The ReCollect Dashboard Framework

A How-To for making dashboards and widgets

Contents

How the dashboard works

When a client connects to the server hosting the dashboard service, they are immediately served with a blank shell of a page, containing only some outline formatting, and the client-side JavaScript needed to run the dashboard requests.

They then begin to request updates from the server at a continuous rate. Each time the client requests an update, the server follows these steps:

The server looks into the clients' configuration data to see which set of dashboards this particular client is supposed to receive in the rotation. It checks to see if the current dashboard displayed on the client has been displayed for its entire time allotment, and if it has, it creates the next dashboard for the client.

When a dashboard is created, the server looks into the dashboards' configuration data to determine which widgets are to be displayed on the given dashboard. Each of these widgets is created and displayed in the order they are entered into the configuration data.

When creating a widget, the server reads the widgets' configuration file to see what data (the model) the widget is to use to render which widget type (the view). The server calls on the widget's model's getData() function, which returns an associative array of view parameters. These parameters are then passed into the view parser, which takes the html view fragment associated with a widget, and replaces any elements found in {braces} with the corresponding value of the parameters array. Widgets can also bring in JavaScript files which they may require to run their content.

The dashboard full of widgets is then returned to the client, who displays it.

Models

Models are classes which must inherit from the provided baseModel class. They are only required by that inheritance to implement the getData() function.

Models must return all the data required for the view parsing, in the form of an associative array. If a model is intended to be used with different views, it must return enough information to satisfy all of the views.

The getData() function is passed one parameter, which is the content on the "additionalParameters" item in the widget's configuration if present, or null otherwise.

Each model object is created, used once, and then destroyed. They cannot be used to store persistent data themselves.

```
<?php

require_once 'baseModel.php';

class myDataModel extends baseModel {


    function getData($widgetParams) {
          $ret = array();
          ...
          return $ret;
          }
}
```

A model's file name must be the same as its class name. In the above example, the file should be saved under the models/ folder as "myDataModel.php". It can then later be referenced as "myDataModel".

Views

A view is an html fragment that will make up the widget.

Each specific view is placed inside the html container defined in the baseWidget view, which enforces the size, and specifies the background color and/or image.

baseWidget

```
<div id="{id}" class="widget"
style="width:{width}px;height:{height}px;{backgroundImage};background-
color:{backgroundColor}">
                  {content}
</div>
```

Example Widget

```
<div>
    <h1 class="title" >{title}</h1>
    <h3>{text}</h3>
    <p class="more-info" >{footer}</p>
</div>
```

A “view” is used in this dashboard framework as a type definition.

Save views under the views/ folder as php files and as the name they will later be referenced by.

If this view was saved as “myWidgetType.php”, it can later be referenced as “myWidgetType”

Making A Widget

After having constructed or selected an appropriate model and view for your widget, the widget needs its definition laid out in a JSON file in the data/widgets folder.

A widget definition has the following structure:

```
{
        "time": 30,
        "model": "myDataModel",
        "type": "myWidgetType",
        "width": 1,
        "height": 1,
        "script": "assets/js/myScript.js",
        "additionalParams": ...
}
```

- The time is how long (in minutes) a widget will take between updates
- The model is the name of the model class to populate the data from
- The type refers to the name of the view used for the widget
- The width and height are in dashboard grid units (a dashboard is 6 x 3)
- *The script can be the path to a JavaScript file which will be included in any dashboard containing this widget
- *Additional Params can be any value, object, or array you wish to have passed into the model

*these can be omitted from the definition with no adverse effect.

ex

```
"additionalParams": {“city” : “Vancouver”, “value” : 1, “data” : [...], ... }
```

The widget’s usable name is the name of the JSON file the definition is stored as. That is to say if the above is saved as “myDataWidget.json”, it can be referred to later as “myDataWidget”.

Making A Dashboard

After having created or selected a set of widgets which will form a dashboard, a new dashboard needs its definition laid out in a JSON file in the data/dashboards folder.

A dashboard definition has the following structure:

```
{
        "time": 1,
        "name": "My Dashboard",
        "widgets":
                [
                "myDataWidget",
                ...
                ]
}
```

- The time is how long (in minutes) a dashboard will take before it cycles to the next
- The name is the title displayed at the top of the dashboard
- The widgets is an array of widget definitions in the order they will appear on the dashboard

The dashboard's usable name is the name of the JSON file the definition is stored as. That is to say if the above is saved as "myDashboard.json", it can be referred to later as "myDashboard".

Assigning Dashboards to Clients

Once one or more dashboards have been assembled, to assign a dashboard rotation to a client, a client needs an entry in the clients configuration file.

The clients configuration file has the following structure with no client entries

```
{
        "clients":
                [
                ...
                ]
}
```

The content of each entry in the clients array is an object containing an id, and an array of dashboards.

```
{ "id": 1, "dashboards": [ "myDashboard", ... ] }
```

The dashboards entry can be an array of as many dashboards as the client should have. It can include duplicate dashboard entries as the client will receive the dashboards in the order they are listed in the entry. When the list it exhausted, it will be restarted seamlessly.