[electron-window-manager](https://www.npmjs.com/package/electron-window-manager)

窗体顶端

窗体底端

What it is

A NodeJs module for [**Electron**](http://electron.atom.io/) (Atom Shell, previously) that will help you create, control, manage and connect your application windows very easily.

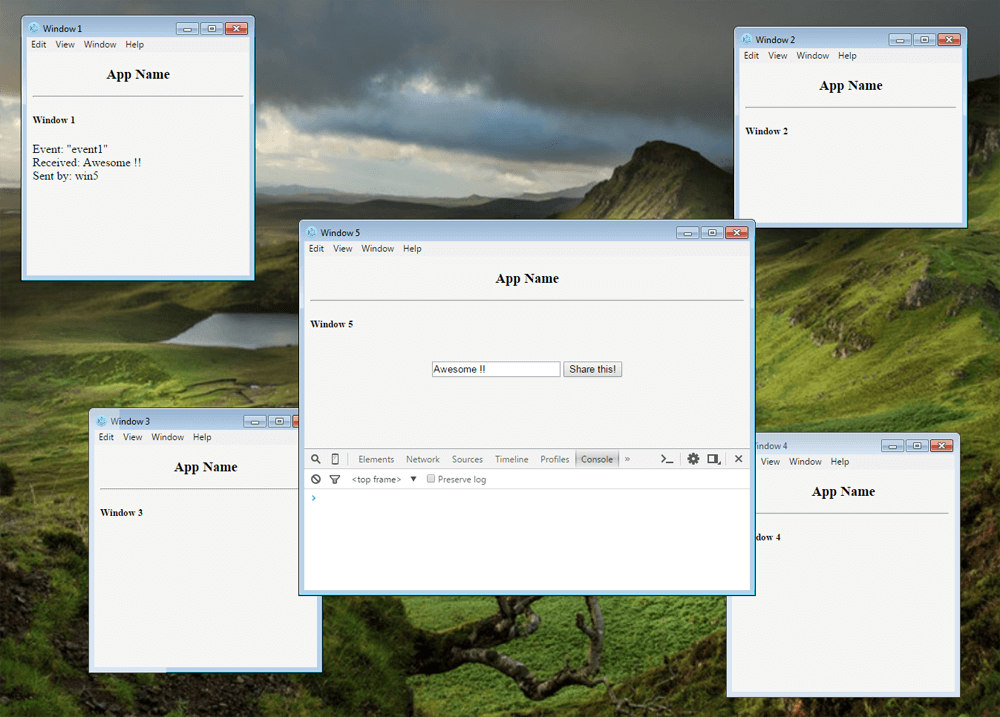
Most of the applications created using Electron are one-window applications. Why only one? Maybe because we as web developers are used to building only inside the browser. But if you are to build a multi-window Electron application then you may want to have a look at this package module.

* [**Features**](https://www.npmjs.com/package/electron-window-manager#features)
* [**Installation**](https://www.npmjs.com/package/electron-window-manager#installation)
* [**How it works**](https://www.npmjs.com/package/electron-window-manager#how-it-works)
* [**Class: WindowManager**](https://www.npmjs.com/package/electron-window-manager#class-windowmanager)
  + [**Method: createNew**](https://www.npmjs.com/package/electron-window-manager#createnew-name-title-url-setuptemplate-setup-showdevtools-)
  + [**.templates**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagertemplates)
  + [**.layouts**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagerlayouts)
  + [**.sharedDate**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagershareddata)
  + [**.bridge**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagerbridge)
  + [**.utils**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagerutils)
* [**Class: Window**](https://www.npmjs.com/package/electron-window-manager#class-window)
* [**Final notes and upcoming updates**](https://www.npmjs.com/package/electron-window-manager#final-notes)

**Features**

Creating a "window" in Electron is done using Electron's native [**"BrowserWindow"**](http://electron.atom.io/docs/v0.36.0/api/browser-window/) module, which is easy to use and very straight forward, until the point when you need to access/control a window from another one, or share values/events between these windows, or use a unified layout/design. That's when this modules come into picture. If you need it I guarantee you will love it :)

* You can easily open, close, move, resize, clone ... etc windows. Of course all of that IS DOABLE through Electron's BrowserWindow, but our module here makes it much easier. It's a kind of wrapper for Electron's module, plus some extra functionalities and tools.
* **You can access any window from within any other window**. That's because all the work is done on the "Main" process, and because each window is given a unique name, so that you could "access" it from anywhere!
* **You can simply**[**share**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagershareddata)**data between the created windows, and even watch for changes in this data**. Cool, right ?!
* You can, very easily, **connect all of the created windows through events; you can trigger an event in window 1 and listen for it in window 2 and 3 and vice versa**. This will help you make your application parts more connected and live. [**More details**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagerbridge)
* For an offline/local multi-window application scenario YOU WILL NEED a way to make these windows share one layout/design, or else you will find yourself building a website in the 90s! This module will help you with this problem. [**Here is how**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagerlayouts)



**Installation**

Simply type the following command in the terminal, inside the application folder

npm install --save electron-window-manager

Then, inside the application main.js *(or whatever you've chosen for your application)*, require the module, like this:

var windowManager = require('electron-window-manager');

Now, **this module can be used in both the "Main" and the "Renderer" processes of Electron**. In the Main process you can use it to create the application's first/main window, and later, in the Renderer process you can create the other windows of the application, or any other way you like.

**On the Main process you can use it like this**:

    const electron = require('electron');

    const app = electron.app;

    const windowManager = require('electron-window-manager');

    // When the application is ready

    app.on('ready', function(){

        windowManager.init( ... );

     // Open a window

        windowManager.open('home', 'Welcome ...', '/home.html');

    });

**And in the Renderer process** (inside the created window), you can use it like this:

<script>

    var remote = require('remote');

    var windowManager = remote.require('electron-window-manager');

    // Create a new window

    var win2 = windowManager.createNew('win2', 'Windows #2');

    win2.loadURL('/win2.html');

    win2.onReady( ... );

    win2.open();

</script>

Please go ahead and check out the module code and see for yourself how it all works, there's no magic involved, but it's consistent and easy to read. Just have a look at the code and you are good to go.

**How it works**

* **This module is basically a wrapper for Electron's BrowserWindow** module, so it's definitely a good start to check out Electron's [**documentation for BrowserWindow**](http://electron.atom.io/docs/v0.36.0/api/browser-window/).
* At its core, this module **works as a holder for the BrowserWindow instances created, each instance is stored by a unique name**. So you could later on call any window by name.
* **Each BrowserWindow instance is created inside the**[**"Window"**](https://www.npmjs.com/package/electron-window-manager#class-window)**class**, which in a way extends and adds more functionalities to it.
* When creating a new window you are required to provide a basic SETUP for it, stuff like the width, height, title, url ... etc. The setup options BrowserWindow offers are plenty, and this module adds couple more. **If some/many of the windows share the same setup, you can simply create a**[**"Setup Template"**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagertemplates)**and pass this template (by name) to the window in the making to apply the setup**.
* After a window is created you can change its content, resize it, move it, full screen it, ... pretty much anything you may need, even you can execute JS code on it, from another window ... cool ha!
* A development-mode is on by default, until you finish developing the application and set it off. **The development mode will make debugging a bit easier; you will be able to reload ANY window just by pressing CTRL + R, also you will be able to toggle the development tools Chrome offers just by pressing CTRL + F12**, for any window and without any pre-configuration. The developer tools can be turned on by default for a window just by passing *"showDevTools: true"* in the window setup, or even by calling the method *showDevTools()* on the window object.

API

**Class: WindowManager**

The windowManager class is the interface of the module, through which you can access the created windows, and of course create new ones. Also through it you can access everything the module uses internally, like utilities and other modules used in the process.

**windowManager.windows**

This property is where the created Window instances are stored, each by its name

**windowManager.config**

The basic configuration of the module

**init( config )**

This method initiates the module so that you could use it. It takes one *optional* argument, which is the configuration. You only need to initiate the module once, and that's on the "Main" process.

* **config**
  + **appBase** (string): The path to the application base, whether it's local or remote. **By default the local path to the application directory is used**.
  + **devMode** (boolean): Sets whether the development mode is on or off, it's on by default.
  + **layouts** (object): As mentioned, this module will help you use layout(s) for the content of your windows. This option is where you can pass a list of the layout file(s), each under a specific name. Here's an example:

    windowManager.init({

        'layouts': {

            'default': '/layouts/default.html',  // The "/" at the start will be replaced with the 'appBase' value

            'secondary': '/layouts/secondary.html'

        },

        'defaultLayout': 'secondary'

    });

This of course will be effective only when the app is local, or at least the target file is stored locally. [**More details on the "Layouts" feature**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagerlayouts)

* + **defaultLayout** (string): The name of the default layout.
  + **defaultSetupTemplate** (string): The name of the setup template that you want as the default for all of the application windows. "Setup Templates" is a way of presetting shared setup properties for the application windows. [**More details**](https://www.npmjs.com/package/electron-window-manager#class-windowmanagertemplates)
  + **defaultWindowTitle** The default title for the windows; will be used if a window title wasn't specified, and if the setup template didn't provide a title either! Here you can simply use the name of the application.
  + **windowsTitlePrefix** (string): Each window can have its own title, this value will allow you to set a prefix for this title, for all the windows, ... maybe the application name!
  + **onLoadFailure** (function): Here you can set what happens whenever the target URL of a window isn't accessible, meaning that the loading has failed, for whatever reason. By default a small message will be be displayed instead of the page (it basically loads the *loadFailure.html* file from the module directory). Here's an example:

    windowManager.init({

        'onLoadFailure': function(window){

            window.loadURL('/404.html');

            // -- or ---

            window.write('<h3> Cannot load the requested page! </h3>');

        }

    });

**setDefaultSetup( setup )**

This method sets the default setup for the application windows, it basically creates a new setup template with the name "default", and marks it as the default. Example:

    windowManager.setDefaultSetup({'width': 600, 'height': 450, 'position': 'right'});

**When creating a new window this setup will be used automatically, you can override this by passing FALSE as the setup template name.**

**createNew( name, title, url, setupTemplate, setup, showDevTools )**

This method, as the name suggests, creates a new window, it will create and return a new instance of the class [**Window**](https://www.npmjs.com/package/electron-window-manager#class-window). **All of the listed arguments up there are optional by the way**, even the url; you can omit it now and set it later.

* **name** (string): The name of the new window, if omitted a serialized name will be used instead, like "window\_1" ~> "window\_2" ~> ...
* **title** (string): The window title. If omitted the setup template 'title' property will be used, and if this property wasn't provided the config.defaultWindowTitle will be used instead. If all of that isn't set the document title will be used.
* **url** (string): The targeted URL for the window, it could be a local file (file:// ... .html), or a URL (http:// ...). If the passed value here starts with "/" the value will be preceded by the "appBase" (which is the path to the application base, set in the config), also you can use {appBase} inside the passed value and it will be replaces with the application base path.
* **setupTemplate** (string) The name of setup template you want applied to this new window. "Setup Templates" is a way of sharing preset setup properties with more than one window.
* **setup** (object|string) [optional] The new window setup. **The full options list is available**[**here**](http://electron.atom.io/docs/v0.36.0/api/browser-window/).This module offers couple more options to use in the mix, but we will get to that later. BTW, as a shortcut you can pass the new window dimensions like this "300x200", where 300 is the width and 200 is the height!
* **showDevTools** (boolean) Whether to show the developer tools offered by Chrome or not, for debugging. False by default.

Here's an example:

    var homeWindow = windowManager.createNew('home', 'Welcome ...', '/pages/home.html', false, {

        'width': 600,

        'height': 450,

        'position': 'topLeft',

        'layout': 'simple',

        'showDevTools': true,

        'resizable': true

    });

    homeWindow.registerShortcut('CmdOrCtrl+N', function(window){

        window.minimize();

        windowManager.open( ... );

    }).open();

As mentioned, beside the the setup options BrowserWindow offers we offer couple more:

* **layout** (string) The name of the layout you want the window's content to be displayed inside.
* **position** (string|array) This setup option sets the position of the window on the screen, **you can pass the x & y coordinates as an array (ex: [300, 200])**, or simply pass the position name, the available positions are: *top, right, bottom, left, topRight, topLeft, bottomRight, bottomLeft*. The default position by the way is "center".

    var win = windowManager.createWindow(false, false, false, false, {'position': 'bottomRight'});

* **onLoadFailure** (function) This callback will be triggered whenever the URL loading fails.

    var win = windowManager.open(false, false, false, false, {

        'onLoadFailure': function(window){

            window.close();

        }

    })

If you don't set this a default callback will handle it, by showing a global "Not available" message page, this global callback can be set in windowManager.init()

* **showDevTools** (boolean) Whether to show the dev tools or not

**open( name, title, url, setupTemplate, setup, showDevTools )**

This is the same as the createNew method, except that it opens the window directly. Returns a [**Window Object**](https://www.npmjs.com/package/electron-window-manager#class-window) on successfully opening a window.

**importList( file )**

Using this method you can create more than one window instance, with the setup information retrieved from a JSON file. You will use it like this:

    // windows.json

    {

        "home": { "title": "Home", "url": "http:// ...", "setup": { ... } },

        "about": { "title": "About", ... }

    }

    // Import the window list

    windowManager.importList('windows.json');

    // Open a window, by name

    windowManager.get('home').open();

**clone( name )**

Creates a clone of the specified window and returns the Window class instance

**get( name )**

Returns the window instance of the specified window

    var win = windowManager.get('home');

    win.resize(300, 200).restore();

**getById( id )**

Returns the window instance of the specified window by BrowserWindow instance's id attribute

    var win = windowManager.getById(1);

    win.close();

**getCurrent()**

Returns the Window instance of the currently-under-focus window

    windowManager.getCurrent().close();

**close( name )**

Closes a window, by its name

**closeCurrent()**

Closes the current window

**destroy( name )**

Destroys the window instance, by its name

**closeAll()**

Closes all the windows created by this module

**closeAllExcept( name )**

Closes all the window, except for one, the one you pass its name

**maximize( [ name ] )**

Maximizes a window. If name is specified it will target the named window. If left out it will target the currently focused window. If the window is already maximized it will restore.

    // Maximize focused window. Works well for window button functionality.

    windowManager.maximize();

    // Target window named 'help'

    windowManager.maximize('help');

**minimize( [ name ] )**

Minimizes a window. If name is specified it will target the named window. If left out it will target the currently focused window.

    // Minimize focused window. Works well for window button functionality.

    windowManager.minimize();

    // Target window named 'help'

    windowManager.minimize('help');

**restore( name )**

Restores a minimized window, by name

**devModeChoice( whenDevMode, whenNotDevMode )**

This method simply takes two values, the first is the one that goes when the development mode is on and the other is when it's off, and according to whether it's on or off, the corresponding value will be returned

    var api = windowManager.devModeChoice({'key': ... }, {'key': ...});

**Class: windowManager.templates**

When creating a new window you will need to provide a basic setup; things like the width, height and the window URL (the targeted content). The available setup options are plenty, and in most cases you will find yourself repeating it with each window you create. **The "Setup Templates" feature will help you make presets of the setup properties you use and name them, and later when creating a new window you will just pass the template name and the associated setup will be applied.**

    windowManager.templates.set('small', {

        'width': 600,

        'height': 350,

        'resizable': true,

        'layout': 'classy',

        'showDevTools': true,

        'title': 'App name, for small windows!', // Yeah, even the window title!

        'onLoadFailure': function(){ ... },

        'menu': { // Sets the window menu. Set to null for a window without a menu!

            label: 'File',

            submenu: [

                {

                    label: 'New',

                    accelerator: 'CmdOrCtrl+N',

                    role: 'new',

                    click: function(){ ... }

                },

                {

                    label: 'Open',

                    accelerator: 'CmdOrCtrl+O',

                    role: 'open',

                    click: function(){ ... }

                }

            ]

        }

    });

    windowManager.open(false, false, 'welcom.html', 'small');

    windowManager.open(false, false, 'byebye.html', 'small');

You can set a default template for all the windows to inherit

    windowManager.init({

        'defaultSetupTemplate': 'small'

    });

    windowManager.templates.set('small', { ... });

You can override this option for a specific window by passing FALSE as the setupTemplate name.

    windowManager.open('home', 'Welcome', '/pages/welcome.html', FALSE);

**Pass null to explicitly define the template in the next parameter!**

    windowManager.open('home', 'Welcome', '/pages/welcome.html', null, { ... });

**windowManager.templates.set( name, setup )**

Use this method to create a new template, you provide a name and the preferred setup, and later you can use that name when you are creating a new window, to apply the specified setup.

    windowManager.templates.set('big', {

        'width': 1000,

        'height': 600

    });

**windowManager.templates.get( name )**

Fetches a setup template by name

**windowManager.templates.modify ( name, setup )**

Use this method to modify the setup of a specific template, to override one or more of its properties

    windowManager.templates.modify('big', {'height': 650});

**windowManager.templates.getProperty ( name, property )**

Returns the value of a specific property of the given template

    windowManager.templates.getProperty('big', 'width');

**Class: windowManager.layouts**

A nice feature this module offer is "Layouts". Through this feature you can set a layout/theme/design for your application. If you are working on a multi-window app this feature will be very handy. This feature doesn't offer much right now, but I will put more focus on in the next releases. **Here's how it works: you create a layout file, with all the assets and code you want, when you create a new window the content (HTML) of this window will be embeded/included within the layout code.** Thus you won't duplicate your code with each window; you will just include the window content withing a ready layout.

    // layout.html

    <!doctype html>

    <html lang="en">

    <head>

        ....

    </head>

    <body>

        {{content}} <!--This will be replaced with the window content-->

        <script>

            ...

        </script>

    </body>

    </html>

    // welcome.html (the window content)

    <h3>Welcome ...</h3>

Of course you can create more than one layout, and when creating a new window you chose which layout you want the window content included in. And of course you can set a default layout for all of the application windows.

    var win = windowManager.createNew('home', 'Welcome ... ', '/pages/welcome.html', false, {'layout': 'simple'});

    // or

    win.useLayout('simple');

To set a default layout for the whole application you simple pass its name in the initiation config

    windowManager.init({

        'defaultLayout': 'simple'

    })

To override this option for a specific window you will need to pass FALSE for the property "layout" when creating the window

**windowManager.layouts.add( name, path )**

Adds a new layout, you need to provide the path to the layout file, and a name that represents this layout, so that you could use it when creating a new window.

    windowManager.layouts.add('default', '/layouts/default.html'); // The "/" at the beginning = {appBase}

**windowManager.layouts.get( name )**

Fetches a layout by name

**class: windowManager.sharedData**

This class offers a simple way of sharing data between windows, you can simply set a value on window #1 and get it on window #2

**windowManager.sharedData.data**

Is where the data itself is stored

**windowManager.sharedData.set( key, value )**

Stores a value by a key name

    windowManager.sharedData.set('user', {'name': ' ... ', 'email': ' ... '});

**windowManager.sharedData.fetch( key )**

Fetches a value, by key name

    windowManager.sharedData.fetch('user');

**windowManager.sharedData.watch( prop, callback )**

You can use this method to watch for changes in the saved data

* **prop** Is the key you want to watch
* **callback** Is the callback that will be triggered whenever the value of this property gets changed

    windowManager.sharedData.watch('user', function(prop, action, newValue, oldValue){

        console.log('The property: ', prop, ' was:', action, ' to: ', newValue, ' from: ', oldValue);

    });

This feature is available using [**WatchJS**](https://github.com/melanke/Watch.JS), please visit the module docs for further details. Also, you can access WatchJS itself in case you needed the whole API like this:

    windowManager.sharedData.watcher; // The WatchJS module

    var watcher = windowManager.sharedData.watcher;

    watcher.unwatch( ... );

**class: windowManager.bridge**

This is a simple feature to help you make your app more alive and connected windows-wise. You will be able to emit and listen to events inside the created windows.

Here's an example:

    // On window "home"

    let handler = windowManager.bridge.on('new\_chat\_message', function(event){

        ...

    });

    // On window "chats"

    windowManager.bridge.emit('new\_chat\_message', {'message': ' ... '});

    // On window "home"

    windowManager.bridge.removeListener('new\_chat\_message', handler);

**windowManager.bridge.emit( event, data, target )**

This method emits an event to whatever page listening for it

* **event** The name of the event that will be emitted
* **data** [optional] Any extra data you need to broadcast along with the event
* **target** [optional] The name of the targeted. In case you are targeting a specific window.

**windowManager.bridge.on( event, callback )**

This method adds a listener for a specific event, and whenever this event is triggered the callbackwill be called, and return the handler added into the event listeners array.

* **event** The name of the event the window will be watching for
* **callback** The callback to call when the event is emitted. This call back will be passed 1 parameters with the following properties:
  + **event** The event name
  + **target** The name of the targeted window, if specified
  + **emittedBy** The name of the window that emitted the event

This feature is basically a wrapper for NodeJs native EventEmitter class, which is used heavily almost every where inside Electron itself. Check it the [**docs**](https://nodejs.org/api/events.html#events_class_eventemitter) for extra knowledge about the subject. You also can access the EventEmitter used by this module simply by calling windowManager.eventEmitter

    windowManager.eventEmitter.addListener( ... );

**windowManager.bridge.addListener( event, callback )**

This method is the alias of windowManager.brider.on

**windowManager.bridge.removeListener (event, handler )**

This method remove the listener returned by windowManager.bridge.on or windowManager.bridge.addListener

* **event** The event name
* **handler** the handler returned by windowManager.bridge.on or windowManager.bridge.addListener

This feature is basically a wrapper for NodeJs native EventEmitter class, Check it the [**docs**](https://nodejs.org/api/events.html#events_class_eventemitter) for extra knowledge about the subject.

    windowManager.eventEmitter.addListener( ... );

**Class: windowManager.utils**

This object holds a couple of utility method, for module internal use, and for you if you need it. I will likely add more methods to this class later.

**windowManager.utils.getAppLocalPath()**

Returns the path to the application directory.

**windowManager.utils.readyURL( url )**

It readies the given URL for use with in the module, basically it replaces the "{appBase}" with the path to the application directory.

**windowManager.utils.resolvePosition( setup )**

This method takes a position name and returns the corresponding x & y coordinates, the accepted values are: "top", "bottom", "right", "left", "topRight", "topLeft", "bottomRight", "bottomLeft" and "center".

* **setup** (object) The window setup object, inside which the "*position*", "*width*" and "*height*" properties must be present.

You probably wont be needing this method, but it's here just in case.

Class: Window

The Window class is basically the whole thing, [**windowManager**](https://www.npmjs.com/package/electron-window-manager#class-windowmanager) is merely an access point for its instances. Whenever you use windowManager.createNew( ... ) or windowManager.open( ... ) you are creating a new instance of Window.

    var window = new Window( name, title, url, setupTemplate, setup, showDevTools );

**But don't try to use the above code, Window isn't available in your app scope**, use windowManager.createNew/open instead, and as you can see the arguments are the same in the 3 cases, **check out**[**windowManager.createNew**](https://www.npmjs.com/package/electron-window-manager#createnew-name-title-url-setuptemplate-setup-showdevtools-)**for more info on the arguments**.

**Window.name**

Stores the widnow name.

**Window.setup**

Stores the window setup object.

**Window.object**

Stores the BrowserWindow instance created.

**Window.set( prop, value )**

Updates the window setup. You can either provide a property-value pair or pass an object to override the current setup.

    win.set('width', 300);

    win.set({'width': 300, 'height': 250});

**create( url )**

Creates the browserwindow instance.

    var win1 = windowManager.createNew(false, false, 'win1.html');

    win1.create();

    win1.object.on(...);

    win1.open();

    // or

    var win2 = windowManager.createNew();

    win2.create('win2.html');

    win2.open();

**open( url )**

Opens/shows the created window.

    var win1 = windowManager.createNew(false, false, 'win1.html');

    win1.open();

    // or

    var win2 = windowManager.createNew();

    win2.open('win2.html');

**focus()**

Makes the window under focus.

**useLayout( name )**

Sets the layout to use in the window, by name.

    var window = window.createNew( ... );

    window.useLayout('classy');

    window.open();

**setURL( url )**

Sets the target URL for the window, to open a URL *after* the window is open use loadURL().

**applySetupTemplate( name )**

Sets the setup template to use, by name.

    var window = window.createNew( ... );

    window.applySetupTemplate('big');

    window.open();

**loadURL( url, options )**

Sets the content of the new window; the url it will open. Same as with [**BrowserWindow**](http://electron.atom.io/docs/v0.36.0/api/browser-window/#win-loadurl-url-options) you can use both local and remote targets.

    var win = windowManager.createNew();

    win.loadURL('file://' + \_\_dirname + 'index.html');

    // or

    win.loadURL('http://google.com');

The same way you would open a url using any browser. Now, to make things easier you can set the base path to the application in the config (while initiating the module) and use this path in any URL-value you path to the module, appBase, or by simply starting the value with "/".

    win.loadURL('/pages/index.html');

    // or

    win.loadURL(' ... {appBase} ... ');

**html( code, options )**

It simply sets the HTML code of the window, instead of loading a url.

    win.html('<h3> Electron is AWESOME </h3>');

**down()**

It simply takes the page down! It will trigger the onLoadFailure callback, which by default will force display a "Not available" message page. This method is called whenever the target url of the window isn't available, instead of displaying a blank page.

**content()**

Returns BrowserWindow's [**webContents**](http://electron.atom.io/docs/v0.36.0/api/web-contents/) object for the window.

    win.content().on('did-fail-load', function(){ ... });

    win.content().downloadURL( ... )

    win.content().reload()

    win.content().print()

**reload( ignoreCache )**

Reloads the URL of the window, if TRUE is passed the page will be reloaded with the cache ignored.

    win.reload(); // With cache

    win.reload(true); // Without cache

**currentURL()**

Returns the URL open inside the window.

**onReady( withTheDomReady, callback )**

Registers a callback that triggers when the page is ready. If you pass TRUE for the withTheDomReadyargument the callback will trigger only when the DOM is ready, and not before.

    win.onReady(true, function(window){

        window.resize(600);

    });

**execute( code )**

Executes JavaScript code on the window content.

    win.execute(' alert(" Hi! ") ');

**goBack()**

Goes back to the previous page, Electron is a browser after all!

**close()**

Closes the window.

**maximize()**

Maximizes the window. Restores if already maximized.

**minimize()**

Minimizes the window.

**restore()**

Restores the window back in focus.

**toFullScreen()**

Takes the window to fullscreen.

**toggleDevTools( detached )**

Toggles the developer tools. **By default, and when the devMode is on you can open the developer tools by pressing CTRL + F12 on any window.**

* **detached** (boolean) [optioanl] Whether to open the devTools in a separate window or not

**registerShortcut( accelerator, callback )**

Registers a keyboard shortcut on the window

    win.registerShortcut('CTRL+N', function(){

        windowManager.open( ... );

    });

This feature is available thanks to the [**electron-localshortcut**](https://github.com/parro-it/electron-localshortcut) module. Here's more details on the [**shortcuts codes**](https://github.com/atom/electron/blob/master/docs/api/accelerator.md). The module itself can be access through windowManager.shortcuts in case you wanted to use more of it, to globally-register a new shortcut or something.

    windowManager.shortcuts.unregisterAll();

**move( x, y )**

Moves the window to a specific x (and/or) y coordinates. You can also provide a position "name" and it will be resolved to the correct position according to the screen size and the window dimensions. The available position names are *top, right, bottom, left + topRight, topLeft, bottomRight, bottomLeft*

    win.move(300, 200);

    win.move('topLeft');

**resize( width, height )**

Resizes the window to a specific width and/or height

    win.resize(1000); // Only set the width

    win.resize(800, 400);

**Final notes**

* The module is definitely still under development, and I am always updating and fixing the code.
* You may find this module a bit redundant, but if you use Electron for complex multi-window projects you will probably need it.
* I actually spent more time on writing this documentation than I spent on writing the module itself, so **PLEASE if you have any comments or suggestions of any kind write me or open an issue**.
* **The development of this module WILL be continued**, so feel safe to use it and know I got your back ;)
* **The next releases** will ...
  + Put more focus on the **Layouts** feature, I will probably use [**Handlebars**](http://handlebarsjs.com/) to offer templating capabilities.
  + Offer more debugging tools and feature.
  + Better documentation and code examples

And yeah, THANKS GITHUB FOR ELECTRON, IT'S A DREAM CAME TRUE.

The MIT License (MIT)

Copyright (c) 2015 Ahmed Zain [**tamkeenlms@gmail.com**](mailto:tamkeenlms@gmail.com)