A Data API for Drupal 7: Key steps to enabling transactional web service support

Written by: Nedjo Rogers and Henrique Recidive
Sponsored by: CivicSpace
Date: October 25, 2007

1. Need and Moment

The powerful and flexible open source Drupal content management system (CMS) rapidly is being adopted for high performance web applications. Drupal has a slim and elegantly coded core, an active worldwide developer community, strong and inspired leadership, and a wealth of contributed plugins (modules) providing extended functionality.

Yet there are key legacy limitations to Drupal's design and core APIs that present barriers to building distributed applications on the platform. Specifically, there is significant unevenness in internal data handling APIs and no comprehensive built-in support for web services.

The leading edge of web development is no longer centrally concerned (if it ever was) with a discrete site and its features, no matter how rich. Rather, it's the structured and nuanced ability selectively to pool information, ideas, images, and contexts seamlessly across many sites regardless of details like platform or implementation. Drawing profile information from social networking sites, pulling place-based attributes from map services, distributing photo and video—Drupal is up to the challenge, but needs some focused renewal to fully realize its potential.

Drupal evolved out of one programmer's student project. Core object types and their accompanying APIs were added one by one in a relatively piecemeal way. Some were treated as objects, others as arrays, others as a combination of the two depending on the context. Primary focus was given to the content (node) object type. A database abstraction library adapted from the PEAR repository provided some limited multiple database support. Later, developers contributed a new and relatively fully fleshed API for managing user input (the Forms API).

The result is a set of core APIs that are uneven between object types (node, user, file, term, etc.), mixed in data handling, and deeply tied to the logic of user input. Solutions designed to act on multiple object types and to conduct direct data transactions not mediated by user actions face major headaches.

In the Drupal 6 development cycle, productive discussions and development helped prepare the way for data API improvements. Developers drafted more consistent data APIs which, while not adopted, provided valuable insight into tasks at hand. Late in the development cycle, Drupal got a new Schema API, providing a key missing piece in enabling introspective data transactions. Outside of Drupal core, a new and robustly designed web services framework emerged.

Many of Drupal's core systems (e.g., code-level database abstraction, minimal use of object properties) reflect the time they were authored in combination with the minimum versions supported for

underlying databases and PHP. Major new horizons are opened by shifting to newer dependency versions, particularly PHP 5.

A guiding idea and main source of strength in Drupal is that "the drop is always moving" ("drop" being a reference to origin of the term "Drupal"). In contrast to many other softwares, there is a firm commitment to continual improvement and renewal, even if it means challenging long-accepted truths and scrapping backward compatibility. There is an emerging consensus in the Drupal developer community that the core data APIs need to be fundamentally reconceived—that only this renewal will enable Drupal to continue to fulfil its tremendous promise.

And the pieces are coming rapidly into place.

This discussion paper aims to sketch in some detail the place we are and the work at hand.

2. Puzzle

There are several components to fully enabling distributed applications on Drupal (five points, below), but the core challenge is in data APIs. So, to start off, some idea of what a Data API needs to accomplish:

1. All data transactions are completely independent of form/user input.
2. Data have a consistent structure in all their forms. E.g., what's returned by a load operation can be sent directly to an update operation without change of format.
3. All objects are of a single data format. Here we need to choose between our two current forms: array and object.
4. Rather than disparate methods for individual object types, we need a single set of CRUD (create, read, update, delete) methods that can be applied uniformly to all object types (node, user, etc.).
5. Create and update operations can accept data without IDs set and respond appropriately. For example, we can save a node with an associated user (author), that user being identified not by ID but by an array of properties. If the user does not exist, she/he will be created as a user. If the user does exist already, the corresponding user id will be saved.
6. Data API implementations, e.g., save a node, include no direct SQL. All needed database operations are handled by the API.

3. Puzzle Pieces

The problem of fully facilitating web services in Drupal divides into five (easy?) pieces, which we'll take as our organizing frame, following them from the old through the new to what's still to do.

1 Schema modelling

Old

- Hard coded CREATE TABLE and ALTER TABLE SQL statements. Schema-related attributes hard-coded into SQL selects, joins, etc.

New

- Schema modelled in PHP arrays by Barry Jaspan.
- Schema creation and updating done on this basis.
- Recent introduction of schema-based drupal_write_record() and drupal_get_fields() methods by Earl Miles, Moshe Weitzman and others, http://drupal.org/node/169982.
- Several addition methods and features available in contrib Schema module.

Still to do:

- Additional attributes to add to Schema API
    - foreign key mapping, implementation discussion at http://groups.drupal.org/node/4328 and in Record module by Henrique Recidive, http://cvs.drupal.org/viewvc.py/drupal/contributions/modules/record/,
    - title field: boolean, should this field be used as a title?
    - rewrite_sql: boolean, should e.g. load operations based on this object type be passed through SQL rewriting?
    - serialize (half-added recently, needs completion): boolean, is this field serialized on saving and unserialized on loading?
    - encrypt: boolean, should this field be passed through encryption (md5) before being saved?
    - validate: where should this field be validated?
    - title: title to use for e.g. form elements based on this field
    - description: description to use for e.g. form elements based on this field
- Additional generic transaction handlers:
    - load: load an existing object, if found
    - delete: delete an existing object, if found, optionally with all its related data
    - validate: validate an object e.g. prior to saving
    - new: create a new empty object with default properties.

2 Database abstraction

Old

- Custom PHP database abstraction functions adapted originally from the PEAR DB package.
- Focus on MySQL support, Postgresql more limited, others sketchy and in contrib.

New

- PHP 5 as minimal requirement for Drupal 7 would open the way to using PHP's native PDO methods.
- Discussion and ideas: Larry Garfield, http://www.garfieldtech.com/blog/drupal-7-database-plans.
- Patch, Add PDO support to Drupal: http://drupal.org/node/134580.
- Record module by Henrique Recidive (first version committed October '07),

http://cvs.drupal.org/viewvc.py/drupal/contributions/modules/record/ includes PDO implementation.

A key benefit of PDO is the ability to load/process fields "on demand". For example, for a node, we can load only the direct properties first. Then, on referencing $node->user, the dataapi will query the users table automaticaly, returning the user object.

Still to do

- Replace custom DB abstraction in Drupal core with PDO.

  Note: This is not strictly necessary to renew Drupal data APIs, but will open new efficiencies and options, like lazy loading of nested (foreign) objects using PHP5 magic methods __get() and __set() in conjunction with PDO 'load into object' features.

3 Data APIs

Old

Custom non-parallel implementations, see Appendix.

New

- Outside of core, CivicSpace produced a "packages" system for Drupal providing a full set of methods for saving and updating all core Drupal (and many CiviCRM) object types: https://customprofiles.civicspaceondemand.org/api/file. Authors, Nedjo Rogers and Henrique Recidive. This "packages" system includes solutions to key Data API problems, among them: creation of parallel methods for all object types; support for external (non-Drupal) APIs; and programmatic export of existing objects (nodes, taxonomy terms, users, etc.) into generic and portable structured objects.
- Several patches and discussions that have helped map out needed steps.
  - Data API: first steps, begun Jan. '07 by Nedjo Rogers, http://drupal.org/node/113435
  - Data API: further steps, May '07' by Nedjo Rogers http://drupal.org/node/145684
  - Creating a library of CRUD API functions for Drupal, begun Aug. '06 by Angela Byron and others, http://drupal.org/node/79684
  - Deletion API, by Chad Phillips, applied but then rolled back, http://drupal.org/node/147723
  - Restructure node object, by Derek Wright and others, http://drupal.org/node/148420
  - Consistent table names to facilitate CRUD operations, by Nedjo Rogers and Angela Byron, http://drupal.org/node/140860

Still to do

1. *Rename tables and fields for consistency.*
   E.g. to be able to drupal_load('term', 21) based on the schema, we need a 'term' table rather than the current 'term_data'.
2. *Define a consistent data type for objects (object or array).*
   Arrays are the default candidate following the current forms API design and historical precedence. However, there are strong reasons to consider objects, since PHP5 has powerful OOP features and 'magic' methods. Justifying a fuller use of object types will require some sound arguments in terms of both performance and code elegance. Henrique Recidive's Record

module is a significant step in this direction (see accompanying writeup on the Record module).

3. *Recast the way we current load properties into objects to reflect their data structure.*
   Currently we have two ways we load relational data into objects: directly, and as a nested array or object. For example, for a book node, the 'parent' property goes directly in the $node object. Some other properties, though, are nested in an array or object (e.g., organic group properties of a node). We need to move consistently to this nested approach.
   To enable schema-aware handling of a data item array, we need to mirror the relationships in the data structure. The most obvious way is an approach analogous to the Forms API. That is, when loading data (or directly creating a data array), we mark keys in a way to identify them as foreign references. A node, then, would have direct properties from its primary table--the node table. All of its other properties would be in arrays associated with their respective tables of origin. E.g., a book with a parent have, instead of a 'parent' key, a '#book' key, which is an array of properties held in the book table, in this case, the parent.
4. *Implement a set of API methods (load, save, delete, validate, new) that can handle all object types.*
   A full data operation should be possible by calling these methods alone. We remove all existing object-type-specific methods, e.g., node_load. However, we still pass the results through dataapi hooks, allowing modules to e.g. add additional properties.
   Aside from major performance and code footprint improvements, introducing a unified set of methods will make it possible to generalize several systems that have so far been limited to nodes. E.g., it will be possible (outside the scope of this present work) to translate the node_access system into a general system for object access, including files, users, etc.

4 User and group level authentication and access control

Old

- The traditional Drupal user authentication system assumes form-based user authentication through a web browser, using cookies to maintain a session. A rudimentary inter-site authentication system as implemented in the core "drupal" module sent user name and password in plain text between Drupal sites.
- Access restrictions implemented through two primary means: (a) user authentication (b) path based restrictions as implemented in the menu system.

New

- As of version 6, Drupal gets an OpenID implementation and drops the legacy drupal module.
- Authentication routines in Services module.

Still to do

- Relatively little, but some further work on exposing authentication to remote applications.

5 Web service support

Old

- Implementations for XMLRPC, RSS, and JSON in core.
    - XMLRPC implemented through parallel system bypassing the regular Drupal menu-based handling.
    - RSS and JSON implemented through the menu system via custom menu items and rendering overrides.

New

- Services module in Drupal contributions repository by Scott Nelson, http://drupal.org/project/services.
- Patch, enable dynamic page loading and rendering into different formats (JSON, XML) by Nedjo Rogers, http://drupal.org/node/145551, outlined relevant approaches.

Still to do

- Bring Services module into core.
- Rewrite existing XMLRPC, RSS, and AJAX implementations to use Services module.
- Implement RESTful APIs.

4. Work plan

1. Enhance Schema API with additional required properties
    - foreign key mapping, implementation discussion at http://groups.drupal.org/node/4328
    - title field: should this field be used as a title?
    - rewrite_sql: should e.g. load operations based on this object type be passed through SQL rewriting?
    - serialize (half-added recently, needs completion): is this field serialized on saving and unserialized on loading?
    - encrypt: should this field be passed through md5 before being saved?
    - validation: provide per field validation.

2. Complete patch: Consistent table names to facilitate CRUD operations, http://drupal.org/node/140860. Meantime, in contrib, implement schema altering to map object types to their corresponding tables, e.g., ‘term’ is the object type linked with the table ‘term_data’.

3. Drop existing DB abstraction library in Drupal core, replacing with native PHP PDO. http://drupal.org/node/134580

4. Write generic data handling functions to complement the existing drupal_write_record:
    - drupal_read_record(): reads a record based on the schema
    - drupal_delete_record() deletes a record based on schema

5. Define loaded object structure as nested arrays, following Forms API model or, preferably, as nested objects with help of new PHP5 OOP features.

6. Write set of methods for object handling: drupal_load, drupal_save, drupal_delete, drupal_validate, drupal_new. (Rename existing drupal_load to drupal_load_file.) Some of this is mapped out in the patch at http://drupal.org/node/113435 and roughly sketched out in the conceptual code at http://cvs.drupal.org/viewvc.py/drupal/contributions/modules/dataapi/ and the existing Schema module in contrib. The new Record module begins to map this out in better detail.

   drupal_new will return an empty object, but with the schema information attached to it. Then we can fill out the fields programatically, e.g.

   ```
   $new_node = drupal_new('node');
   $new_node->title = 'some title';
   drupal_save($new_node);
   ```

7. Rewrite index.php (the file that handles almost all Drupal page requests). Draft code is available in the patch at http://drupal.org/node/145551.

8. Add hook_service and hook_server to core, enabling modules to expose/implement external APIs and implement new webservice services types.

5. Anticipated Outcomes

Renewing Drupal's core data APIs, taking full advantage of new possibilities in PHP 5, and exposing these newly flexible APIs through web services will result in major benefits to the Drupal platform and those developing on it.

- Increased code flexibility based on parallel forms. E.g., a single load method for all object types means we don't need different hook implementations for each.
- New horizons for programmatic creation of much that is now hand-coded. E.g., methods for generating default input forms from schema information.
- Efficiency improvements accruing from increased reliance on inbuilt PHP functionality rather than code-level implementations.
- Seamless integration of web services, support for data transactions in multiple XML, JSON, and other formats.
- Significantly lower barrier to entry for new Drupal developers due to improved consistency in APIs.

6. About the Authors

Nedjo Rogers and Henrique Recidive have collaborated for the past two years as lead developers of the CivicSpace hosting platform, where their work included a focus on data API development. Solving the challenges of producing a consistent set of data handling methods for all Drupal (and many CiviCRM) object types has given them detailed and specific insights into the work needed to improve Drupal core.

Their collaborative projects range from AJAX, e-commerce, and social networking applications to a new AJAX-ready Javascript behaviours registry for Drupal 6, http://drupal.org/node/120360.

Nedjo Rogers, http://drupal.org/user/4481, has been active in the Drupal community for over four years. He has written over thirty modules in diverse areas, contributed documentation and organizational expertise, and authored various core patches, e.g., the theme region system. Data API improvements have been a significant focus for the past year and a half, during which time he has contributed several relevant core patches and contributed to improving the Schema API and related pieces. Nedjo lives in Victoria, BC, Canada.

Henrique Recidive, http://drupal.org/user/12564, has contributed to Drupal over the past three years, beginning with Brazilian Portuguese localization and moving on to module development and core contributions. As a CivicSpace developer Henrique wrote several key pieces of the CivicSpace On Demand hosting platform including integration of external services (payment gateways, domain name registration, etc.), generalized APIs for deploying and upgrading pre-configurated Drupal and CiviCRM sites and theme development modules. Also Henrique developed numerous modules for client sites, including e-commerce, user profiles and AJAX. A current focus is his new Record module, introducing Active Records Pattern to Drupal. Henrique lives in Belo Horizonte, MG, Brazil.

7. Appendix: Existing methods for selected Drupal data types

Object type	create	read	update	delete
node	• node_save(), calls hook_nodeapi() • passed by reference • no return value • accepts object	• node_load(), calls hook_nodeapi() • passed by reference	• node_save, calls hook_nodeapi() • passed by reference • no return value • object accepted	• node_delete, calls hook_user() and hook_nodeapi() • no return value • user id accepted
node type (content type)	• node_type_save() • accepts object	• node_get_types() • returns all types, or a single type either as array of objects or array of strings • accepts node object	• node_type_save() • accepts object	• node_type_delete() • accepts string: node type

Object type	create	read	update	delete
user	• user_save(), calls hook_user() • passed by reference to hook_user() but not to user_save() • returns object • three arguments: user object, array with values and category	• user_load(), calls hook_user() • passed by reference	• user_save(), calls hook_user() • passed by reference to hook_user() but not to user_save() • returns object • three arguments: user object, array with values and category	• user_delete(), calls hook_user • arguments are form values and user id
role	• user_admin_role_submit() • it relies on forms actions	• user_roles() • accepts two arguments: membersonly (boolean) and permission (filters roles which contain a permission)	• user_admin_role_submit() • it relies on forms actions	• user_admin_role_submit() • relies on forms actions
taxonomy vocabulary	• taxonomy_save_vocabulary() • accepts array: form values	• taxonomy_vocabulary_load() • accepts vocabulary id • returns object	• taxonomy_save_vocabulary() • accepts array: form values	• taxonomy_del_vocabulary() • accepts vocabulary id
taxonomy term	• taxonomy_save_term() • accepts form values (array) • calls hook_taxonomy()	• taxonomy_get_term • accepts term id • returns object	• taxonomy_save_term() • accepts form values (array) • calls hook_taxonomy()	• taxonomy_del_term() • accepts term id • calls hook_taxonomy()
comment	• comment_save(), calls hook_comment() • accepts array	• comment_render() • accepts node object and comment id • returns themed comments	• comment_save(), calls hook_comment() • accepts array	• comment_delete() • accepts comment id

Object type	create	read	update	delete
custom block (box)	• block_box_save() • accepts array	• block_box_get() • accepts block id	• block_box_save() • accepts array	• block_box_delete_submit() • relies on forms
variable	• variable_set()	• variable_get()	• variable_set()	• variable_del()