will be passed to the socket. The name of the socket channel to be used for this purpose is “**msgFromApp**”, here you have a javascript example of the browser app sending the message and waiting for the response:

```
socket.emit('msgFromApp', objectToBeSent, (response) => {
  // Inside this function you would handle the backend answer
  console.log (response);
})
```

The backend will receive the message and will look for the right file (internal or external) and function to execute. If you want to see this in action please check out the example apps at the GitHub repo: in most of them you will find in the “app” subfolder the “**index.html**” file, and there you will see how this is used.

There is no obligatory format to implement responses, but in case the backend finds an error the response will be like this:

```
{ msgError: 'The message to be displayed to the user if needed' }
```

So, just to make everything more standard, my suggestion for you to implement any type of valid response (value, message or whatever) would be:

```
{ msgOk: { foo: bar } } // In this case an object, just as an example
```

This gives you the possibility to implement an standard behavior upon backend responses.



```
JS index.js  X
database > app > JS index.js > retrieveFromBackend
20
21 socket.emit('msgFromApp', message, (response) => {
22   if (response && response.msgError) { alert(response.msgError); }
23   document.getElementById('registryResult').innerText = response.msgOk;
24 });
25
```

You could for example set an standard div element with the same format for all your frontend apps, visible only when response.msgError

If you want to use a utils/public package the object (or string) to send would have the same format, but the content would be slightly different and you would use another socket channel named “utils”:

```
const objectToBeSent = {
  app: "yourAppFolderName",
  to: "theFile.js", // within the /utils/public folder
  action: "theFunctionYouWantToInvoke",
  data: {
    params: ["the params for", "the function invoked"]
  }
}
```

And the socket event:

```
socket.emit('utils', objectToBeSent, (response) => {
  // Inside this function you would handle the backend answer
  console.log (response);
})
```

For example, if you want to encrypt something directly in the frontend but without exposing passwords or any critical data that could be a potential security breach, you could just invoke the encryption module of the public utilities in the backend and it will safely do it for you:

```
JS index.js  X
using-modules > app > JS index.js > sendToUtilsFromInternalJS > encryptedFromInternalJs
7  function sendToUtilsFromBrowser() {
8      let textToEncrypt = inputToEncrypt.value.trim();
9      const encryptedFromBrowser = document.getElementById('encryptedFromBrowser');
10
11      const message = {
12          app: 'using-modules',
13          to: 'encryption.js',
14          action: 'cipher',
15          data: {
16              params: [textToEncrypt]
17          }
18      }
19      socket.emit('utils', message, (response) => {
20          if (response && response.msgError) { alert(response.msgError) }
21          encryptedFromBrowser.innerText = response;
22      });
23  }
24
```

This will make the backend to look for a `utils/public/encryption.js` module, there invoke a function `cipher(textToEncrypt)`, and return its result to the frontend app in the browser.

(Please note: the field “to” can NOT have two dots together (..) or the petition will be ignored).

Now you see how you could go to the GitHub source code, include more `utils/public` utility modules, and without changing any backend code compile it to a new custom production package and instantly have all the new modules available for your backend apps.

Using socket from the function files: as we have discussed before, when you are at internal.js or external.js, you can't directly require Node modules but the backend will use an import function to send you an object that contains instances of its required modules.

The Socket module is an exception to this, because the only way of connecting with the backend (once Electron-compiled) is using the socket instance sent by the backend. This is what the backend sends you:

```
backendInfo = {
  socketPort: config.server.port,
  modules: {
    // Node packages present in package.json
    PouchDB: require('pouchdb-node'), // In-file database
    fs: require('fs-extra'), // Like the standard Node fs, but with more functions
    cron: require('node-cron'), // Scheduled tasks
    nodemailer: require('nodemailer'), // Email client
    // Other elements
    encryption: require(path.join(__dirname, '..', 'utils', 'public', 'encryption.js')),
    // Important! read the readme in utils/public about the encryption backend module
    socket: io(`http://127.0.0.1:${config.server.PORT}`) // To use in push notifications
  }
};
```

And this is how you would use it:

- First you declare a socket module variable.
- Then you have the import function (will be executed before any other, so you can be sure the socket will not be *undefined* when being used). So now your socket is an actual Socket.IO socket instance and you can work with it.
- And then you use the socket in the functions you want.

See it in action here:

```
const myFunctions = {};  
  
let socket; // MODULE VARIABLE  
let backendInfo;  
myFunctions.importBackendInfo = (bInfo) => {  
  if (!backendInfo) { backendInfo = bInfo; }  
  // We bring an instance of the Node socket-io-client module from the backend  
  if (!socket) {  
    socket = backendInfo.modules.socket; // and this is the socket the others functions will use  
  }  
};  
  
myFunctions.triggerPush = () => {  
  const messageToBroadcast = {  
    app: 'push',  
    to: 'apps', // The other possibility would be "user", that would trigger a Logout broadcasted event  
    action: 'not used in this case',  
    data: {  
      params: [],  
      // user: 'the user id' // If to.user this field would be required  
    }  
  }  
  // "msgToBroadcast" is the event being listened by the backend to redirect to everyone connected  
  socket.emit('msgToBroadcast', messageToBroadcast);  
};
```

Now the socket uses by the function modules are restricted for security reasons. These modules will not listen any events (the backend will do it and will then route the requests), and will use it only to emit an specific type of thing: broadcasting.

In this last screencapture you see that the event to emit from the functions modules is “msgToBroadcast”. There are only two types of broadcast messages allowed:

- You can tell the frontend apps to reload the information they have from you (the typical use case is when you have updated something in your DB and you want the users to have push-refresh without reloading their pages).
- Or you can tell the frontend apps to log out an user.

So the sistem works as usual: you emit a message (channel “msgToBroadcast”), the backend will be listening to it and will process it, and then the backend will

broadcast an event in the “broadcast” channel (that the frontend apps in the browser should be listening for).

This is what you would do to send the order to reload data (we are at internal.js or external.js, or any of their children):

```
const objectForReloading= {  
  app: “push”, // Your app folder name, whatever it is  
  to: “apps”,  
  action: “”, // This param will not be used  
  data: {  
    params: [] // Will not be used  
  }  
}
```

And you would send it like this:

```
socket.emit('messageToBroadcast', objectForReloading);
```

Then, your frontend apps will have an listening function like this:

```
socket.on('broadcast', (message) => {  
  // message format: {"sender": "push", "action": "reLoad"}  
  
  // Meaning: the "push" app wants you to refresh its data  
});
```

In the other case scenario (user logout) the two key properties would be “app” and “data → user”:

```
const objectForUserLogout= {  
  app: “push”,  
  to: “user”,  
  action: “”,  
  data: {  
    params: []  
    user: “user_123”, // whatever identification you have set  
  }  
}
```

```
socket.emit('messageToBroadcast', objectForUserLogout);
```

And then your backend would get this:

```
socket.on('broadcast', (message) => {  
  // message format: {"sender":"push","action":"logout", user: 'user_123'}  
  
  // Meaning: the "push" app wants you to logout that specific user  
});
```