DQX Library documentation

1 Introduction

DQX is a framework that assists in the creation of web app with a look and feel that is similar to a desktop application. It was originally created to build the

MalariaGEN P. Falciparum population genetics web application, and has some focus on visualization tools for genomic data. It offers the following functionality:

- A layouting mechanism to organize visual components on a page, mimicking a desktop application interface (e.g. no single overall scroll bar).
- Emulation of a multi-page application using a single JavaScript environment, enabling the usage of the browser "back" button.
- Encapsulation of a set of commonly used GUI elements (trees, lists, various controls, etc…).
- A messaging system to assist communication between several application components.
- Blocking and non-blocking popups.
- A wizard creation framework.
- A framework to fetch database data from the server.
- A paged grid viewer that represents a (queried) table on the server.
- An interactive graphical query builder that can be used in conjunction with the table grid viewer.
- A graphical genome viewer, able to show a rich variety of channels.
- An abstraction layer for the Google Maps API.
- Various small utilities.

The framework consists in two major components: **DQX** and **DQXServer**. In addition, there also is a sample web application that utilizes most of the functionality in the DQX library in a simple way: **DQXTest**. Most of the functionality can be learned by looking at this sample application.

These components are stored in the following GitHub repositories:

https://github.com/malariagen/DQXServer

https://github.com/malariagen/DQX

https://github.com/malariagen/DQXTest

IMPORTANT NOTE*: this documentation does not describe the complete functionality of the library, or of the individual objects mentioned. Consult the source code for more information.*

2 Architecture

DQX is aimed at the creation of rich, desktop-like web apps with a single page interface (SPI). The SPI approach allows for a smooth, fluid user experience, and also causes the entire application to run in a single JS runtime environment. T does mimic multiple pages (called 'views') in the app.

It heavily relies on Ajax for client-server updates, and follows the principles of a Thin Server Architecture: the app GUI is entirely managed client side, and the server only streams the app code and the data.

The server follows the principles of a REST protocol.

The web page GUI elements (DOM) are entirely created through JS (the initial html page is empty). The library also attempts to shield the web app code as much as possible from the creation of raw html markup.

2.1 Components

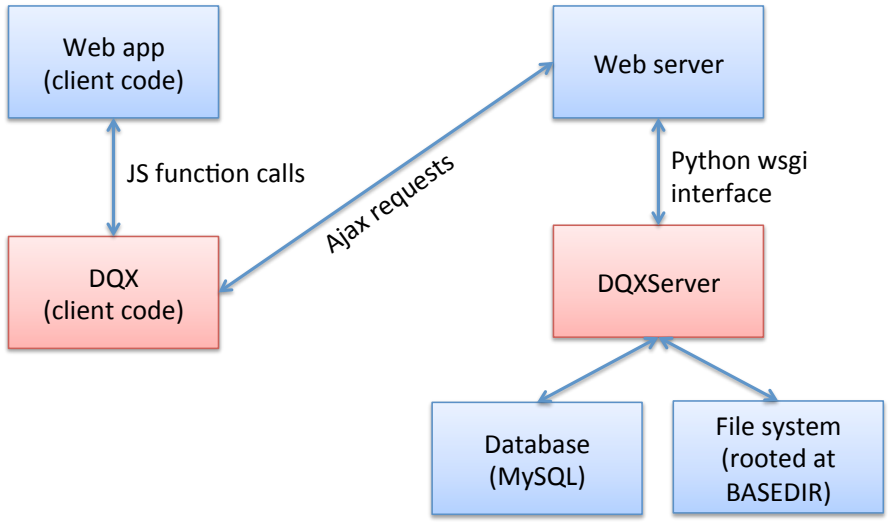


Figure 1. Modules of a DQX web app.

Web app
Contains the code specific to a particular application. It creates the web application, relying on the functionality provided in DQX.

DQX
Contains client-side JavaScript code of the framework, running in the client browser. Contains the layouting mechanisms for the html page. Functionality in this JS code communicates with corresponding code in DQXServer through Ajax requests.

Web Server
Serves the static content and the JS code, and redirects the DQX Ajax requests to DQXServer through a wsgi interface.

DQXServer
Contains server-side Python code, handling requests from DQX. This layer communicates with one or more SQL databases, and a subset of the file system. Access parameters to the database and the file system are specified in config.py.

Database
Currently, MySQL is supported (using MySQL-Python). Currently, the framework supposes that full access to that database is acceptable (i.e. only public data should be served). It is planned to hook DQXServer to a single sign-on authentication system.

File system
DQXServer has read/write access to a subset of the file system, located in a directory called BASEDIR.

2.2 External libraries

DQX uses the following external libraries:

- **jQuery**: appropriate version is included in the distribution.
- **lodash**: idem.
- **d3**: idem
- **handlebars**: idem.
- **require.js**: used for packaging and delivering JS code as modules and handling dependencies. `require.js` and `async.js` have to be provided by the web application in the `scripts` directory (see DQXTest for correct version)

3 Sample application: DQXTest

DQXTest is a sample web application that utilizes the functionality in the DQX library in a simple way. Most of the functionality can be learned by looking at this sample application.

3.1 DQXTest structure

3.1.1 Html document

The DOM tree containing the GUI elements that make the application, is entirely built by the JS code. The html document (`main.html`) only contains statements for:

- Loading the DQX standard style sheet (`scripts/DQX/DQXCommon.css`).
- Setting a version variable (`versionString`), used to force cache updates when the version was changed.
- Initializing some JS libraries (Modernizr & Google Maps).
- Loading the startup script (`main.js`) through the RequireJS loading mechanism.
- Specifying an initial 'Loading page' animation.
- Specifying an empty div `DQXUtilContainer`. The app html code will be rendered to this div.

3.1.2 Main JS module

The JS script `main.js` contains the app initialization code. The app contains a single `Application` object, which is instantiated by loading the `DQX/Application.js` module. DQXTest calls the following functions on this object:

- `SetHeader()`, defining the content of the app header (the top bar of the page, which is persistent over all views).
- `addNavigationButton()`, adding a navigation button in the top right part of the header.
- `customInitFunction()`, attaching a custom function step in the app initialization scheme. In this step, some app-specific data is loaded from the server.

- `init()`, initializing the application after all the views have been defined (see 3.1.3)

3.1.3 JS modules for the views

The views of the app are defined by the JS source files in the `Views` directory.

- Each view is defined by an object that derives from `Application.View`.
- A view object implements the function `.createFrames()`, which defines the layout of the view (see 4.1).
- A view object implements the function `.createPanels()`, which initializes all the panels that will be displayed in frames on this page.

3.2 DQXTest deployment

3.2.1 Required software components

- Apache2 web server
- Python 2.7
- Mod_wsgi
- MySQL
- MySQL-Python

3.2.2 Code installation

- Copy `DQXServer` to a directory served by the web server.
- Adapt the settings of `DQXServer/config.py` to reflect the database connection settings and the location of local file storage (see 3.2.3)
- Make sure that the installed `DQXServer/app.wsgi` is recognized by the web server as a wsgi handler.
- Copy `DQXTest/webapp` to a directory served by the web server.
- Copy or link `DQX` to the subdirectory `scripts/DQX` of the installed `DQXTest`.
- In the `DQXTest` file MetaData.js, make sure that the line `MetaData.serverUrl=…` specifies the correct url for accessing the `DQXServer/app.wsgi` handler.

3.2.3 Required data components

Location of the sample data: WTCHG internal virtual machine 'panoptes' (129.67.45.41)

Database

The content of the database 'pfpopgen' should be copied to a MySQL database with the same name on the deployment server. The location of this database, as well as the access credentials should be specified in the DQXServer file config.py.

Files

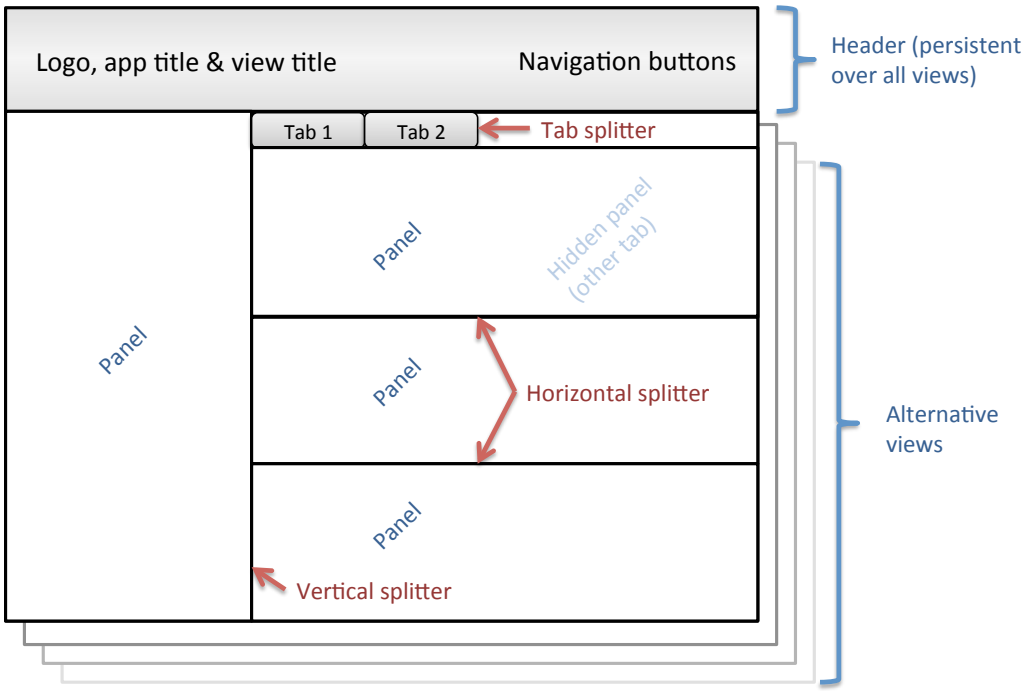
The content of the directory /mnt/storage/webapps/Tracks-PfPopGen2.1 should be copied to the **base directory** on the deployment server, under a subdirectory with name Tracks-PfPopGen2.1. The location of the

base directory should be specified in the DQXServer file config.py.

4 Major components of DQX

4.1 Page layouter

Example of a typical DQX web app layout:



At the highest level, the app structure is divided into a set of ***views***. Each view corresponds to a separate page, and navigation through the views is kept track of in the browser history. On top of each view, there is a persistent header that contains the application title, logo('s), and some persistent navigation buttons.

Each view contains a set of ***frames*** that are organized using a framework layouter. Each individual frame contains a client ***panel***, that manages a specific GUI element (e.g. a table, tree, list, genome browser, etc…).

The framework layouter relies on the hierarchical structuring of three types of layouter frames:

- **Vertical grouper**. Member frames are arranged in a vertical fashion, with draggable separators between each frame.
- **Horizontal grouper**. Member frames are arranged in a horizontal fashion, with draggable separators between each frame.
- **Tabbed grouper**. Only a single member frame is visible at a time, and set of tabs allows the user to control which one.

Note that these elementary types of layouters can be used to compose more complex layouts in a hierarchical fashion: a member frame of a layouter frame can be another layouter frame.

"Final" frames contain client panels holding GUI elements.

***NOTE**: The same layouter functionality based on frames and panels can be used in advanced popups (see 4.5).*

4.2 Types of panels

A panel object derives from FramePanel, and takes the frame object it is located in as a construction argument.

4.2.1 Framework.Form

Example: DQXTest/Views/FormDemo.js.

Holds a set of controls (see 4.3).

- Use `.addControl` to add a new control. Typically, a single layout control is inserted as the root element.

4.2.2 FrameList

Example: ???.

Creates a vertical list of items.

- Use `.setItems` to define the elements of the list.
- Use `.render` to update the visualization.

4.2.3 FrameTree

Example: ???.

Creates a tree-structured set of items

- Each item in the tree is of the type `FrameTree.Branch`.
- Use `.addItem` to add a a subbranch to a branch.
- Use `.root.addItem(FrameTree.Branch(…))` to add a branch to the root of the tree.
- `FrameTree.Control` is a special branch object that can be used to encapsulate a control in a tree item.
- Use `.render` to update the visualization of the tree.

4.2.4 Framework.TemplateFrame

???

4.2.5 FrameCanvas

Example: ???.

Encapsulates a canvas drawing element as a panel.

- Implement `.draw` to define the drawing of the canvas.
- Call `.invalidate` to force the redrawing of the canvas.
- By default, the canvas automatically resizes to exactly fill the space provided by the panel. Alternatively, a fixed width can be set using `.setFixedWidth`. In that case, a scroll bar will automatically appear whenever necessary.

4.2.6 GMaps.GMap

Example: DQXTest/Views/MapDemo.js.

Encapsulates a Google Maps view.

- Construction takes the initial position (as a `Map.Coord` object) & zoom factor.
- Various kinds of overlays can be defined for the map (point sets, SVG drawing elements, …).

4.2.7 QueryTable.Panel

Example: DQXTest/Views/TableViewer.js.

Encapsulates a table view that visualizes the content of a (queries) table in the server database.

- Takes a `DataFetchers.Table` object as construction argument (see ???).
- Use `.getTable()` to return the underlying table representative object.
- Use `.getTable().createTableColumn()` to add a column to the table (takes a `QueryTable.Column` object).
- Use `.getTable().queryAll()` to start with a query that returns all records from the table.
- Use `.createPanelAdvancedQuery()` to create a panel that shows an interactive graphical query builder, determining the query used for the table.

4.2.8 QueryBuilder.Builder

Example: DQXTest/Views/TableViewer.js.

Encapsulates a graphical, interactive query builder, to be used in conjunction with `QueryTable.Panel` (see 4.2.7).

4.2.9 ChannelPlotter.Panel

Example: DQXTest/Views/GenomeBrowser.js.

Encapsulates a Channel Plotter: a set of vertically stacked plots (called channels) that can be scrolled and zoomed horizontally, and share a common X-axis.

An important class derived from this is `GenomePlotter.Panel`, which specializes this into a genome browser. The horizontal axis is the genomic position (see ???).

- Use `.addChannel()` to add a new channel to the plot.
- Use `.channelModifyVisibility()` to show or hide a specific channel.
- Use `.setPosition` to specify a new scroll position & zoom factor.

4.3 Forms Controls

Example: DQXTest/Views/FormDemo.js.

DQX contains classes that encapsulate most commonly used html form controls, such as edit boxes, check boxes, lists, etc. These controls can be used in a number of places, including Form Panels (4.2.1), FrameTree branches (4.2.3), and Wizards (4.6).

4.3.1 Layout controls

A number of compound controls assist in layouting individual controls.

`Controls.CompoundHor`: Arranges members in a horizontal fashion.

`Controls.CompoundVert`: Arranges members vertically.

`Controls.CompoundGrid`: Arranges members on a 2D grid.

On compound controls, `. setLegend()` can be used to surround the member controls with a box, having a title.

`Controls.AlignRight`: Aligns a single member control to the right.

`Controls.ShowHide`: Allows the application to control the visibility of a single member control.

`Controls.HorizontalSeparator`, `Controls.VerticalSeparator`: Introduce an empty space in the horizontal or vertical direction (can be used in the list of members of a compound control).

Note that layout controls are controls by themselves, and hence can be nested to create more complex designs.

4.3.2 Controls containing data

A large set of controls contains a state that can be modified by the user who interacts with the control (e.g. checked state of a check box). These controls follow some common principles:

- The construction function takes an ID (which can be null), and an object with a set of additional properties that depend on the type of control.
- Use `.getValue` to obtain the current state of the control (note that this function can only be called when the control is live in the DOM tree).
- Use `.modifyValue` to modify the state of the control.
- Use `.setOnChanged` to attach a function to the control that will be called when the state changes.
- Use `.modifyEnabled` to change the enabled/disabled condition of the control.
- Use `.setHasDefaultFocus` to specify this control to have the focus when the form goes live.

`Controls.Check`. State: Boolean.

`Controls.Edit`. State: string.

`Controls.Button`. No state. Clicking the button triggers an OnChange event.

`Controls.LinkButton`. No state. Clicking the button triggers an OnChange event.

`Controls.HelpButton`. No state. Clicking the button loads a help page.

`Controls.Hyperlink`. No state. Clicking the button triggers an OnChange event.

`Controls.Combo`. State: current selection.

`Controls.RadioGroup`. State: currently active item.

`Controls.List`. State: current selection.

`Controls.ValueSlider`: State: value (slider position).

`Controls.FileUpload`. No state.

`Controls.ColorPicker`. State: selected color (`DQX.Color` object, see ???).

`Controls.Html`. State: html content.

4.4 Simple Popups

Example: DQXTest/InfoPopups/SnpPopup.js.

`Popup.create()` creates a simple popup box, with a specified title and html body. The function returns a unique ID, that can be used as an argument for `DQX.ClosePopup()`.

This functionality is typically used for simple transient popups, that block the flow of the application and must be closed to continue (e.g. showing a message that must be acknowledged).

4.5 Advanced popups

Example: DQXTest/InfoPopups/PopupFrameDemo.js.

`PopupFrame.PopupFrame` instantiates an object that encapsulates a more sophisticated popup. This function takes a popup type identifier, that is used to remember specific settings, such as position and size. In this way, reopening the same type of popup causes it to be present on the same place where the user last closed it.

A PopupFrame contains a layouting mechanism identical to a view in the app, but confined to the popup box. As such, it can contain any set of panels described in 4.2, arranged using the frame layouter mechanism described in 4.1.

`PopupFrame.PopupFrame` is always used as a derived class, implementing the functions `.createFrames()` and `.createPanels()`. The popup is rendered to the DOM and visualized by calling `.create()`. Calling `.close()` destroys the popup.

The method `.onClose()` can be overridden to be notified when the popup is closing.

4.6 Wizards

Example: DQXTest/InfoPopups/WizardDemo.js.

A wizard is a transient, blocking popups that presents a sequence of pages where the user can navigate forward and backward through, each view consisting in a set of controls. At the last page of the wizard, there is an OK button to validate the choices. At each page, there is a Cancel button to back out from the wizard, and the action that would be triggered when the wizard completes.

A wizard is created and executed using the following sequence of steps:

1. A wizard object `wz` is instantiated using `Wizard.Create()`.
2. Pages are added to the wizard using `wz.addPage()`. A unique ID of the page is provided, as well as a control element that will be displayed on the page (typically, this will be a compound layouting element, see 4.3.1).

3. The wizard is executed by calling `wz.execute()`.

NOTES:

- For each page, a validation can be executed prior to going to the next page (or completing), by providing a member function `reportValidationError`. If this function returns a non-empty string, it is treated as an error condition.
- During execution, the wizard can be forced on a specific page using `wz.jumpToPage()`.
- After completion of the wizard, the states of the controls on the pages can be queried by `wz.getResultValue()`. To this end, a non-default control ID must be provided for controls that will be queried in this way (see 4.3.2).

5 Query Table

???

6 Genome Browser

???

7 Custom server functions

???

DQX.customRequest

7.1 Asynchronous custom server tasks

???

7.2 Getting / setting custom server data

???

DQX.serverDataStore

DQX.serverDataFetch

8 Other utilities

???

8.1 Server data getters

???

8.2 Custom server functions

8.3 Text interpolation

???

8.4 Color

???

DQX.PersistentAssociator

8.5 Messaging

???

8.6 Touch events

???

DQX.augmentTouchEvents

8.7 Smaller utilities

???

DQX.highlightText