Dashkiosk

Release 2.7.3

October 04, 2017

Contents

Dashkiosk is a solution to manage dashboards on multiple screens. It comes in four parts:

1. A **server** will manage the screens by sending them which URL they should display in realtime. A web interface enables the administrator to configure groups of dashboards as well as their associations with available displays.
2. A **receiver** runs in a browser attached to each screen. On start, it contacts the server and waits for it to tell which URL to display.
3. An **Android application** provides a simple fullscreen webview to display the receiver.
4. A **Chromecast custom receiver** which will run the regular receiver if you want to display dashboards using Google Chromecast devices.

The Android application and the Chromecast receiver are optional components. Any device able to display a fullscreen web page should work.

A live installation, reset every hour, is publicly available:

- administration panel
- receiver

You can also quickly pull a ready-to-use version with Docker (you may want to replace `latest` by a stable tag, like `2.6.1`):

```
$ docker run -d -p 8080:8080 \
             -v /var/lib/dashkiosk/database:/database \
             vincentbernat/dashkiosk:latest
```

Here is a demonstration video:

To contribute, use GitHub.

CHAPTER 1

Installation

The server is a Node.js application providing a rendez-vous point for all displays to subscribe using a Websocket protocol as well as an administration page to manage all displays, group them and tell them which URL to display.

Server and receiver

To install it, you need to execute the following step:

1. Grab the latest tarball for Dashkiosk from GitHub.
2. Install Node.js and optionally npm. Currently, *Dashkiosk* works with *Node.js* 0.12.x or more recent. If the version available in your distribution is not up-to-date, have a look at how to install Node.js via the package manager before trying to build from the sources.
3. Install *bower* and *grunt*, two package managers for Javascript with the following command:

   ```
   $ npm install -g bower grunt-cli
   ```

4. Unpack *Dashkiosk* in the directory of your choice and go into that directory.
5. Install the appropriate dependencies with the following commands:

   ```
   $ npm install
   ```

6. Build the final version of *Dashkiosk* with the following command:

   ```
   $ grunt
   ```

   If you get an error about `jpegtran`, install `jpegtran`, `optipng` and `gifsicle` from your distribution. For example, on Debian:

   ```
   $ sudo apt-get install jpegtran optipng gifsicle
   ```

7. Upon success, you will get a `dist` directory that you can put on some server. It includes both the receiver and the server part. Then:

   ```
   $ cd dist && npm install --production
   ```

8. If you want to use the Android application, you still need to build it and install it. See *Android application*.

If you get an error while compiling mDNS extension on Linux, ensure you have the appropriate development package for Avahi. Specifically, on Debian, you need `libavahi-compat-libdnssd-dev`.

Database

Dashkiosk stores its data inside some database. By default, it uses SQLite. If you prefer to use another database, this is quite easy. We will use PostgreSQL but this should be easy to transpose to another database supported by Sequelize.js, the ORM used in *Dashkiosk*. The databases currently supported are:

- MySQL,
- MariaDB,
- SQLite, and
- PostgreSQL.

Here are the steps:

1. Create a dedicated user inside your RDBMS. For *PostgreSQL*, this is done as the `postgres` user with the following command:

   ```
   $ createuser -P dashkiosk
   Enter password for new role:
   Enter it again:
   Shall the new role be a superuser? (y/n) n
   Shall the new role be allowed to create databases? (y/n) n
   Shall the new role be allowed to create more new roles? (y/n) n
   ```

2. Create an empty database. For *PostgreSQL*, this is also done as the `postgres` user:

   ```
   $ createdb -O dashkiosk dashkiosk
   ```

The database will be populated automatically when running *Dashkiosk* for the first time.

Unassigned dashboard

By default, displays are put in a group with a dashboard with cycling images. You can add more images in `app/images/unassigned` if you want. Then, rebuild with `grunt`.

CHAPTER 2

Upgrade

To upgrade, you can use the exact same instructions that you used for installation. You can choose to do an in-place upgrade if you want to:

```
$ npm install
$ grunt
```

If you kept the default SQLite database, be sure to save it (it's in `dist/db/`) before upgrading and restore it after upgrade. The `dist/` directory where *Dashkiosk* is built is wiped out on upgrade.

CHAPTER 3

Configuration

Before running the server, there are some options you may want to tune. Those options can be specified either on the command-line or in a configuration file (that should be specified on the command-line).

Available options

Here are the three more important options:

`configuration` This option allows to specify a configuration file in JSON format.

`environment` This option sets the environment to use. By default, the `development` environment is used. Unless you want to debug *Dashkiosk*, this is not what you want. You can use any other keyword. Use `production` if you don't know.

`port` The port to listen to. Quite important.

If you want to be able to use Chromecast devices, you also need to set:

`chromecast.enabled` Enable Chromecast support. Disabled by default.

`chromecast.receiver` The URL to the receiver. This is used to tell Chromecast devices where to find the receiver. This option is mandatory as there is no reliable way to guess it. You should put the URL to access Dashkiosk and ends it with `/receiver`. For example, put something like `http://dashkiosk.example.com/receiver`. This needs to be a fully qualified URL (including the protocol part `http://` or `https://`).

If the reciever is on a server with only internal DNS records use the IP address of the server rather than URL. Chromecasts are hard coded to only use Google's DNS servers (`8.8.8.8` and `8.8.4.4`) for lookups, rather than those given by DHCP.

The Chromecast custom receiver is hosted on GitHub. If you want to modify it or to host it yourself, you need to register a new application in the Google Cast SDK Developer Console and report the provided application ID as `chromecast.app`.

You can protect *Dashkiosk* with a login and a password. For more advanced options, you should put it behind a reverse proxy.

`auth.enabled` Require a login/password. This will enforce the use of HTTP basic authentication to access to *Dashkiosk* (both the receiver and the admin panel).

`auth.realm` Realm to use for HTTP basic authentication.

`auth.username` Valid username for authentication.

`auth.password` Valid password for authentication.

The remaining options can usually be left untouched unless you decided to not use the integrated SQLite database.

path.static Path where the static files to be served for the receiver and the integrated dashboards are located. Unless you moved them to some other location, there is no need to change this.

db.database Database name. This is not needed if you kept the default SQLite database.

db.username Username to access the database. This is not needed if you kept the default SQLite database.

db.password Password to authenticate with the above username. This is not needed if you kept the default SQLite database.

db.options.dialect Dialect to use for the database. This can be `sqlite`, `mysql`, `mariadb` or `postgres`.

db.options.storage Location of the SQLite database. Not used for other databases.

db.options.host Hostname (or IP) where the database is located. This is not needed for SQLite.

log.level Log level to use for logging messages. Use either `info` or `debug`.

log.file Location of a log file where to write logs in JSON format. By default, no such file is generated.

Branding

You can brand a bit *Dashkiosk*. For example, the `deezer` branding provides the following perks:

- The spinning vinyl is located in `app/images/loading-deezer.svg`. You should be able to use anything that will give a cool effect while spinning.
- The *Deezer* logo appearing both in the administration interface and in the default dashboard is located in `app/images/stamp-deezer.svg`.

You can create alternate version of those images and drop them at the same place with a different suffix. Currently, the available brandings are:

- `default`
- `deezer`
- `dailymotion`
- `exoscale`

If you do your own branding, they can be integrated into *Dashkiosk* if the used images can be freely distributed. Submit a pull request.

To select a branding, use the `--branding` option.

Command-line or configuration file

On the command-line, the options are specified using the classic GNU long option style by prepending them with `--`. For example:

```
--port 8087 --environment production
```

Environment variables

You can also specify options using environment variables. In this case, substitute . by __ to get valid values:

```
$ export port=8088
$ export db__database=dashkiosk4
$ export branding=exoscale
```

JSON configuration file

Alternatively, you can specify a JSON configuration file with `--configuration`. In this case, the options with a dot should be understood as being a sub-object. For example, to configure a PostgreSQL database:

```
{
    "environment": "production",
    "db": {
        "username": "dashkiosk",
        "password": "dashkiosk",
        "database": "dashkiosk",
        "options": {
            "host": "172.17.42.1",
            "dialect": "postgres"
        }
    },
    "log": {
        "file": "/var/log/dashkiosk.log"
    }
}
```

Reverse proxy

You may want to put a reverse proxy in front of *Dashkiosk*. You should know that it uses Socket.IO whose preferred backend is WebSocket. Some reverse proxy may not like it.

Here is a configuration for nginx:

```
upstream dashkiosk {
        server localhost:9450;
        server localhost:9451;
}

server {
   listen 80;
   listen [::]:80;
   server_name dashkiosk.example.com;

   location / {
     proxy_pass http://dashkiosk;
     proxy_http_version 1.1;
     proxy_set_header Upgrade $http_upgrade;
     proxy_set_header Connection "upgrade";
     proxy_set_header X-Forwarded-For $remote_addr;
   }
}
```

Setting `X-Forwarded-For` header allows *Dashkiosk* to display the IP address of each display in case you want to log on it to debug it.

CHAPTER 4

Usage

Running the server

Once installed and configured, running *Dashkiosk* should be straightforward. While in the `dist` directory:

```
$ node server.js --environment production
```

Don't forget to specify the environment! See *Configuration* for available options. They can also be specified in a configuration file.

Testing

The server has three browser endpoints:

- `/unassigned` which is the default dashboard for unassigned displays,
- `/admin` which is the administration interface,
- `/receiver` which is the receiver part that a display should load.

To test that everything is setup correctly, point your browser to `/unassigned` (for example `http://localhost:9400/unassigned` if you kept the default parameters and installed *Dashkiosk* on your PC).

You should see the default dashboard displayed for unknown device. These are just a few photos cycling around.

Then, you should go to the administration interface located in `/admin`. While in the administration interface, open another tab and go to `/receiver` which is the URL displaying the receiver. In the `/admin` tab, you should see yourself as a new display in the "Unassigned" group and in the `/receiver` tab, you should see the default dashboard that you got by going in `/unassigned`.

To customize the default dashboard, see *Unassigned dashboard*.

Troubleshooting

If something goes wrong, be sure to look at the log. Either you run the server through something like supervisord and you can have a look at the log in some file or you can use `--log.file` to get a log file.

Administration

The administration interface allows to create new dashboards, see active displays and associate them to a group of dashboards. When pointing a browser to the `/admin` URL, you should see an interface like this:

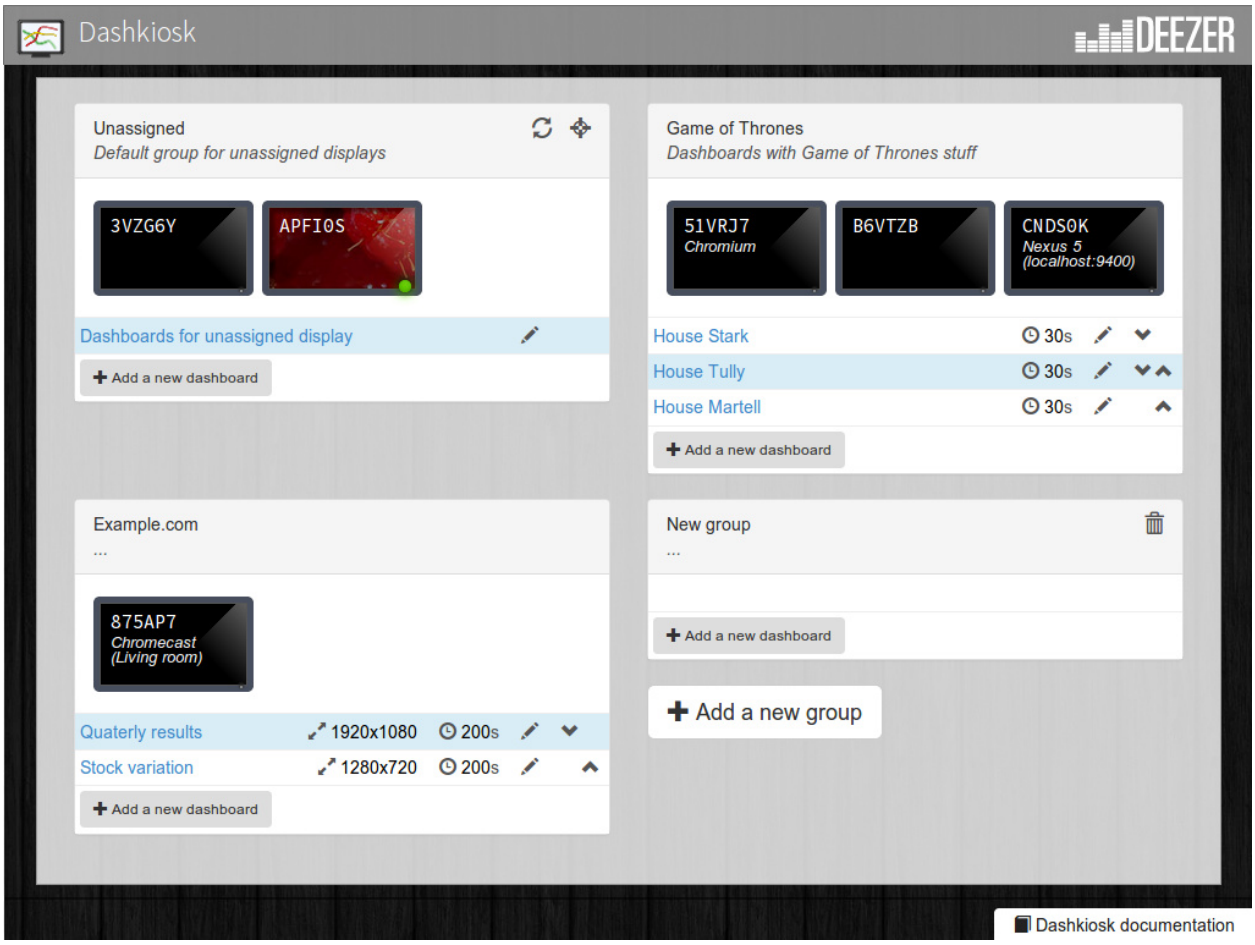

Figure 4.1: The administration interface with a few groups. At the top, the special "Unassigned" group.

On the figure above, you can see the three main entities in *Dashkiosk*:

1. The monitors with a 5-digit serial numbers are the **displays**. For each of them, the serial number is attributed on their first connection and stored locally in the display [1]. They come with a green light when they are actually alive.
2. Each display is affected to a **group** of displays. In the above figure, we have three groups. It is possible to move a display from one group to another. Each group can have a name and a description. It is possible to create or rename any group. The group named "Unassigned" is special and new displays will be attached to it on first connection. Other than that, this is a regular group. The other special group is "Chromecast devices". See *Chromecast devices*.
3. Each group of displays contains an ordered list of **dashboards**. A dashboard is just an URL to be displayed with a bunch of parameters. You can reorder the dashboards in a group and choose how much time they should

[1] The serial number is stored either in the local storage of the browser or in a cookie. If a display comes without this serial number or with an invalid one, it will be granted a new one. The appropriate token is also put in the URL in case neither cookies or local storage are available. This way, you can point the browser to the receiver part, then bookmark or turn the web page as an application.

be displayed.

The first time, you will only have the special "Unassigned" group [2].

Displays

Clicking on a display will show a dialog box with various information about the display.

Figure 4.2: The dialog box of the `APFI0S` display.

First, you get the IP address of the display. This could be useful if you need to connect to it for some other purpose (like debugging a problem related to this display). If the display is offline, the IP displayed is the last known IP.

Then, on the top right corner, there are contextual icons relevant to the current display. On a display, you can execute two actions:

- force a reload of the receiver (after an update, for example),
- toggle the OSD on the receiver.

The receiver OSD is a neat feature to check if the display you are inspecting is really the one you are interested in. It will display an overlay with the display name as well as some technical information that may be useful when displaying dashboards.

Not shown on the above figure, you can destroy a display by clicking on the `Delete` button in the lower left corner.

You can assign a description to the display, like "*In the kitchen*". You can also change the group the display is currently attached to by choosing another group in the dropdown menu. The display should immediatly display the current dashboard of the group.

[2] If you don't have this group, this may be because no display has ever been registered. In this case, point your browser to the `/receiver` URL to register one.

The viewport will be explained in a dedicated section. See *Viewport*.

On a desktop browser, it is also possible to move the display to another group by dragging it to the appropriate group.

Groups

By default, you only get the "Unassigned" group. But you can create any number of groups you need by clicking on the "*Add a new group*" button.

The name and the description of a group can be changed by clicking on them. If you change the name of the "Unassigned" group, a new "Unassigned" group will be created the next time a new display comes to live.

As for displays, you can execute contextual actions on a group. There are three of them:

- for a reload of all the displays in the group,
- toggle the OSD of all the displays in the group,
- destroy the group.

The group can only be destroyed if no display is attached to it.

Each group has a list of dashboards. You can reorder them by using the up and down arrow icons on the right of each dashboard. You can add a new dashboard by using the "*Add a new dashboard*" button.

You can also move or copy a dashboard from another group by using drag'n'drop. By default, a move operation is issued. However, by using a modifier, you will copy the dashboard. Copying a dashboard from one group to the same group will result in a duplicate.

Dashboards

When creating a dashboard or modifying an existing one (by clicking on the little pen icon), you will get the following dialog box:

Currently, a dashboard has:

- an URL
- an optional description
- a timer to tell how much time the dashboard should be displayed
- a timer to tell how much time the dashboard should be delayed before displayed once ready
- a viewport size (see *Viewport*).
- some availability rules

The first timer (*timeout*) is optional but it doesn't make sense to omit it if you have several dashboards in a group. Without it, once the dashboard is displayed, the next one will never be displayed unless you remove or modify the current one.

The second timer (*delay*) is useful for dashboards triggering the *ready* event while they are not really ready to be displayed. This is common for dashboards grabbing stuff with Javascript. There is no easy way to tell when the dashboard is really ready to be displayed. A workaround is to specify this delay. The receiver will wait for the given amount of time before displaying the dashboard.

The time spent loading the dashboard and the additional specified delay are reducing the amount of time a dashboard is displayed. If a dashboard takes 5 seconds to load and an additional delay of 5 seconds is requested, if it is specified to be displayed for 30 seconds, it will in fact be displayed only during 20 seconds.

You can also modify the timer and the viewport by clicking on them directly in the list of dashboards in each group.

It is possible to make some dashboards selected only when some time criteria are matched. The rules can be written in plain English. Here are some examples:

- `after 9:00 am and before 6:00 pm` will only display the dashboard during the day
- `except on saturday and sunday` will not display the dashboard on Saturday nor on Sunday.
- `after 9:00 am and before 6:00 pm except on saturday and sunday` will only display the dashboard during work hours.
- `after 9:00 am and before 6:00 pm on monday through friday` achieves the exact same effect.

If one rule matches, the dashboard will be displayed. You need to ensure that there is at least one dashboard in each group that can be displayed at anytime. For the complete grammar, have a look at the documentation of Later.js. Each rule is prefixed by `every 1 second`.

The availability of a dashboard is only assessed when moving from one dashboard to another. If a dashboard becomes unavailable while the timeout is not elapsed, no change will happen.

Display configuration

For Android displays, see *Android application*. For Chromecast devices, see *Chromecast devices*. For all other displays, you need to point a web browser to the receiver URL (the one ending with `/receiver`).

If you want additional animations, you can use `/receiver-fast` instead. However, if your device is quite slow (for example, a Raspberry Pi without hardware acceleration), you can disable most animations by using `/receiver-slow`.

About the dashboards

The dashboards to be displayed can be any URL accessible by the displays. When a new dashboard has to be displayed for a group, the server will broadcast the URL of the dashboard to each member of the group. They will load the dashboard and display it. This may seem easy but there are several limitations to the system.

Network access

So, the first important thing about those dashboards is that they are fetched by the displays, not by the server. You must therefore ensure that the dashboards are accessible by the displays and not protected by a password or something like that.

Processing power

Some dashboards may be pretty dynamic and use special effects that look cool on the average PC. However, when using a US$ 30 low-end Android stick to display it, it may become a bit laggy. Also, please note that the Android application uses a modern webview but some functionalities may be missing, like WebGL.

Viewport

By default, a dashboard is displayed using the native resolution of the display. If the display is a 720p screen and your dashboard can only be rendered correctly on a 1080p screen, you have a problem. There are several solutions to this problem.

1. Use a responsive dashboard that can adapt itself to any resolution.
2. Change the viewport of the display. By clicking on the display, you can specify a viewport. When empty, it means that you use the viewport matching the native resolution of the screen. By specifying another resolution, the display will render the dashboards at the given resolution and zoom in or out to fit it into its native resolution.

   The support of this option depends on the ability of the browser running the receiver to exploit this information. Android devices are able to make use of it but other devices may not. If you don't see any effect when changing the viewport, use the next option.
3. Change the viewport of the dashboard. This is quite similar to the previous option but it is a per-dashboard option and it will work on any device. It works in the same way: the rendering will be done at the given resolution and then resized to fit in the screen. Both options can be used at the same time, there is no conflict.

IFrames

Technically, the receiver is a simple app rendering the requested URL inside an IFrame which is like a browser inside a browser. There are some limitations to an IFrame:

- The receiver has almost no way to communicate with the IFrame [3]. It can know when an IFrame is ready but not if there is an error. The IFrame can therefore be displayed while it is not fully rendered and on the other hand, we cannot detect any error and try to reload the IFrame.
- The IFrame can refuse to be display its content if there is a special `X-Frame-Options` in the headers forbidding the use of an IFrame.
- The content embedded inside the IFrame can be confused because it uses IFrame too and expects to be at the top of the structure. Some Javascript code therefore tries to access to *Dashkiosk* window and gets an unexpected error. This is the case with JIRA.
- If you are serving *Dashkiosk* from an HTTPS URL, you cannot display dashboards using HTTP. The other way is authorized. Hence, it seems just easier to serve Dashkiosk receiver on HTTP.

The second limitation can be quite annoying. Here are some workarounds:

1. Find an embeddable version of the content. Youtube, Google Maps and many other sites propose a version specifically designed to be embedded into an iframe.
2. Use a web proxy that will strip out the offending header. A good base for such a proxy is Node Unblocker. It should be easy to modify it to remove the `X-Frame-Options` header. If you use *nginx* to reverse proxy a service, you can use `proxy_hide_header X-Frame-Options`. If the application is using Ruby on Rails, add `set :protection, :except => :frame_options` in `config.ru`.
3. Use a screenshot service. Instead of displaying the real website, just display a screenshot. There are many solutions to implement such a service with headless browsers like Phantom.JS. For example this one.

Writing dashboards

***Dashkiosk* can only displays existing dashboards. Here is a list of** services you can use to create them:

- *Grafana <https://grafana.com/>*, dashboards for time series
- *hoplaJS <https://hoplajs.golflima.net/>*, a web application running JS code stored in URLs
- *Dashing <http://dashing.io/>*, a dashboard framework (discontinued, see *Smashing <https://github.com/Smashing/smashing>*)

[3] If the iframe is in the same domain, it can communicate with the iframe. However, most of the time, this is not the case. The receiver can also communicate with a cooperating iframe by sending messages. This is currently not implemented.

CHAPTER 5

API

There are three API available in Dashkiosk:

- a REST API to manipulate groups, dashboards and displays,
- the change API which is a Socket.IO based API which broadcasts changes to subscriber,
- the display API which is a Socket.IO based API which tells displays which URL they should display.

There is also an internal bus API.

REST API

The REST API is available on the `/api` endpoint. Only JSON is currently supported. On error, the HTTP error code is important and the error message is also encapsulated into a JSON object:

```
{
    "error": {
        "httpCode": 404,
        "message": "No display named \"CNDS0KD\".",
        "name": "NotFoundError",
        "stack": [
            "NotFoundError: No display named \"CNDS0KD\".",
            "    at dashkiosk/lib/models/display.js:154:15",
            "    at process._tickDomainCallback (node.js:459:13)",
            "    at process._tickFromSpinner (node.js:390:15)",
            "From previous event:",
            "    at new Promise (dashkiosk/node_modules/sequelize/node_modules/bluebird/js/main/promi
            "    at module.exports.CustomEventEmitter.then (dashkiosk/node_modules/sequelize/lib/emit
            "    at Function.Display.get (dashkiosk/lib/models/display.js:152:6)",
            "    at dashkiosk/lib/api/rest/displays.js:21:20",
            "    at callbacks (dashkiosk/node_modules/express/lib/router/index.js:164:37)",
            "    at param (dashkiosk/node_modules/express/lib/router/index.js:138:11)"
        ]
    },
    "message": "No display named \"CNDS0KD\".",
    "token": "1397254337651-5YHK0SFJRC"
}
```

The `error` attribute is only present in development mode. It can also be found in the logs thanks to the `token` attribute.

Displays

GET /api/display
The list of all known displays.

Example request:

```
GET /api/display HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "3VZG6Y": {
      "connected": false,
      "description": null,
      "group": 1,
      "id": 2,
      "ip": null,
      "name": "3VZG6Y",
      "viewport": null
  },
  "51VRJ7": {
      "connected": false,
      "description": "Chromium",
      "group": 2,
      "id": 7,
      "ip": "127.0.0.1",
      "name": "51VRJ7",
      "viewport": null
  }
}
```

Status Codes

- **200** – no error

PUT /api/display/(*name*)
Modify the attributes of the display *name*.

Example request:

```
PUT /api/display/CNDS0K HTTP/1.1
Accept: application/json

{ "viewport": "1920x1080" }
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "connected": true,
  "description": "Nexus 5 (localhost:9400)",
  "group": 2,
  "id": 5,
```

```
  "ip": null,
  "name": "CNDS0K",
  "viewport": "1920x1080"
}
```

Parameters

- **name** – name of the display

JSON Parameters

- **description** – new description for the display
- **viewport** – new viewport for the display

Status Codes

- **200** – no error
- **404** – display not found

PUT /api/display/(*name*)**/group/**
int: *id* Attach the display *name* to the group *id*.

Example request:

```
PUT /api/display/CNDS0K/group/10 HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "connected": false,
  "description": "Nexus 5 (localhost:9400)",
  "group": 10,
  "id": 5,
  "ip": null,
  "name": "CNDS0K",
  "viewport": "1920x1080"
}
```

Parameters

- **name** – name of the display
- **id** – ID of the group

Status Codes

- **200** – no error
- **404** – display or group not found

POST /api/display/(*name*)**/action**
Request an action on a display. Only if connected.

Example request:

```
POST /api/display/CNDS0K/action HTTP/1.1
Accept: application/json

{ "action": "reload" }
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "connected": true,
  "description": "Nexus 5 (localhost:9400)",
  "group": 10,
  "id": 5,
  "ip": null,
  "name": "CNDS0K",
  "viewport": "1920x1080"
}
```

Parameters

- **name** – name of the display

JSON Parameters

- **action** – requested action, either `reload` or `osd`
- **text** – for OSD only, text to display or `null` to remove the OSD

Status Codes

- **200** – no error
- **400** – unknown action
- **404** – display not found or offline

DELETE /api/display/(*name*)

Delete the display *name*. Only possible if the display is not connected anymore.

Example request:

```
DELETE /api/display/CNDS0K HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 204 OK
Content-Type: application/json
```

Parameters

- **name** – name of the display

Status Codes

- **204** – no error
- **404** – display not found
- **409** – display still connected

Groups

GET /api/group
The list of all known groups.

Example request:

```
GET /api/group HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "1": {
      "description": "Default group for unassigned displays",
      "id": 1,
      "name": "Unassigned"
  },
  "2": {
      "description": "Dashboards with Game of Thrones stuff",
      "id": 2,
      "name": "Game of Thrones"
  }
}
```

Status Codes

- **200** – no error

POST /api/group
Create a new group

Example request:

```
POST /api/group HTTP/1.1
Accept: application/json

{
  "name": "New group"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 9,
  "name": "New group"
}
```

JSON Parameters

- **name** – name of the group
- **description** – description of the group

Status Codes

- **200** – no error
- **400** – a group should have a name
- **409** – a group with the same name already exists

PUT /api/group/ (**int:** *id*)
Modify a group attributes.

Example request:

```
PUT /api/group/15 HTTP/1.1
Accept: application/json

{
  "name": "Another name",
  "description": "Fancy"
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 9,
  "name": "Another name",
  "description": "Fancy"
}
```

JSON Parameters

- **name** – name of the group
- **description** – description of the group

Status Codes

- **200** – no error
- **409** – a group with the same name already exists

DELETE /api/group/ (**int:** *id*)
Delete the group. Only possible if no display are attached.

Example request:

```
DELETE /api/group/15 HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 204 OK
Content-Type: application/json
```

Parameters

- **id** – ID of the group

Status Codes

- **204** – no error

- **404** – group not found
- **409** – group with displays

Dashboards

GET /api/group/ (**int:** *id*) **/dashboard**
The list of all dashboards in a group

Example request:

```
GET /api/group/15/dashboard HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
      "active": false,
      "description": "House Stark",
      "group": 15,
      "id": 2,
      "timeout": 30,
      "url": "http://www.gameofthronescountdown.com/#stark",
      "viewport": null
  },
  {
      "active": true,
      "description": "House Tully",
      "group": 15,
      "id": 3,
      "timeout": 30,
      "url": "http://www.gameofthronescountdown.com/#tully",
      "viewport": null
  }
]
```

Parameters

- **id** – group ID

Status Codes

- **200** – no error
- **404** – the group doesn't exist

POST /api/group/ (**int:** *id*) **/dashboard**
Create a new dashboard

Example request:

```
POST /api/group/15/dashboard HTTP/1.1
Accept: application/json

{
  "url": "http://www.example.com",
```

```
  "timeout": 30
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
      "active": false,
      "group": 15,
      "id": 6,
      "timeout": 30,
      "url": "http://www.example.com"
}
```

Parameters

- **id** – group ID

JSON Parameters

- **url** – URL of the dashboard
- **description** – description of the dashboard
- **timeout** – timer for this dashboard
- **viewport** – viewport for this dashboard

Status Codes

- **200** – no error
- **404** – group not found
- **409** – the URL is mandatory

PUT /api/group/ (**int:** *id*) **/dashboard/**
int: *dashid* Modify an existing dashboard. The special attribute *rank* can be used to modify the position of the dashboard in the group. The dashboards are numbered from 0 and the rank is the target position we want.

Example request:

```
POST /api/group/15/dashboard/6 HTTP/1.1
Accept: application/json

{
  "timeout": 40
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
      "active": false,
      "group": 15,
      "id": 6,
      "timeout": 40,
```

```
    "url": "http://www.example.com"
}
```

Parameters

- **id** – group ID
- **dashid** – dashboard ID

JSON Parameters

- **url** – URL of the dashboard
- **description** – description of the dashboard
- **timeout** – timer for this dashboard
- **viewport** – viewport for this dashboard
- **rank** – New position for the dashboard

Status Codes

- **200** – no error
- **404** – dashboard or group not found

DELETE /api/group/ (**int:** *id*) **/dashboard**
int: *dashid* Delete the dashboard.

Example request:

```
DELETE /api/group/15/dashboard/6 HTTP/1.1
Accept: application/json
```

Example response:

```
HTTP/1.1 204 OK
Content-Type: application/json
```

Parameters

- **id** – ID of the group
- **dashid** – ID of the dashboard

Status Codes

- **204** – no error
- **404** – group or dashboard not found

Changes API

The socket.IO endpoint for this API is `changes`. Upon connection, a client will get all the current groups. Each group has a collection of displays in the `displays` attribute and an array of dashboards in the `dashboards` attribute. Here is an example:

```
{
  "1": {
      "description": "Default group for unassigned displays",
      "id": 1,
      "name": "Unassigned",
      "displays": {},
      "dashboards": []
  },
  "2": {
      "description": "Dashboards with Game of Thrones stuff",
      "id": 2,
      "name": "Game of Thrones"
      "displays": {
         "CNDS0K": {
            "connected": true,
            "description": "Nexus 5 (localhost:9400)",
            "group": 10,
            "id": 5,
            "ip": null,
            "name": "CNDS0K",
            "viewport": "1920x1080"
         }
       }
      "dashboards": [
            {
              "active": false,
              "description": "House Stark",
              "group": 2,
              "id": 2,
              "timeout": 30,
              "url": "http://www.gameofthronescountdown.com/#stark",
              "viewport": null
            },
            {
              "active": true,
              "description": "House Tully",
              "group": 2,
              "id": 3,
              "timeout": 30,
              "url": "http://www.gameofthronescountdown.com/#tully",
              "viewport": null
            }
      ]
  }
}
```


This message will be labeled `snapshot`.

On changes, only the group or the display affected by the change will be sent. The label of the message will be one of:

- `group.deleted`
- `group.updated`
- `group.created`

And for displays:

- `display.deleted`
- `display.updated` (also for new displays)

If a change affects a dashboard, the whole group will be sent nonetheless.

Display API

This API is used by the display to know what they should do. The socket.IO endpoint to use for it is `displays`.

The server attributes to each new display a serial number. The display is expected to remember it and transmit it back on the next connection. It is encrypted by the server to avoid the display to steal another display identity.

Upon connection, a display is expected to send a `register` message with an object containing the `blob` attribute with the encrypted identity it previously received (if any).

If the server accepts the identity as is, it answers to this message with the same blob that should be stored by the client. If not, it will generate a new blob and sends it back to the client. In both cases, the client just has to store the received blob.

After this handshake, the display can receive the following messages:

`dashboard` The dashboard that should be displayed right now. It is an object containing the same attributes as we would have got when requesting this particular dashboard with the REST API. See *Dashboards*.

`reload` The display should reload itself.

`osd` The OSD should be shown or hidden. If the message comes with a text, the OSD is displayed with the provided text. Otherwise, it is hidden.

`viewport` Modify the current viewport of the display with the provided value.

Internal bus message

To avoid strong coupling between components, *Dashkiosk* uses postal.js as an internal bus message. The messages that are emitted are listed below:

- `display.NAME.connected` when a new display is connected
- `display.NAME.disconnected` when a new display is disconnected
- `display.NAME.group` when a display should change to a new group
- `display.NAME.deleted` when a display is deleted
- `display.NAME.updated` when another change happens on a display
- `display.NAME.dashboard` when a new dashboard should be displayed by the given display.
- `display.NAME.reload` when a display should reload itself
- `display.NAME.osd` when we need to display something on the OSD
- `display.NAME.viewport` when the display viewport should be updated
- `group.ID.created` when a new group is created
- `group.ID.updated` when a group is updated (but not something dashboard related)
- `group.ID.deleted` when a group is deleted
- `group.ID.dashboard` when a whole group should switch to a new dashboard
- `group.ID.dashboard.ID.added` when a new dashboard has been added

- `group.ID.dashboard.ID.removed` when a dashboard has been removed
- `group.ID.dashboard.ID.updated` when a dashboard has been updated

Each message comes with the group, the dashboard and/or the display specified in the message (when this is relevant).

CHAPTER 6

Android application

This is a simple Android application whose purpose is to display fullscreen non-interactive dashboards on Android devices. Its main use is to be run from an Android stick plug on some TV to run the web application to display dashboards.

Supported devices

Currently, the minimal version of Android is 4.1 (Jelly Bean. *Dashkiosk* is using the Crosswalk project to provide an up-to-date webview with support of recent technologies.

There are a lot of Android devices that you can choose to run Dashkiosk on. When choosing one, prefer the ones which can be upgraded to Android 4.2.

Many cheap devices are using a Rockchip SoC limited to a 720p resolution. This can be circumvented with some non official experimental firmwares but usually, this is a sufficient resolution to display dashboards.

The following devices [1] are known to work reasonably well:

- MK809III Mini PC (RK3188 SoC)
- T95N-Mini M8SPro (Amlogic S905X)

Features

- It registers as a possible home screen. It is therefore to run the application on boot.
- It provides a really fullscreen webview. Absolutely no space lost in bars.
- No possible interactions. If run on a tablet, the user is mostly locked out. However, there are still some way to interact with the device while the application is running by invoking the settings and changing the home application from here.
- Prevent the device going to sleep.

Compilation

If you don't want to compile the Android app yourself, you can download a pre-compiled version from GitHub.

[1] Please, open an issue if you want to contribute to this list.

Building from source is just a matter of following those two simple steps:

1. Clone the git repository.
2. Build the application with the following command:

   ```
   ./gradlew assemble
   ```

At the end of the compilation, you get `build/outputs/apk/dashkiosk-android-debug.apk` that should be installed on the Android device.

Installation

You need the `adb` tool. On Debian and Ubuntu, you can install the `android-tools-adb` package to get it. Otherwise, it is available in the `platform-tools` of the Android SDK. If you didn't install the SDK yourself, it should be in `~/.android-sdk`. If `adb` is not present in the `platform-tools` directory, you can install it with:

```
tools/android update sdk --no-ui --all --filter platform-tool
```

You can then install the APK on a device attached through USB on your computer with the following command:

```
adb install -r build/outputs/apk/dashkiosk-android-debug.apk
```

Alternatively, you can just point a browser to the APK and you will get proposed to install it. You need to ensure that you allowed the installation of APK from unknown sources.

The next step is to run the configuration panel. This panel can be accessed by using the back button while the loading screen is running. It can be accessed later by clicking on the pen icon in the action bar.

Configuration

The **orientation** is configured to *landscape* by default. You can choose either *auto* or *portrait*.

If you want to lock a bit the application, you can **lock settings** to prevent any further modifications. You can still revert the changes by invoking the preferences activity with `adb`:

```
adb shell am start -n \
   com.deezer.android.dashkiosk/com.deezer.android.dashkiosk.DashboardPreferences
```

The important part is to input the **receiver URL**. You can check that this is the correct URL with any browser. You should see a dashboard with some nice images cycling.

The **timeout** is not really important. Until the application is able to make contact with the receiver, it will try to reload the receiver if the timeout is reached.

Alternatively, the configuration can be done at compile-time by modifying `res/xml/preferences.xml`.

Certificates

Server certificates

Unfortunately, it is currently not possible to trust third-party certificates. Trusted certificates are built into the app and cannot be modified.

The only possibility is to accept untrusted certificates in the preferences. This makes TLS useless and you could just use HTTP, except if you are interested in client certificates. In this case, blindly trusting the server certificate doesn't allow an attacker to use your client certificate for its own requests (client has to demonstrate its ability to sign a the whole handshake with its certificate, including the "server certificate" message).

Client certificates

It is possible to use client certificates. The support is still quite new and may be troublesome to implement. Be sure to use `adb logcat -s DashKiosk AndroidRuntime` while running to spot any error.

Creating a keystore

Currently, you can only provide one client certificate and it will be used with any site requesting a client certificate. The certificate needs to be provided as a BKS (BouncyCastle KeyStore). You can either use `keytool` or Portecle, a graphical tool to manage such a store. You can find a cheatsheet to use `keytool`. If you already have your client certificate as a PKCS#12 file, you only need to use `keytool -importkeystore`:

```
keytool -importkeystore \
        -destkeystore clientstore.bks \
        -deststoretype BKS \
        -provider org.bouncycastle.jce.provider.BouncyCastleProvider \
        -providerpath /usr/share/java/bcprov.jar \
        -srckeystore client.p12 \
        -srcstoretype PKCS12
```

You will be prompted the password to protect the newly created keystore and the password protecting the PKCS#12 file. Ensure you use the same password for both: `keytool` seems to protect the private key with the password from the PKCS#12 file while *Dashkiosk* will use the same password for the private key and for the keystore.

On Debian, `bcprov.jar` is from the `libbcprov-java` package. Be sure to only put one keypair in the store. *Dashkiosk* wil always use the first one.

If you have your certificates in PEM format, you can convert them in PKCS#12 with the following command:

```
openssl pkcs12 -export -out client.p12 \
               -in cert.pem \
               -inkey key.pem \
               -certfile ca.pem
```

You can import several certificates in the keystore.

Providing the keystore to the application

There are two ways to provide a client certificate to the application. The first one is to put the certificate on the filesystem. For example, in `/sdcard/dashkiosk.bks`. Then, in the preferences, ensure to untick *Embedded keystore* and tick *External keystore*, then specify the path to the keystore in *Keystore path*. The second one is to embed the client certificate directly into the application. Replace the file `res/raw/clientstore.bks` by your own and recompile the application. In the preferences, ensure you tick *Embedded keystore*. In both cases, you also need to provide the password protecting the keystore.

Grant permissions to read the keystore

Starting from Android 6, you also have to grant *Dashkiosk* the permission to access the keystore if you use the external one. This can be done in *Android Settings*. Go to *Applications*, click on *Dashkiosk*. You should see a *Permissions* tab.

The only item in this tab should be *Storage*. Enable it.

Usage

Once configured, just run the application as usual. You can also click on the home button and choose the application from here to make it starts on boot.

Troubleshooting

Still with `adb`, you can see the log generated by the application with the following command:

```
adb logcat -s DashKiosk AndroidRuntime
```

The log also includes Javascript errors that can be generated by the dashboards. Javascript errors from the receiver are prefixed with `[Dashkiosk]`.

CHAPTER 7

Chromecast devices

Dashkiosk can optionnaly handle Chromecast devices. When the support is enabled (see *Available options*), *Dashkiosk* will be able to discover Chromecast devices on the network.

Warning

Due to the closedness of the Chromecast platform, its support in Dashkiosk may break from time to time. As soon as it doesn't work anymore, feel free to signal it in an issue. If needed, it is possible to help debugging by registering a Chromecast device on the same account hosting the custom receiver.

However, there is no guarantee that the issue is easy to fix and it may take some time. Therefore, you must be prepared for the dashboards to break at any time without prior warnings (the Chromecast are updating themselves without user consent).

Setup

To be able to handle Chromecast devices, you need to enable its support (see *Available options*). You also need to ensure that mDNS is working correctly on your setup.

When starting *Dashkiosk*, you may get this error message:

```
*** WARNING *** The program 'nodejs' called 'DNSServiceRegister()' which is not supported (or only su
*** WARNING *** Please fix your application to use the native API of Avahi!
```

This is perfectly harmless.

For Linux, use the following command to check that you can see The Chromecast devices:

```
$ avahi-browse _googlecast._tcp
+ eth0.20 IPv4 Chromecast Here              _googlecast._tcp     local
+ eth0.20 IPv4 Chromecast Demo              _googlecast._tcp     local
```

If this first step doesn't work, check you have Avahi installed and running. Usually, this can be done with `apt-get install avahi-daemon`. Then, check you don't have any firewall preventing multicast traffic to port 5353.

Please note that you need to be on the same network as the Chromecast devices [1].

Then, check that you can resolve the Chromecast names:

[1] The mDNS packets are usually using a TTL of 1 and therefore cannot be routed even if you try to setup multicast routing on your network. The correct solution is to configure a DNS-SD name server.

```
$ getent hosts "Chromecast\032Here".local
192.168.0.195    Chromecast\032Here.local
```

If this doesn't work, you need to configure your resolver to use multicast DNS. This is usually done by putting the following line in `/etc/nsswitch.conf`:

```
hosts: files mdns4_minimal [NOTFOUND=return] dns mdns4
```

On OS X, you can check if you can see the Chromecast devices with `dns-sd -B _googlecast._tcp`, then try to resolve with `dns-sd -G v4 "Chromecast\032Here".local`.

Usage

Discovered Chromecast devices will be assigned to the special group "Chromecast devices". This group has no dashboard on purpose: if a Chromecast device is on a group with a dashboard, *Dashkiosk* will wait for the Chromecast device to be on the home screen and starts a custom receiver which will load the regular receiver and turn your Chromecast device into a regular display.

You can either add dashboards to the "Chromecast devices" group to let all Chromecast devices display a dashboard on inactivity or move the selected Chromecast devices to another group.

Custom receiver

To display dashboards, a Chromecast device is requested to run a custom receiver, which is just some HTML5 application. Unfortunately, we cannot just provide an URL for that, we have to give an application ID. The Chromecast device will ask Google which URL to use and Google will provide the URL. By default, *Dashkiosk* will use an application hosted on some GitHub URL. This should work just fine.

If you want to use your own custom receiver (or modify the existing one), you need to declare a new application (and pay US$ 5). That's a bit unfortunate. Maybe we could do an application that will just forward to another application.

Troubleshooting

Let me explain how the whole thing works.

1. *Dashkiosk* detects the Chromecast device using multicast DNS. If it doesn't see yours, you need to check that multicast is correctly working on your network.
2. If the Chromecast is in a group with dashboards, *Dashkiosk* then asks to load the custom receiver using its application ID (`5E7A2C2C`). The Chromecast will then ask Google which URL it should load for this application ID. Google will send back the GitHub URL hosting the custom receiver. The Chromecast will load it. You should get a screen "*This, Jen, is the Internet*".
3. *Dashkiosk* will send the receiver URL, as you configured it in the *Available options* (or with `--chromecast.receiver`). Once the custom receiver gets this URL, it displays it in the lower left part of the screen. If it isn't displayed, it is likely to be a bug in *Dashkiosk*. Have a look in the logs and open an issue.
4. The custom receiver will then load the regular receiver. If you don't get to this step and are stuck on the "*This, Jen, is the Internet*" screen, it means that your Chromecast is not able to retrieve the receiver you provided. It could be a firewall issue or a DNS issue. Try to connect your laptop on the same network as the Chromecast and load the receiver URL yourself to see what is happening.

Troubleshooting is quite complex. Due to a recent change, users are only allowed to debug their own application. You need to register your Chromecast and register and host a copy of the Chromecast receiver. You'll get an ID for the Chromecast application and should use `--chromecast.app` to specify it.

Once the application is running, you can connect to your Chromecast device using its IP on port 9222. The Chromecast needs to be running the custom receiver. With recent versions of Chrome, you are likely to get mixed content restrictions. Click on the shield in the URL bar to lift this restriction.

Also, the Chromecast is a low-end device (but quite capable): it may have difficulties to display complex dashboards. You may want to try to load the dashboard alone using a simple sender. If the dashboard is unable to render correctly even with this sender, try to reduce its complexity. Moreover, during transitions, the Chromecast has to be able to display the current dashboard while the next one is rendered in the background. This may use too much resources. Try to not load two consecutive complex dashboards.

CHAPTER 8

Cloud install

You can run *Dashkiosk* on a PaaS like Heroku or Dokku. The Chromecast support is unlikely to work in this case but the remaining functionalities should work without any problem. The following documentation is done using Heroku. The process should be quite similar with another provider.

We assume that you already have an account registered on Heroku. It comes with a free tier which should be sufficient to test *Dashkiosk*.

Deployment tool installation

The first step is to install the deployment tool. For *Heroku*, follow the install documentation.

Create the application

The next step is to create the *Dashkiosk* application. For that purpose, a modified buildpack needs to be used as *Dashkiosk* is using `grunt` to be built:

```
$ git clone https://github.com/vincentbernat/dashkiosk.git
$ cd dashkiosk
$ heroku create
$ heroku config:set NPM_CONFIG_PRODUCTION=false
```

You can then push the application:

```
$ git push heroku master
```

If you want a custom branding, you have to use the following command:

```
$ heroku config:set branding=exoscale
```

Configure a database

By default, *Dashkiosk* will use an SQLite database. However, with a PaaS, this means that the database will be lost at each deployment. It is therefore more convenient to use a MySQL database:

```
$ heroku addons:create heroku-postgresql:hobby-dev
$ heroku pg:wait
$ heroku config -s | grep HEROKU_POSTGRESQL
HEROKU_POSTGRESQL_RED_URL=postgres://user3123:passkja83kd8@ec2-117-21-174-214.compute-1.amazonaws.com
```

The settings are available through environment variables. Unfortunately, *Dashkiosk* won't use them. It is also possible to put them into a credential file but *Dashkiosk* won't know how to use them either.

The easiest way is to arrange for those settings to be available as environment variables under the right name:

```
$ heroku config:set db__options__dialect=postgres
$ heroku config:set db__options__host=ec2-117-21-174-214.compute-1.amazonaws.com
$ heroku config:set db__options__port=6212
$ heroku config:set db__database=db982398
$ heroku config:set db__username=user3123
$ heroku config:set db__password=passkja83kd8
```

Deploy the application

The application can now be deployed:

```
$ git push heroku master
```

If you get a problem, use the log subcommand:

```
$ heroku logs --tail
```

Upgrading

Upgrading is just a matter of pulling new changes and pushing them to the PaaS:

```
$ git pull
$ git push heroku master
```

CHAPTER 9

License

Dashkiosk is distributed under the ISC license. It basically means: do whatever you want with it as long as the copyright sticks around, the conditions are not modified and the disclaimer is present.

ISC License

> Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.
>
> THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Branded images

As an exception, branded images are free to redistribute but cannot be modified except when the brand logo is completely removed. Branded images are all images prefixed by `app/images/stamp-` or `app/images/loading-`, except `app/images/stamp-default.svg` and `app/images/loading-default.svg`.

CHAPTER 10

Indices and tables

- *genindex*
- *search*

/api