raSANTIAGO+
raSANTIAGO + Associates LLC
107 SE Washington Street, Suite 545
Portland, OR 97214
tel/fax 800 394 0474

A New Architecture for CiviCRM

by Roberto A. Santiago, Principal

06.22.2009 (DRAFT)

Introduction

raSANTIAGO + Associates LLC (raSA) believes its time to migrate the architecture of CiviCRM. With recent developments involving jQuery, testing and the API it seems like the time is ripe for this move. Proposed here is a new architecture that answers many of the outstanding needs of the community while not interfering with the current stream of development.

CiviCRM is a successful project serving the non-profit sector. The increasing adoption by organizations and the growing developer community attest to the importance and success of CiviCRM. But, with the growing community also comes a need to examine the technical foundations of the project.

As larger (and more) organizations grow dependent upon CiviCRM we have the responsibility to see how as a community we can provide stronger guarantees of stability, security and data integrity. Fortunately, CiviCRM like all good open source projects has been able to respond rapidly to the discovery of bugs and security issues. Moreover, support within the community is easy to find.

With the recent rapid transition from 2.2.4 to 2.2.5, though, it seems that a lot can be done at the architectural level to identify and prevent major issues. Such an architectural change would prioritize testing in order to provide stronger guarantees of the released product. In turn, such an architectural focus naturally lends itself to engineering for reuse, extensibility and integrability.

Moreover, the CiviCRM developer community continues to broaden and diversify which brings with it a broader spectrum of functionality and usability requirements to support. We can address these growing needs through a refactoring of existing functionality and the adoption of architectural guidelines which more strictly separate business logic from application logic. and application logic from user interface logic.

Thusly, the goal of such a refactoring and architecture is to create better “building blocks” for developer community.

Where are we right now in terms of the architecture of CiviCRM?

The current architecture was organically grown and reflects the use of older design patterns including Data Access Objects (DAO) and Quickforms. Both of these architectural features are early predecessors to more robust design patterns, specifically, ActiveRecord and RESTful AJAX user

interfaces, two patterns most commonly associated with Ruby on Rails (RoR) but have both existed long before RoR and are just as useful in PHP. The current CiviCRM architecture looks something like the following:

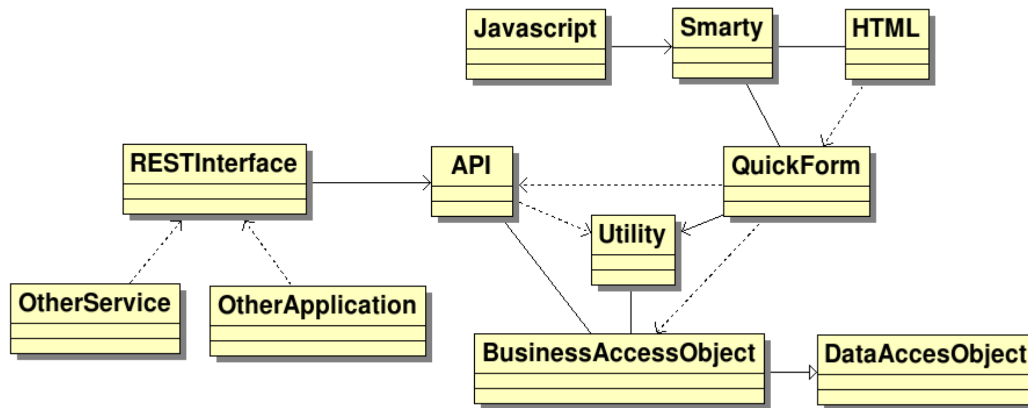

Figure 1. Current CiviCRM Architecture

As can be seen in figure 1 there are a variety of major components which drive the architecture and implement the constituent pieces of the now classic Model View Controller architecture. The most important are Data Access Objects (DAOs) and Quickform. Because DAO is built on a hybrid Data Accessor and Data Mapper pattern it does not handle well many aspects of the underlying Model in CiviCRM. Specifically, implementation of model constraints/filtering, related/associated entity CRUD operations and migrations are lacking. Thus, Business Access Objects (BAOs) have been introduced as a work around to capture and persist specialized code for these needs. BAOs are subclasses of DAOs which are regenerated each time the schema of CiviCRM changes.

Quickforms along with BAOs and their Utility functions carry out the role of the Controller in this architecture. Quickforms is itself a hybrid system for managing business logic and user interface logic. As such it straddles the boundaries of View and Control in the MVC architecture and does even more than that. The core View component within Quickforms is the Smarty templating system which is called out in figure 1 separately. Over time, it appears the boundaries of View and Control have become a bit munged in the current architecture posing issues for expanding the current implementation. This topic will be explored in more detail shortly.

Lastly, there is an API and a RESTful interface which has been added onto the current architecture. The API is interesting in that it seems to reimplement many of the same methods which are currently found in BAOs and within Quickforms. This is likely because implementation was either limited in the BAO or not easily accessed and reused within the Quickform. As a result, there is in some sense two parallel CiviCRM implementations happening right now (the core implementation and the API reimplementation). Added to this API is a RESTful interface so that the API can be accessed by external Services and Applications.

Criticisms of the Current Architecture

For clarity, criticism of the current architecture should not be construed as criticism of anybody who has built,designed or architected CiviCRM. Instead, it reflects the exact opposite: praise for creating an efficient and clever approach to the development of such a powerful and important open source

project. These criticisms are inspired by two powerful forces: 1) the large scale success of CiviCRM and the growth of the CiviCRM developer community and 2) the maturation of design patterns for use in Web 2.0 applications.

The core criticism of the current CiviCRM architecture is that it does not implement the MVC architecture cleanly. The biggest drawback, as a result, is that it is difficult to understand the structure of the controllers and to reuse their business logic. The view implementation is tightly tied to the controllers. In essence, Quickforms is based around the idea of generating user interface pages and forms in a multistep process. Two of those steps (preprocessing and postprocessing steps) capture the meaningful business logic. In this form the business logic is not accessed or reused easily. Recent architectural decisions, specifically Hooks, further complicate this issue. What seems apparent in the code is that there has been some business logic which has been seen as useful and for sake of reuse has been pushed to a BAO or to a Utility class. Unfortunately, this is not uniform and not well documented.

Continuing with the View aspects of the CiviCRM architecture, there is a lot of work being done to provide variables to the Smarty template system, and through the Smarty template system to create nicely interactive user interfaces. These templates are dependent upon Quickforms to populate them. Thus Quickforms in combination with Smarty provide the core implementation of the View. But, as we just discussed, Quickforms also is the core implementation point for business logic. Overloading Quickforms has resulted in business logic being tangled with application logic (sometimes referred to as workflow logic).   The best way to understand the issue is to think about one of the wizards in CiviCRM like the CiviMail wizard. The multistep process is driven through a sequence of forms. Those forms are implemented in Quickforms along with appropriate business logic. But, it is almost trivial to envision a multitude of different wizards which could be developed for setting up a bulk email. But, the current implementation does not separate the important business logic to allow this. Instead the Quickform has to be modified.

The next criticism focuses purely on the Model aspects of MVC. The model currently relies on the DataObjects library found in PEAR. This library is not feature rich and as a result there is need to write adhoc SQL code throughout the application which happens regularly and is unfortunately not limited to the BAO objects. Moreover, many of the utility classes implement the type of functions that are most appropriately implemented on the BAO for manipulating related sets of BAOs. Most of the current needs can be handled with a better Model implemented through an ActiveRecord design pattern. We will discuss this along with other architecture proposals shortly.

Lastly, the future of the API in light of all the other architectural and implementation inefficiencies seems dim. In essence, as discussed before, the API is currently becoming a reimplmentation of CiviCRM which seems not the best way to spend efforts, or, at least if such efforts are going to be spent perhaps we should accomplish more than providing an API, a thrust of the architectural proposal now discussed.

Proposed New Architecture

The proposed new architecture still follows an MVC structure with modifications on the View component. The architecture takes advantage of two trends already in play within the CiviCRM community, the use of jQuery and the development of a rich API accessed through REST interface.

The model component is revamped through the use of Active Record as implemented in the PHP Doctrine library. The view component is now split into application logic with presentation being managed in the browser through JavaScript and the production of XML or JSON in response to calls

to the controller. An authenticating RESTful interface mediates the interaction between these two parts of the view. The Controller now consolidates all business logic and mediates access to the Model component. The new architecture is summarized in Figure 2.

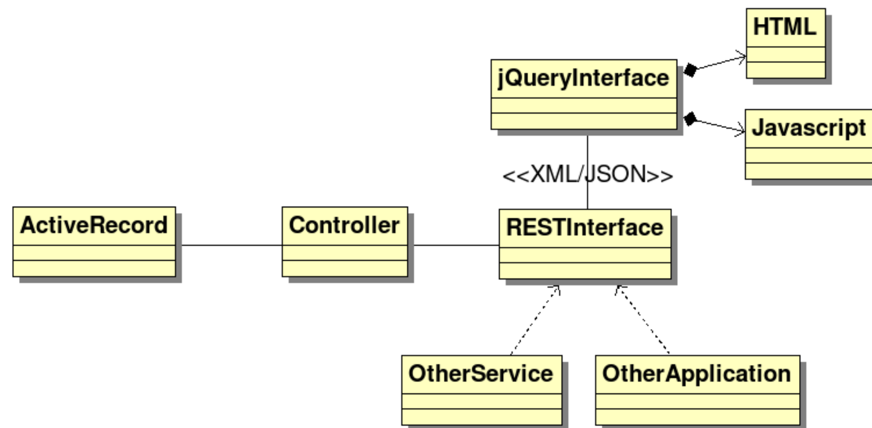

Figure 2. Target CiviCRM Architecture

The new architecture takes a radically different approach to implementing views splitting into Server Side Views and Client Side Views. Server Side Views are the XML and JSON produced by the controller. For clarity, there is no particular limitation to the controller also returning HTML. The proposed architecture does not strictly prohibit this but does not encourage it. The server side views are then sent to the client (the browser). Here in the browser lies the implementation of the Client Side View. this view is made up of HTML and Javascript with heavy reliance on jQuery. Through jQuery we can issue AJAX calls back to the server. Those calls use XML and the server returns XML. The XML becomes the driving data for the user interface logic captured within the Javascript.

In this architecture an email wizard would be implemented in the following way:

1. the screens and widgets would be implemented in javascript and html (e.g. wizard.js within wizard.html)
2. upon load of wizard.html, wizard.js gets loaded and calls to the server side to request initialization.
3. the Authenticating RESTInterface first checks for user authentication and redirects (or throws error) if not
4. The REST Interface then passes the request to a controller (e.g. wizard_controller.php).
5. The controller first verifies the request against ACL (perhaps through a helper object) and then instantiates a new email by calling the Email class (a subclass of Active Record)
6. The controller asks the new Email object to express itself in XML (e.g. new_email.toXML())
7. The controller then returns success indicator along with the new XMLized Email object.
8. The javascript now reveals the widgets for the first step of the wizard (no refresh necessary)
9. The user enters some data and submits to move to the next step. Some simple client side validation occurs (i.e. were all necessary fields filled, were they entered in proper format, etc.). If it passes, the data is XMLized and sent to the server as an update request. If not, errors are reported to the user.

   The wizard controller implements a workflow validation (e.g. checks to see if a previous step has not been completed yet). Since this is the first step there is no problem. If there were workflow errors the controller would return error information. That information would be

presented to the user for correction.

10. Assuming success the Email object is updated and/or other processing is executed with the Email object (with changes persisted in the database). In this update, if any business logic is violated (e.g. we are trying to send to a group we are not authorized to send a bulk email to) an error is reported back to the controller which returns an error signal and the error message from the Email object to the browser. This error is presented to the user for correction.
11. Assuming no errors, success is returned from the server side to the client side and the user interface transitions into the next step.
12. So on and so forth.....

<u>Advantages of the New Architecture</u>

The new architecture has the advantage of pushing most (if not all) business logic to the fundamental objects of CiviCRM (Email, Group, Contact, etc.) which constitute the Model. This in turn allows the business logic to be reused in different workflows and applications. This in turn allows business rules to be implemented in one place ensuring consistency. This in turn, gives us our first major place for implementing testing. (functional, regression, unit, etc.)

Next the controllers offer us similar testing advantages. With the workflow and server side application logic implemented within the controller we can now test that logic separately. (simulating a large number of scenarios as would be presented from the client side). In addition, we can feel free to implement a number of different controller for specialized use without having to simultaneously change user interface code.

The REST interface gives us our next place for testing. In essence we can simulate all the AJAX calls and test for valid processing and response. In addition, security and authentication are centralized on the REST interface again providing consistent use. We can also from the point test the server side implementation of the view. We can run a battery of AJAX calls with a battery of XML documents and verify if we get back a correct web server response and XML document.

The client side aspects of the view can now be implemented separately from the rest of the functionality. This opens up the opportunity to develop widgets which can be reused across modules. It is easy to envision small widgets which can be used in mashups and dashboards as well.

Lastly, as mentioned before, the API for the system comes for free. We do not have to do any extra development or testing. Other applications and services simply access CiviCRM through the API in the same manner as the client side view. A special account for external systems can be configured (like any other account) to limit access to data and functionality. In essence, its just another user of the system like a human user.

<u>Migrating to the New Architecture</u>

What has been described so far can be implemented incrementally without interfering with existing development efforts. Such an approach is possible allowing current modules to exist with new modules implemented on the new architecture. Moreover, current QuickForms implementation can be incrementally converted to the new architecture.

In order to understand the transition from the old architecture to the new architecture the following table serves to cross reference components from the old architecture to the new architecture and provide some further explanation. Notice that often components from the old architecture get broken into multiple components in the new architecture.

Table 1. Mapping from Current Architecture to New Architecture

		DAO	BAO	Utility	Quickform	Smarty	REST Interface	API
MODEL	***Active Record*** pattern implemented through PHP Doctrine Library							
CONTROLLER	Consolidated business logic custom ***Controller*** php files.							
VIEW	**Server Side Views** are XML or JSON produced using PHP HAML called from the ***Controller***.							
	View Mediator: Authenticating ***RESTful Interface*** mediates between server side controller calls with XML/JSON returns and client side rich user interface							
	Client Side Views: Client side rich user interface using ***jQuery Interfaces*** built from AJAX, Javascript, the jQuery library and plugins, HTML and CSS.							

Using Table 1 we see the transition starting with the Model, moving to the Controller and then the Views. Fortunately, the proposed Active Record implementation, PHP Doctrine, has the ability to start with the existing CiviCRM schema and produce a very accurate (but not perfect) set of ActiveRecord objects for the major objects in CiviCRM. Starting there we see the following set of activities to build out the new architecture (note: these activities correspond to upcoming blog posts where we will provide implementation and examples):

1. Using Doctrine assemble a set of ActiveRecords which reflect the objects within CiviCRM. This will act as a new ORM layer. Doctrine and DAO/BAO can exist in parallel. Changes to the schema can be captured as migrations in Doctrine.
2. Building on Doctrine, a simple REST interface can be created which speaks XML and can easily interact with CiviCRM objects through Doctrine.
3. Construct a simple jQuery/JS AJAX component for easily talking to the New REST Interface.
4. Construct simple jQuery widget which can be used in the context of existing user interfaces .
5. Construct first controller. A good candidate would be a merge and deduplication controller. The controller should be subclassed off an extension to the REST Interface enabling the controller to easily talk through the REST Interface.
6. Construct first module built entirely on new architecture
7. Start to refactor existing Quickforms to use Doctrine instead of BAO/DAO/Adhoc SQL
8. Convert QuickForms to Controller for existing module and refactor interface

These are just suggested steps but the first six are already being actively worked on for new projects.

Conclusion

This is a very high level description of a new architecture. The work to implement and flesh out this architecture is already underway. There is a real chance to make some marked improvements into the structure and organization of CiviCRM which will serve well for the long term sustainability of the project. As this is a proposal (and a draft at that) feedback is greatly appreciated. Subsequent blog posts will flesh out more details and provide implementation code and tangible examples.