The PhEDEx next-gen website

R Egeland[a] [1], C-H Huang[b], P Rossman[b], P.Sundarrajan[a] and T Wildish[c]

[a] University of Minnesota, Twin Cities
[b] Fermi National Accelerator Laboratory
[c] Princeton University

E-mail: awildish@princeton.edu

Abstract. PhEDEx is the data-transfer management solution written by CMS. It consists of agents running at each site, a website for presentation of information, and a web-based data-service for scripted access to information.

The website allows users to monitor the progress of data-transfers, the status of site agents and links between sites, and the overall status and behaviour of everything about PhEDEx. It also allows users to make and approve requests for data-transfers and for deletion of data. It is the main point-of-entry for all users wishing to interact with PhEDEx.

For several years, the website has consisted of a single perl program with about 10K SLOC. This program has limited capabilities for exploring the data, with only coarse filtering capabilities and no context-sensitive awareness. Graphical information is presented as static images, generated on the server, with no interactivity. It is also not well connected to the rest of the PhEDEx codebase, since much of it was written before the data-service was developed. All this makes it hard to maintain and extend.

We are re-implementing the website to address these issues. The UI is being rewritten in Javascript, replacing most of the server-side code. We are using the YUI toolkit to provide advanced features and context-sensitive interaction, and will adopt a Javascript charting library for generating graphical representations client-side. This relieves the server of much of its load, and automatically improves server-side security. The Javascript components can be re-used in many ways, allowing custom pages to be developed for specific uses. In particular, standalone test-cases using small numbers of components make it easier to debug the Javascript than it is to debug a large server program.

Information about PhEDEx is accessed through the PhEDEx data-service, since direct SQL is not available from the clients' browser. This provides consistent semantics with other, externally written monitoring tools, which already use the data-service. It also reduces redundancy in the code, yielding a simpler, consolidated codebase.

In this talk we describe our experience of re-factoring this monolithic server-side program into a lighter client-side framework. We describe some of the techniques that worked well for us, and some of the mistakes we made along the way. We present the current state of the project, and its future direction.

1. Introduction

PhEDEx[1] (Physics Experiment Data Export) is the data-placement management system developed by CMS for managing experiment data on the grid. The PhEDEx website[2] was originally written several years ago to allow users to interact with and monitor the activity of

[1] now at Montana State University

PhEDEx. While it has been extended from time to time, it still reflects its architectural origins as a single Perl program, and is becoming increasingly difficult to maintain.

It is also hard to integrate code written for other parts of the project, such as the PhEDEx data service[3] (which provides HTTP-based access to PhEDEx information, such as status information or details of physics data known to PhEDEx). Over the years it has diverged from the rest of the PhEDEx codebase. This has led to duplication of code and the risk of having different representations of the same data, once on the website and once in the data service. Obviously, this can be confusing to users. The difficulty of re-using code in the website also means that the full richness of information now available through the data service cannot be exploited by the website, at least not without extensive effort.

Users are now accustomed to responsive web applications, rather than the static HTML pages of the existing PhEDEx website. Graphs are rendered as images, with no interactivity. Cross-page navigation remembers only a few coarse options on filtering the data that is shown, so it is hard to drill down into the data in great detail.

All this points to the need to re-implement the website. The obvious technology choice was javascript, and we adopted the YAHOO User Interface framework (YUI[4]) as the basis for the project. We refer to this new implementation of the website as *the Next-generation website*, to distinguish it from the older generation existing site. The project began with two core PhEDEx developers working on the javascript framework and one on the extra data service support needed to provide the extra interactions with the PhEDEx database.

2. Application architecture

We decided early on to explore the application architecture before embarking on a serious prototype. We felt this was the best way to proceed for a couple of reasons. One is that we had very little experience with javascript before then, so we wanted to be sure we were doing something sensible. The other is that we wanted to have a solid base of code as soon as possible, so that anything we threw out later would not have too widespread an impact.

We adopted the principles described by Nicholas Zakas in his presentation on Scalable Javascript Application Architecture[5][6]. The next-gen website architecture[7] therefore consists of three main components, the *sandbox*, the *core*, and a set of *modules*. Modules can be enhanced with a set of *decorators*, to further customise their behaviour and abilities. This architecture is illustrated in figure 1.

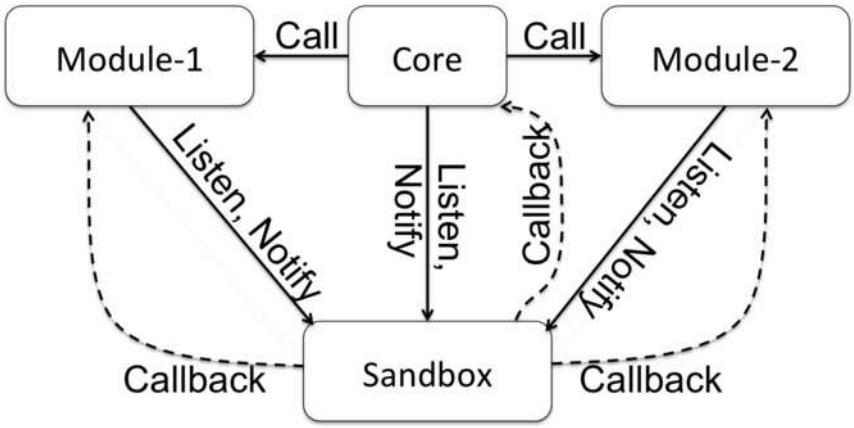

Figure 1. The javascript architecture of the next-gen PhEDEx website. Modules interact only via the sandbox, the core can interact via the sandbox, or can invoke module methods directly.

2.1. The sandbox

The sandbox is, as its name suggests, used to limit the degree of coupling between other parts of the application. It is a simple event-dispatcher that allows modules to subscribe to event notifications (to *listen* to them) and to raise events to be handled by other modules (to *notify* them of the event). The sandbox creates a new YUI event the first time it sees a notification, so the modules do not need to worry about such details. It maintains a list of which modules (or more accurately, which functions) are listening to a given event, and automatically cleans up when no further listeners are attached to a given event.

Having a sandbox to do this simplifies the module code. Modules do not need to concern themselves with creating event handlers and worrying about issues of variable-scope, the sandbox takes care of this. Modules do not need to know where a notification will come from or who is listening to the notifications they send, the sandbox insulates them from that. Coupling between modules is minimised, several modules can exist on the page at the same time and co-operate by sending notifications that other modules will respond to, but no module is ever directly aware of which other modules exist.

Notifications consist of an event name (a string), and an arbitrary series of parameters. The event name is used as the key for listeners when they subscribe, the extra parameters then give details about the specific event. So for an event named 'changeNode', there would be a second parameter which gives the name of the node[2] to change to. For a more complex event, the parameters can be complex objects, with multiple attributes. A 'getReplicaInformation' event, to find out details about a copy (replica) of a dataset at a given site, could have a parameter which consists of an object with 'nodeName' and 'datasetName' properties, to define which replica to get information for.

A single sandbox is shared by the entire application. It is implemented as a singleton, so modules that need it can simply call the sandbox constructor to get the current instance.

2.2. The core

The core is the heart of the application. It drives the modules through their lifecycle, creating them as needed, invoking methods on them to change their state, and destroying them when they are no longer needed. The core contains the bulk of the application logic, everything that is not specific to a given module.

The core can also cause modules to respond by signaling them through the sandbox. This allows for more generic interactions, where several modules may wish to respond to a single event. Typically the core will invoke a method on a module, or notify the sandbox to signal an event that one or more modules will listen for. It will then listen for a callback from the sandbox saying that the module has sent a notification that the action has been performed. It will then invoke or notify further actions, until the desired application state is achieved. This would normally mean that the application is waiting for user input in some form.

The core is not directly responsible for any interaction with the user interface, and does not manipulate the DOM in any way. That is taken care of by the modules.

The core also exists as a single instance for each application page. There is no technical reason why multiple cores could not be developed, each capable of managing different types of module. This would require abstracting the behaviour of a set of modules into a new core, respecting the architectural constraints on communication and method invocation. In practice, this has not been necessary for PhEDEx, but it does open the possibility of sharing the same architecture with modules from different projects within CMS.

[2] A 'node' in PhEDEx corresponds roughly to a site at which data resides. The correspondance is not always exact, for technical reasons of the implementation.

2.3. Modules

Modules implement the specific behaviour of the application. Many modules exist for the on-screen representation of items of interest, with HTML controls to allow user-interaction. One module represents information about the nodes known to PhEDEx, and their associated properties. Another displays information about the status of links between nodes. Yet another module displays information about datasets and files that reside at a given node. We have over 20 modules to date, each representing different information.

Modules do not interact with each other or with the core directly, they interact exclusively through the sandbox, notifying their own state changes and listening for notifications that they have to respond to. Most modules are completely self-contained, not needing other modules to function with them, but they can still co-operate with other modules if they are there.

Thus one module might implement controls for selecting the name of a node from the PhEDEx database. When a node is selected, it will notify the core of the newly selected node-name. Other modules on the page may be listening for a notification of a node-name change, and can then update their internal state, fetching data from the data service if necessary. In this way, modules exist completely decoupled from each other, so they can be written and debugged independently. They need only agree on the notification specification, but they don't care where the notification comes from. It could come from a visible HTML control, selected by the user, or from a URL parameter in a bookmarked link, or it can be generated by a javascript timer to make the page cycle through a series of states automatically.

Modules can listen for any number of events. Typically they listen for generic messages from the core and for messages directed to them specifically by their unique ID or name. The event handler, the function that listens for the notifications, can examine the parameters that come with the event to decide exactly how to process the event, and can then invoke methods from the module which in turn may lead to notifications that the module has done something.

Modules may listen for the same notification with more than one event handler, which allows the module to be segregated into sub-objects if so desired. Each sub-object can perform a specific action based on the values of the parameters that come with the notifications. This is particularly useful for implementing the *decorators*, described below.

Modules which present data to the user are predominantly of two sub-types, *datatable* modules2 and *treeview* modules3. Datatable modules use the YUI DataTable widget[8] as it stands, while modules representing treeviews use a much-enhanced version of the YUI TreeView widget[9]. Other modules use a custom widget to present themselves. We have established a three-level hierarchy for modules. There is a base class which implements the bare module behaviour and provides stub functions for features which may not always be used, a derived class each for datatable and treeview modules, which implements the generic behaviour of those specific types of modules (drag-n-drop re-ordering, sorting, selecting and highlighting, context menus etc), and finally the specific module that implements the details of a given data-view (e.g. a table of PhEDEx nodes, treeview of datasets, blocks and files).

Some modules do not interact with the DOM at all, but implement specific activity that is shared by other modules. One example is the module that interacts with the data service, fetching data on behalf of other modules, validating the returned object, and passing it back to the calling module. Another is the history module, which deals with encoding and decoding URLs that contain stateful information and with driving the transitions between states as the user navigates the browser history. These modules could be considered as extensions to the core, but since they are highly specific in nature, and in some cases optional for the page, we keep them separate from the core.

Figure 2. An example of a next-gen datatable, based on the YUI datatable. The decorators are the controls above the table itself.

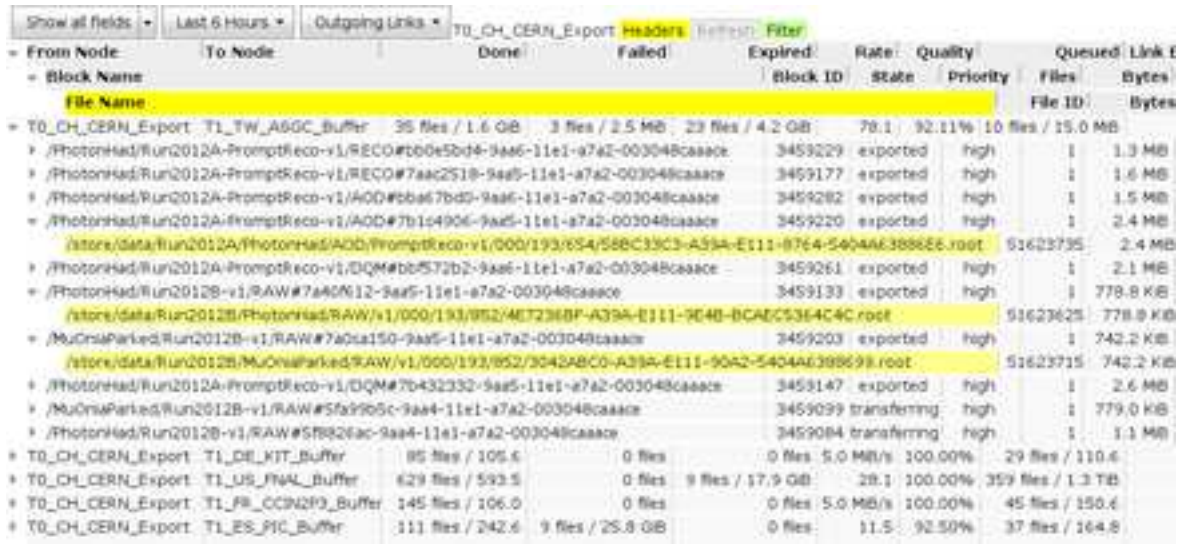

Figure 3. An example of a treeview module, showing the complexity of information that can be shown per-row, while still being able to drill down into lower-level objects.

2.4. Decorators

A decorator is an element that enhances the interaction with a given module by implementing specific functionality that is generic among modules, but is not strictly needed to allow a module to perform its work. As such, this functionality does not belong in the core, nor should it be reproduced in each module. The core takes care of instantiating the decorators for a module automatically, the module simply contains an array of metadata objects that describe what decorators it needs and what parameters they need to initialise themselves correctly.

Decorators exist in the context of a specific module instance, and cannot function without that context. So the rules for their interaction with the module are slightly relaxed. Where necessary, a decorator is allowed to directly invoke a method of the module it belongs to. This is strictly one-way, modules never invoke methods of their decorators since they may not have any, but a decorator always does have a module to interact with. More commonly a decorator simply notifies its module (via the core) that some action is needed.

Decorators function primarily by listening to events from the module they belong to, or by receiving input from the DOM elements they themselves create.

One example of a decorator is a refresh button which is disabled by default, and becomes enabled once the cache lifetime of the data represented by the module has expired. Users often want to refresh their view, but without knowing the how long the data will be cached on the server, they will waste time and network resources refreshing the entire page. The refresh button solves that problem by letting the user know exactly when new data will be available (via a tooltip), and by allowing them to fetch and display new data with the minimum of overhead.

The refresh button works by listening to notifications sent specifically to the module it is attached to, telling the module that it has received fresh data. It then takes the expiration header from the HTTP response and uses that to set a timer that will re-enable the button. When the button is enabled, clicking it will invoke the module 'getData' method, which notifies the core (via the sandbox) that it wants new data. The refresh button needs only to know which module it is attached to, and where in the DOM to render the button. The module itself knows nothing about the refresh button, beyond the metadata declaration that it should exist. See figure 4.

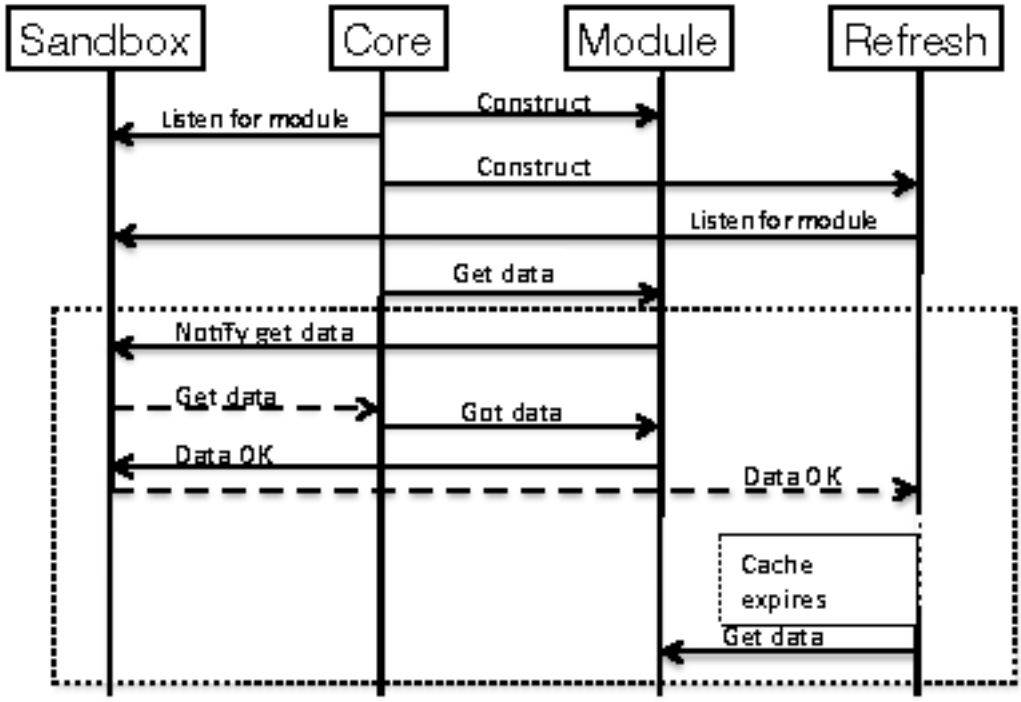

Figure 4. Simplified control flow between the refresh-button decorator and the module it belongs to. The core constructs the module and listens for notifications from it. It constructs the refresh decorator, which also listens for notifications from the module. The core tells the module to get data. The module notifies the sandbox what it needs, and the core fetches it (using the Datasvc module, omitted for clarity). The core then tells the module it has data, the module processes it to verify its validity, and notifies that the data is OK. The refresh decorator receives this notification, and sets its expiry timer. When the timer expires, and the user clicks the now-active refresh button, the refresh decorator tells the module to get data. The sequence of events in the dashed box then repeats itself.

Another decorator implements a filter panel. This uses metadata in the module to build a panel that allows the user to filter on any field of the data the module represents, with additional logic to support ranges for numerical fields and other features. By factorising this component, we eliminate a lot of extra coding in each module, reducing it to a simple declaration of a metadata structure. There are two forms of this decorator, one for the datatable views, one for treeviews. The core loads the appropriate type for each module.

Other decorators implement DOM controls, such as menus and buttons which allow the user to interact with the module. Because the module does not implement the controls directly, it is easy to change the representation of a given control (e.g. to replace a menu with radio buttons) without having to make significant changes to the module. Because decorators are defined by the module configuration, it is easy to remove them from pages where appropriate, e.g. from pages that may be on display on a public terminal somewhere.

2.5. The YAHOO User Interface framework

The YAHOO User Interface framework (YUI[4]) was chosen as the basis for the next-gen website. YUI is an open-source javascript framework which provides extensive documentation and examples, and has a complete library of features. From version 2, the features we made most use of are:

- the Datatable module, which represents tabular data as an HTML table. The table has sortable columns, and columns can be re-ordered by drag-and-drop.
- the Treeview module, which represents an 'explorer-like' treeview of more structured data
- the Autocomplete module, which allows context-sensitive auto-completion of form fields, based on partial input from the user
- the Menu, Button, and Calendar modules, for various navigation and interaction features
- the Connection Manager, for communicating with the data service
- the History module, for in-page bookmarking of state
- the Loader module, which allows dynamic loading of page components
- the Logger and Profiler nodules, for debugging

At the time the next-gen website project was initiated, YUI version 3 was not yet released to the public, and we used the then current version, version 2. Version 3 has been re-designed from the ground up by the YAHOO developers, using the lessons learned in implementing version 2. As such, it is not backwards compatible, and some effort will be needed to port the application forward. For this reason we have not yet adopted version 3. However, versions 2 and 3 can co-exist in the same application, so we expect to be able to migrate in a non-disruptive manner.

3. The first prototype

We decided to start the project by producing modules for information that was not available at that time in the existing website. We intended to supplement, and eventually replace, the existing website completely, and thought the best way to encourage early adoption would be to present new views or data that was not available by other means.

We worked with the operations teams to define a set of views that would be useful to them, prioritised them according to their importance to the operations staff and the difficulty of implementation, and targeted them for the first prototype. Not all of these views made it into the first prototype, some of them actually required changes to the schema and agents in order to implement them efficiently, otherwise they would have required large table-joins that would have impacted performance of the database.

Nonetheless, we implemented a number of these views, presenting the data as either tables or treeviews, whichever best matched the properties of the data. Context-sensitive menus were

provided, allowing users to right-click on a data-field and to access a menu of options related to that data type, so they can drill-down deep into the data. Sorting and filtering were implemented, with filtering options more advanced than was available in the existing site.

The prototype, in its first release, therefore had a rich set of data that the user could explore, but not much overlap with the existing site. The look and feel were completely different, as the next-gen site had been designed from scratch around access to the data, with no regard to the legacy of the existing site.

3.1. Users' reaction to the prototype

As could probably have been predicted, some users liked the new prototype, but many did not. The people who liked it tended to be those who most wanted the new views that had been implemented, those who liked it least tended to be those who were happy with the information from the existing site. They did not find the extra information and capabilities intuitive or easy to grasp. However, the distinction between the two ends of the spectrum was by no means sharp, some people who were involved in the consultation were not happy with the prototype while some people who were not involved until the prototype was released were quite happy with it.

Many people responded to the prototype by requesting features which had in fact already been implemented, which demonstrates that we had over-estimated how intuitive the interface would be for users. That came as a surprise, since we expected that people would experiment, freely clicking on things to find out what might happen. Rather than spend time exploring the new site, it seems people came with a specific task in mind, and were expecting to be able to achieve it with no learning curve.

It also became clear that people approached the use of the prototype website with different pre-conceptions. Some expected it to be a full replacement for the existing website already, or to have a basic navigation pattern similar to the existing website. We had essentially shifted the paradigm from a web page to something more like a desktop application, designing the navigation around the data, whereas people want to navigate according to the problem they wanted to solve. Clearly we failed to manage the users' expectations of what was clearly a prototype, not yet a finished product, with the result that many of them were disappointed by what they saw.

Overall, user feedback was not positive. It became clear that we, the developers, had interpreted the needs of the users in ways which were very different to theirs. Even some experienced users didn't understand the logic behind the next-gen site. For many, they only wanted what they needed to perform a specific set of tasks that they already understood. When faced with solving new problems via the next-gen site, or with a different way of solving problems they already knew how to handle, they found the richness of navigation options and information confusing. Many preferred simpler representations of the information, dedicated to specific tasks.

4. The user-experience workshop

To correct this, we decided to hold a user experience ('UX') workshop. The workshop ran for five days in December 2010. Nine people participated, consisting of two expert operations staff, six people who interact with the site on a regular basis in various capacities, and one PhEDEx developer responsible for the website.

The first two days were user-oriented, dedicated to analysing the results of a survey which was sent out to the collaboration, and brainstorming possibilities by considering existing websites that we know and like. The second two days were design-oriented, in which we made wireframe mockups of a selection of user-views to explore how they could be presented to the user. The final day was spent defining a workplan, based on the results of the first four days.

The survey was performed online, with volunteers from all parts of the collaboration. A total of 74 people took the survey, with 58 of them completing it. The results[10] show a mix of roles

in CMS, with some roles better represented than others, reflecting the relative importance of the PhEDEx website for a given activity. Users were asked questions about their routine activities and about their personal priorities for enhancements to the website. They were asked to list things they liked from the existing site, and to list things they disliked about it. They were also asked to list things they liked and disliked about the next-gen prototype, if they had used it.

It became clear from the survey results that people were concentrated on the few tasks they regularly perform, such as administering transfer requests, monitoring link performance, and examining problems at their own site. Most of them like the existing website enough that they do not want to see any major changes to it. Few of the complaints about the existing site justified a re-design from scratch.

There were a few who liked the next-gen prototype and the features it offered, though in general they were not satisfied with it because it did not provide everything that the existing site already had.

The main lesson from the survey, therefore, was that the users were largely satisfied with what they had, and did not want major changes, needing only evolutionary enhancements instead of something radically new.

The wireframing exercise was conducted in a well structured manner. We chose a number of existing website views, such as the page that displays information about existing data-subscriptions, or the form for creating new requests, and spent 30-60 minutes on each one. First, we independently sketched our idea for what that page could look like if it were to be re-implemented from scratch, restricting ourselves to a maximum of 5 minutes to produce our sketch. Then we discussed our ideas, one by one, to see what range of options were considered useful, how we would implement the details, how the user would interact with it, and so on. By sketching independently, we were able to maximise the breadth of ideas. By limiting the time for sketching we were able to spend more time in discussion rather than on individual elaboration of an idea.

After collecting a number of wireframe sketches for several pages, we assembled all the sketches together, and grouped them according to similarities and differences. This provided a cross-check that we were not looking to mix too many different styles or behaviours which would cause the site to be inconsistent internally. This was then used to guide us in our final choice of how to proceed.

The conclusion from the user-experience workshop was that, instead of providing a new, completely separate website, we would systematically enhance and augment the existing site. Pages would be replaced one by one with new pages which had equivalent functionality, but improved through the use of the next-gen framework. This would allow us to incrementally achieve our internal goals of eliminating redundant code and improving maintainability while simultaneously improving the users interaction with the site, and to do it without forcing the user to learn new habits or to change their expectations too rapidly.

5. The new next-gen website

We decided that, instead of presenting new views of the status and performance of PhEDEx, we would start with the most commonly used existing pages, and enhance them first. Since some of these pages were in need of an overhaul for other reasons (notably the change in schema for block-level and dataset-level subscriptions), we decided to start with those. As the new site gained acceptance, and users gained experience with it, we would be able to introduce new elements (navigation by context menus, new pages with more complex interactions) without causing such a shock for the user community.

Re-engineering the existing pages was relatively straightforward in concept. The existing website has separate page-handlers for each page, these handlers generate everything below the page head and navigation bar. Each page-handler was implemented as a next-gen module.

The existing website uses an internal template system, where the basic navigation elements and page contents are generated automatically, either by a specific Perl package for each page or by the template engine for generic features. We have retained the navigation elements for now, and are replacing the individual Perl packages one by one. To keep the old and new websites separated cleanly, we prefer not to insert javascript tags into each package of the existing website to load the corresponding next-gen javascript replacement. Instead, we replace the page contents with an HTML div element with the same name as the page URL.

A javascript function, loaded with the navigation elements in the page wrapper, executes once the page is loaded, for every page. It looks for an HTML div element with the same name as the page URL. If it doesn't find one, it does nothing, assuming that the page will continue to be managed by the Perl script on the server (the old website). If it does find a matching div element, it dynamically loads the javascript file that contains the module for that page, then creates a core, a sandbox, and an instance of the module that implements the page. The module performs its basic initialisation and notifies the core of its state. The core and the module then co-operate to manage the interactions with the user.

This prevents the next-gen javascript code from leaking back into the old website. The old website header is fixed, loading a new javascript file from the next-gen code, but the old website does not explicitly load different modules for different pages.

This makes it very easy to replace the pages one by one. All that is left is to re-implement the page behaviour in javascript, with whatever enhancements are foreseen. This is not so trivial, the advantage of static pages, generated entirely server-side, is that the flow of interaction is under much tighter control. Only one thing is happening at a time, and the user is not able to interact with the page while the server is responding to an action. With javascript, different elements of the page may load in different orders from one time to the next, and the user may be able to interact with the page before all elements are fully instantiated.

For example, whereas the static page would contact the database directly to get a list of nodes known to PhEDEx, then the list of groups, and then to check the users' access rights, and would delay rendering the page until it had all that information, the javascript-driven page uses the data service to get the same data. This causes an extra round-trip to the server for each piece of data, but since it can ask for them in parallel, it may actually be faster, benefitting from using multiple threads of the data service. It also allows the user to start interacting with the rest of the page earlier, so the perceived page-load time can be reduced.

The disadvantage of this approach is that the javascript code must be aware that the page state may not be fully consistent at all times, and that the order in which things happen is not guaranteed. Since a page like the transfer-request form can require several calls to the data service, there are several potential race conditions to handle. Overall, coding a page with responsive javascript is far more complex than coding a static page, generated on the server. Nonetheless, the advantages of a more responsive, interactive page far outweigh the extra complication of implementation.

The application can be made more responsive still by intelligent ordering of the way the page components are assembled. For example, data needed for a default view can be retrieved immediately the page has loaded, and the DOM elements drawn while waiting for the data from the server. Other elements, maybe dependent on the data-content, can be rendered in an appropriate form after the data is presented, rather than before. Minor enhancements, such as manipulating the DOM to insert summary messages or helpful links to perform necessary actions, are very easy to implement in javascript. Many such improvements suggest themselves over the course of time as users experience the new style of interaction.

Replacing server-side pages means that, where there is custom SQL on the server, we have to re-implement that SQL as a data service API, which can then be called from the javascript application. This has led to a number of enhancements to existing APIs as well, which benefits

the community of users who access PhEDEx information by scripts instead of interactively. Moving all the SQL into the data service, of course, ensures that everyone, however they access PhEDEx data, is certain to get the same results.

Finally, incremental replace-and-enhance has proven to be a very effective strategy for the developers. Manpower for PhEDEx is limited, and replacement of a working website with a better one has often taken a back seat to more pressing matters. The incremental nature of the replacement makes it easier to do this, there are fewer major transitions which take a lot of focussed time and effort from the developers.

6. Conclusions

The re-implementation of the existing website is still ongoing. Considerable time and effort were invested up-front in establishing an architecture that is flexible and performant, and this has proven to be very useful. The same architecture works well for the pure next-gen prototype and for the re-implementation of the existing site.

Our original idea of providing new and more powerful ways to drill down into the data did not match what the users actually wanted, which surprised us. Despite having close contact between the developers and the users, asking a few key users for guidance was not enough. We needed the results of a full survey to finally be able to focus on what the users wanted most.

The User Experience workshop was invaluable. Designing a user interface requires a balance of art and science which is easy to underestimate. Users who are accustomed to a given look and feel of a website may be very reluctant to change to something new, no matter what technical improvements might be on offer.

Eliminating functionally duplicate SQL from the existing website has, in turn, led to richer APIs available to those who monitor PhEDEx via the data service. The goals of consistent representation of information between website and data service are being met as we proceed, as is the goal of replacing the existing website with a more maintainable, flexible architecture.

Incremental replacement and enhancement of the existing website has proven to be an effective strategy. Users get accustomed to the new feel of the interface without having to learn new work patterns, developers get more positive feedback about future directions, and the work can proceed at a pace that matches the available manpower. The delay in introducing new views is compensated by the greater degree of acceptance from the users.

References

[1] Egeland R, Wildish T and Metson S 2008 Data transfer infrastructure for CMS data taking *XII Advanced Computing and Analysis Techniques in Physics Research* (Erice, Italy: Proceedings of Science)

[2] The PhEDEx website, https://cmsweb.cern.ch/phedex/

[3] Egeland, R., Wildish, T. and Huang, C-H. PhEDEx Data Service. *Journal of Physics: Conference Series,* 219(062010), 2010.

[4] YUI, the YAHOO User Interface framework, http://developer.yahoo.com/yui/

[5] Zakas N. Scalable Javascript Application Architecture. Video: http://cern.ch/go/7s7f

[6] Zakas N. Scalable Javascript Application Architecture. Slides: http://cern.ch/go/Cl6S

[7] The Next-gen website architecture http://cern.ch/go/Tl8t

[8] YUI DataTable documentation http://developer.yahoo.com/yui/datatable/

[9] YUI Treeview documentation http://developer.yahoo.com/yui/treeview/

[10] The next-gen user-experience workshop survey results http://cern.ch/go/BmS8