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

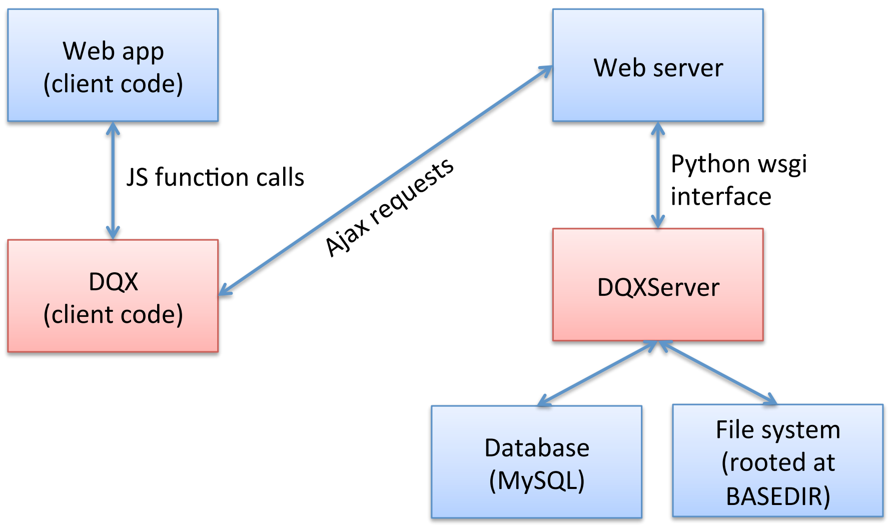
DQX is aimed at the creation of rich, desktop-like web apps with a single page interface (SPI). The SPI approach allows for a smooth, fluid user experience, and

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



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 Deploying DQXTest

3.1 Required software components:

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

3.2 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 further)
- 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.

3.3 Required data components:

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

3.3.1 Database

The content of the database ‘pfpopgen’ should be copied as 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.

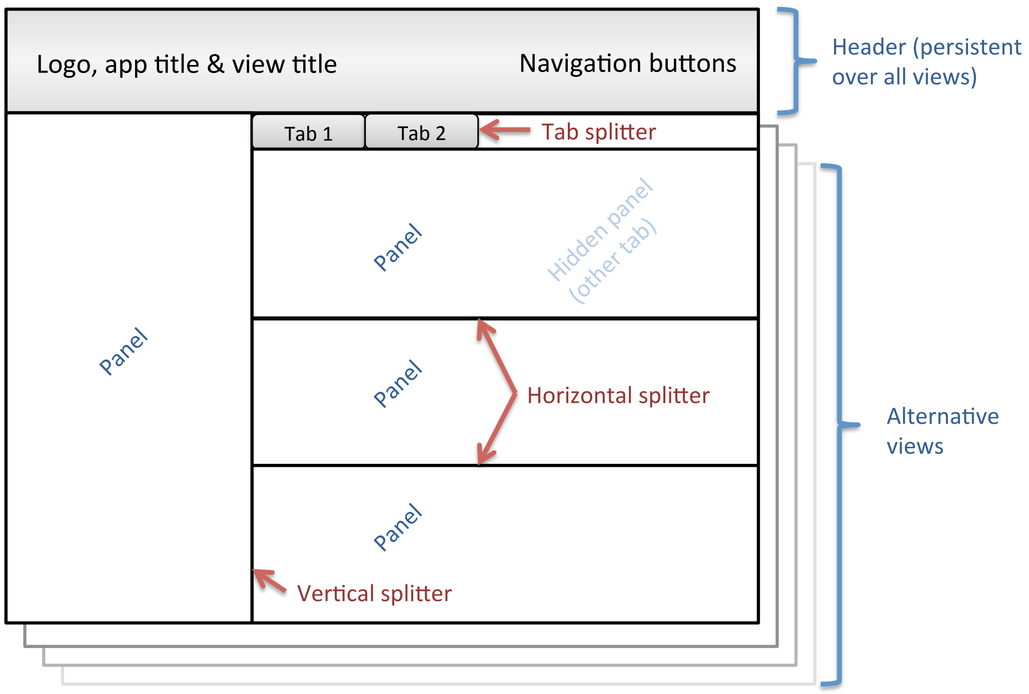
3.3.2 Files

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 layouting

Example of a 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.

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

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

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

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

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

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

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

4.2.9 ChannelPlotter.Panel

(+derived: GenomePlotter.Panel)

4.3 Forms and Controls

4.3.1 Layout controls

4.4 Popups

4.5 Wizards

5 Other utilities