Table of Contents

Visualisation Review

This document is the output of the Visualisation working group: David, Jacobo and Ivan.

As requested in the WLCG monitoring consolidation meeting 18 July 2013, we attempt to answer the following questions regarding visualisation:

- Are we doing things consistently for all of our applications?
- If we are doing things in multiple ways, is there a good reason for it?
- Are there any new technologies that would help us here?
- How long would it take to change?
- How would that impact the other layers?

Are we doing things consistently for all of our applications?

No. Here are the main differences we found across our multiple applications:

- Different backend frameworks: django, dashboard
- Some applications request data through ajax, some others request rendered html fragments
- Datatables is not used consistently across all tabular views
- MVC framework is not used consistently, or not used at all sometimes
- Plots come from different sources: Google Image Charts API, Google Charts API, Highcharts, Graphtool, Raphael
- Content is rendered using different technologies: XSLT, HTML skeleton, django templates

In this table, we summarise the design and technology used for the key aspects of visualisation by the various WLCG monitoring applications.

Application	Xbrowse-based: DDM2 WLCG Transfers XRootD FAX / AAA (ATLAS & CMS)	Historical Views-based: Historical Views (ATLAS and CMS) DDM Accounting old CMS Task monitoring also fits here (apart from BBQ)	Hbrowse-based: Interactive View (ATLAS & CMS) User and Production Task Monitoring
Use cases	Read-only data in tables and plots.	Read-only data in tables and plots.	Read-only data in tables and plots. Kill jobs (User Task Monitoring only).
Design	Single page (no reload) + AJAX	Single page (no reload) + AJAX	Single page (no reload) + AJAX
Server-side			
> language	Python	Python	Python
> design	HTML + API	HTML + plots + API	HTML + API
> html	Single static skeleton.		Single static skeleton.

		Single dynamic page + fragments using XSLT.	
> plots	n/a	Graphtool (matplotlib)	n/a
> api	JSON / XML serialised from Python object.	CSV / JSON / XML serialised from Python object	JSON / XML serialised from Python object.
> session state	n/a	n/a	Client-side generated id stored in database, with N hours expiry.
> key libraries	Dashboard-web	Dashboard-web, Graphtool	Dashboard-web
Client-side			
> languages	JavaScript	JavaScript	JavaScript
> design	Uses Xbrowse (in-house MVC) • Model and views are JS objects. • Model holds view state and app data. • Views push UI control changes to model. • Controller synchronises model with URL #hash • Views listen for model change events. • Views load data as JSON via AJAX, and render tables and plots. • Pluggable views.	No MVC framework • UI controls hold view state. • Submit button pushes view state to URL #hash. • URL hashchange event triggers UI controls update, AJAX request to create temp plots, write HTML image tags, and load HTML fragments via AJAX.	Uses Hbrowse (in-house MVC hierarchal view specific) • Model is JS object. View supported hierarchical data. • Model holds view state and app data. • View pushes UI control changes to model. • Controller synchronises model with URL #hash. • Controller notifies view of URL change • View loads data as JSON via AJAX, and renders tables and plots • Single hierarchical view configurable via settings file.
> view state	Stored in model until pushed to URL #hash using BBQ	Stored in UI controls until pushed to URL #hash using BBQ	Stored in model until pushed to URL #hash using BBQ
> html	jQuery DOM manipulation + Handlebars templates	JavaScript DOM manipulation	jQuery DOM manipulation.
> plot	Highcharts + custom xplot jQuery plugin	n/a	Highcharts and/or Google charts
> ajax	JSON from API	HTML fragments	JSON from API
> key libraries	jQuery, Xbrowse, BBQ, Highcharts, Datatables,	Dojo, jQuery, BBQ, Highslide, Datatables.	jQuery, Hbrowse, BBQ, Highcharts, Google

	Handlebars, jQuery-layout, underscore, Google maps API (for FAX and AAA)		Charts API, Datatables.
Integration			
Bookmarking / Browser history	Yes	Yes	Yes
API	JSON and XML via HTTP. Links on all pages.	CSV, XML, JSON. Links on all pages.	JSON and XML via HTTP. Links on all pages.
Export plots	PNG, JPEG, PDF, SVG via Highcharts server (not perfect)	PNG.	PNG, JPEG, PDF, SVG via Highcharts server (not perfect)
Embed plots	Yes, using iframe. Links on all plots.	Yes, using image tag. Links on all plots.	No

========== MyWLCG related ==========

Application	MyWLCG A/R	MyWLCG Monthly Reports	MyWLCG Trends
Use cases	Read-only data in tables and plots.	Read-only data in plots, PDF and CSV formats	Read-only data in plots.
Design	One page per tab (full page reload on submit) + AJAX.	One page per tab (full page reload on submit) + AJAX.	One page per tab (full page reload on submit) + AJAX.
Server-side			
> language	Python	Python	Python
> design	HTML + plots + API	Plots retrieved from other MyWLCG applications then included in HTML using Django template, which is converted to PDF using Pisa.	HTML + JSON for plots + API
> html	Dynamic pages + fragments using Django templates	Static pages + django json templates	Dynamic pages using Django templates
> plots	Server requests plot from Google Image charts service and serves to client	Plots retrieved from other MyWLCG applications.	Google Charts API
> api	JSON / XML from Django templates. Not used by UI.	CSV / JSON / XML from Django templates. Not used by UI.	JSON / XML from Django templates. Not used by UI.
> session state	n/a	n/a	n/a
> key libraries	Django, Google Image charts.	Django, Pisa	Django, Google Charts
Client-side			
> languages	JavaScript	JavaScript	Javascript
> design	No MVC framework • Each submit triggers a page get with view state parameters in the URL. • HTML loads filters through	No MVC framework • Each submit triggers a page get with view state parameters in the URL. • HTML loads filters through	No MVC framework • Each submit triggers a page get with view state parameters in the URL. •HTML loads filters

	AJAX. • On page load, data content is loaded as HTML fragments via AJAX, using Datatables.	AJAX. • For web view: On page load, plots are requested through AJAX to A/R view • For PDF: served directly	through AJAX •On page load, data content is loaded as JSON via AJAX, using jQuery, and rendered as plot using Google charts API.
> view state	URL parameters	URL parameters	URL parameters
> html	n/a (Datatables takes care of this).	n/a	n/a (Google charts API takes care of this)
> plot	n/a	Inserted from A/R fragments	Google charts API
> ajax	HTML fragments	Plots	data (JSON)
> key libraries	jQuery, Datatables	jQuery.	jQuery, Google Charts API
Integration			
Bookmarking / Browser history	Yes	Yes	Yes
API	JSON and XML via HTTP. No links.???	CSV, JSON and XML via HTTP. Link for CSV. ???	CSV, JSON and XML via HTTP. Link for CSV. ???
Export plots	PNG	PDF.	PNG using canvas
Embed plots	Yes, but not well supported	Yes, but not well supported. Links to PDF.	No

Provided by Django:

Poem provides an admin interface enterily built within django's admin site. It allows:

- Read access for all profiles, metrics, fqans (anonymous + identified users)
- Write access for those identified by certificate

==========Site/Services monitoring==========

Application	SSB	SUM	CMS Critical Services
Use cases	Read-write data in tables and plots.	Read-only data in plots.	Read-only data in table.
Design	• Index: Single page (full page reload on submit) • Expanded Table: Single page (no reload) + AJAX • History: One page per tab (no reload) + AJAX	One page per tab (no reload) + AJAX	Single page (no reload) + AJAX
Server-side			
> language	Python	Python	Python
> design	HTML + API	HTML + API	HTML + API
> html	Skeleton pages using XSLT	Skeleton pages using XSLT.	Single static skeleton.
> plots	n/a	n/a	n/a
> api	JSON serialised from Python object.	JSON / XML serialised from Python object.	JSON / XML serialised from Python object.

> session state	n/a	n/a	n/a
> key libraries	Dashboard-web	Dashboard-web	Dashboard-web
Client-side			
> languages	JavaScript	JavaScript	Javascript
> design	Index: • Each submit triggers a page get with view state parameters in the URL. Expanded Table: In-house MVC • View state in model synchronised with URL #hash. • View and Data loader listens for model change events. History: • UI controls hold view state. • Submit button pushes view state to URL parameters and loads JSON data via AJAX to render plot.	No MVC framework • UI controls hold view state. • Submit button pushes view state to URL #hash. • URL hashchange event triggers UI controls update, and loads JSON data via AJAX to render plot.	No MVC framework • Static HTML page • On load, AJAX request for JSON data and HTML rendered. • No UI controls.
> view state	Index : URLS parameters Expanded Table: Stored in model until pushed to URL #hash using BBQ History : Stored in UI controls until pushed to URL #hash using BBQ	Stored in UI controls until pushed to URL #hash using BBQ	n/a
> html	jQuery DOM manipulation	JavaScript DOM manipulation	jQuery DOM manipulation and jit-treemap.js
> plot	Highcharts, Custom heatmap using Raphael	Custom heatmap using Raphael	Plot is really HTML divs generated using jit-treemap.js
> ajax	JSON from API	JSON from API	JSON from API
> key libraries	jQuery, BBQ, Highcharts, Raphael, Datatables.	jQuery, BBQ, Raphael	jQuery, jit-treemap.js
Integration			
Bookmarking / Browser history	Yes	Yes	n/a
API	Index: No Expanded Table: CSV, Excel, PDF. History: JSON and XML. Links for JSON.	JSON and XML. Links for JSON.	JSON and XML via HTTP. No links

Export plots	PNG, JPEG, PDF, SVG via Highcharts server (not perfect)	No	No
Embed plots	No	Yes, using iframe tag. No links.	No

If we are doing things in multiple ways, is there a good reason for it?

No. The main reason in all cases seems to be historical.

Are there any new technologies that would help us here?

Based on the review and discussion with developers, we would suggest the following best practice approach for our applications:

Server :

- Serve one HTML skeleton per UI.
- REST API serving JSON data for UI AJAX requests.
    - Django seems the obvious choice but may not fit well with old db schemas or NoSQL.
    - Ideally a framework. Failing that object serialisation. Not string manipulation or templates.
    - Need investigating: django-tastypie, django-rest-framework, …
- Server side caching where appropriate
-
    - memcache as django cache backend
    - Varnish as HTTP proxy

Client :

- Common library + plugins
    - jQuery seems the obvious choice.
- Common MVC :
    - No obvious choice.
    - Ideally open-source. Failing that a common in-house solution. Not multiple in-house solutions.
    - Must support URL synchronisation for bookmarking and browser history.
    - Must support pluggable views.
    - Need investigating: Backbone.js, Angular.js, Ember.js …
- Plots: Interactive, Exportable, Embeddable :
    - Highcharts seems the obvious choice but for some plots other libraries will be needed.
    - Need investigating: d3.js, …
- HTML:

- jQuery DOM manipulation seems the obvious choice but it can be verbose and hard to maintain.
- Ideally an templating solution can be found. This may be linked to the MVC.
- Need investigating: Handlebars.js, …

How long would it take to change?

How would that impact the other layers?

If best practices are followed, the impact on other layers should be minimal:

- Visualization is a top layer
- Most backends already provide JSON based REST API

This topic: LCG > VisualisationReview
Topic revision: r10 - 2013-08-02 - IvanDzhunov