Data API Design Sprint Final Report

1.Executive Summary

Six Drupal developers met at the offices of Palantir.net in Chicago from 4 February through 6 February to discuss the future of Drupal's data handling, commonly referred to as the somewhat mythical "Data API". In attendance were:

Yves Chedemois
Larry Garfield
Barry Jaspan
Karoly Negyesi
Nedjo Rogers
Karen Stevenson

Over the course of three days, the team reached a number of conclusions:

1. Drupal's strategic advantage in the market (its "secret sauce") is an architecture that allows contributed modules to easily value-add content in the system. That design allows for very rapid development of new functionality, both of general use and of use specific to a single installation.

2. The future of the web is interoperability and web services. Drupal's current architecture is ill-equipped at present to take advantage of that, as it is largely bound to a single, entirely-local data workflow (nodes) that lives essentially in a vacuum. For Drupal to continue to lead the CMS/community plumbing market, it needs to refocus its data handling to seamlessly integrate both local and remote data and be able to mix and match the two.

3. The best way to accomplish that goal is to integrate an evolved, more robust CCK-like system into Drupal core. A Field is the basic unit of rich content (i.e., "e-mail address" and "comment", not "string" and "int"), be it local or remote, and Fields are organized into Nodes or Entities.

4. If at all possible, all content should be controlled through a Field API. The "giant blob of data" that is the current node object is not an acceptable way forward.

5. In order for such a system to be successful, it must be API-centric rather than UI-centric. That is, all Data API elements must be fully manipulatable via API calls without any involvement of the Form API or user input of any kind. Any user interface to manipulating Nodes, Fields, or Entities must be built on a solid developer API, not vice-versa. Consequently, data structure optimizations that bind the API to convenient Form API structures should be avoided. The Data API itself must be fully self-contained and data-centric.

No decision was reached on the exact form that the API should take. Two models were proposed, arbitrarily dubbed "Model 1" and "Model 2". They are described in more detail later in this document. Both shared the common principles that:

- All Fields are multi-value.
- A given Field has a data source.
- Different entity types have a different key-space, which for non-local data may be non-numeric.

As a first step, the following actions have or will be taken:

1. CCK 2 for Drupal 6 will drop its variable schema and use the multi-value schema style for all fields.

2. An attempt will be made to integrate fields into the node_load() process directly for Drupal 7.

3. Additional implementation plans are pending.

The attendees would like to thank Palantir.net for hosting the design sprint, Acquia, NowPublic, and Publicis Modem for providing financial assistance, and all of the above for value developer time.

2.Drupal today

Drupal currently has a number of different, parallel mechanisms for data management. The three primary mechanisms are nodes, users, and comments. Each has a separate, somewhat parallel workflow for the object life cycle. That life cycle consists of loading, viewing, displaying an edit form, validating an edit form, saving an edit form, and deleting the object. Different object types have additional steps in their life cycle, depending on their API.

Of these, nodes are the most flexible. There are in practice three ways of integrating content into a node:

1. Implement the node yourself in a module, and provide the corresponding hook_load(), hook_form(), hook_update(), hook_delete(), and hook_view() implementations. This method provides the most flexibility but the highest barrier to entry. It is also tightly coupled with the Form API.

2. "Value-add" to a node via hook_nodeapi()/hook_form_alter(). This method allows modules to add data onto one or more node types arbitrarily in any form in any structure. It has a slightly lower barrier to entry than method 1, but still requires manual handling of code, form, and database and is also tightly coupled to the Form API.

3. Use CCK, which allows wizard-style assembly of node types. This method has a far lower barrier to entry and offers rich data types, but development of new field types is complex, it is not possible to cleanly automate via code (in Drupal 5), and does not cleanly allow for more complex node form logic such as dependent fields.

Of these, only methods hook_load() and hook_nodeapi() offer support for remote data.

Between the three of them, the Drupal community has developed a vast array of contributed modules that can, in one form or another, value-add amazingly complex features in a pluggable form. That capability has been one of Drupal's greatest strengths, and each version of Drupal allows for more and more varied pluggable functionality, creating a positive feedback loop that keeps Drupal strong.

However, all three node modifying mechanisms suffer from some common flaws:

1. The node save routines are tightly-coupled to the Form API.

2. The node data structure is not controlled. Modules may add data to the node object in any way they desire, which leads to namespace collisions, unpredictable data structures, and implementation of certain functionality scattered around various parts of the system.

3. Even if the functionality is similar to that offered on other common object types, the structure and implementation is different enough that every object type (nodes, users, comments, etc.) must be coded for separately.

This situation creates the following conundrum for Drupal implementors: If you want rich data types, you must use CCK and nodes. If you want flexibility, you must use custom code. If you want remote data, you must make your own implementation in a vacuum or else manually wrap data into a node. If you want to work with something that is not node-ish, you are typically on your own.

While Drupal offers a great deal of flexibility nonetheless, it is not always easy to perform certain tasks. Those tasks will become more and more common in the near future. We can do better.

3.Web services

There is a growing consensus that The Next Big Thing on the Internet is the long-discussed promise of Web Services. That is, the ability of different web servers and web applications, both server-side and client-side, to expose data in common, predictable forms to each other, and allow users to mix-and-match that data with additional functionality, will be the driving force of the Web marketplace. Such data sharing takes many forms, including SOAP, JSON, RSS, RDF, REST services, and other acronyms.

Drupal currently offers such mix-and-match functionality within its own node sandbox, and does so well (with the caveats mentioned in the preceding section). However, there is no standardized way to handle external data sources, nor is there a standardized mechanism for exposing Drupal's data to other services. The closest such standards are Views-based RSS feeds or the Services module. Both, however, deal with exposing Drupal data in non-HTML form, not with consuming remote data.

In general, there are four non-exclusive ways in which a web site or web application can function:

1. Island site: No external data sources, no exposed data feeds.

2. Web services server: Exposing Drupal-based data to the outside world in some standard, predictable, readily-digestable format.

3. Web services client: Consuming data from a non-Drupal remote source and exposing it to Drupal's value-add systems.

4. Legacy front-end: Exposing some legacy data store (non-Drupal SQL database, flat file system, old mainframe, etc.) to Drupal's value-add systems.

Methods 3 and 4 are similar in that they both involve consuming data from a non-controlled source and exposing it to the rest of the system using the same interface as locally-defined data, and for the purposes of this discussion can be treated as a single use-case. And, of course, the holy grail is being able to do all three simultaneously, with the same data.

4.The Data API

The team established early on the following working definitions:

- *Property*: An individual piece of meta data about a Entity. Same across multiple variants vs. the field which varies against subtypes..
- *Entity*: A collection of multiple Data and related Properties that can exist in multiple variants.
- *User*: An Entity representing a user with access to the system.
- *Node*: An Entity representing an abstract piece of content.
- *File*: An Entity representing a binary object.

A *Field*, analogous to how it currently exists in CCK, was also established as the desired basic unit of content. An Entity then is a collection of multiple Fields. Non-Field data on an Entity must be deprecated in favor of a common Field API to allow a consistent data structure and access mechanism.

The *Data API*, then, is a unified storage and retrieval interface for Fields and Entities. As a strict interface, the implementation details of storage can vary, over time and per-field, without impacting code that uses that interface. Similarly, data destinations (HTML, RSS, forms, XForms, JSON) can be added and refactored without impacting the storage mechanism. While that may make certain structural optimizations (e.g., a Form API structure that happens to map directly to a database record) impossible, that is an acceptable trade-off for the looser coupling such a system provides.

A basic set of storage systems, that is, a basic set of Fields, can live in core while additional and more esoteric Fields can live in contrib. Fields may be internally complex, but must expose a predictable API. For ease of implementation and greater predictability, all Fields should be treated as multi-value, even if they happen to have only a single entry.

5.The Institute of Contemporary Art

Throughout much of the discussion, the team considered the use case of a museum web site exposing its catalog of artwork through Drupal. This use case was not entirely fictional, as one team member (Garfield) was at the time in the process of launching a second such site. The description below, used as an example in later discussion, is based on that site.

The ICA has their entire archive in a legacy database system. That system has a read-only SOAP interface, using RPC-style calls that return an array structure. There are several types of "thingies" that the system exposes to us: Artworks, Resources, Artists, and Categories. Each has its own integer key-space, in addition to between 2 and 30 single-value fields, ranging from an artwork title to the URL of the large-sized jpg of an artwork. Resources also have a type, which is one of "quicktime", "full HTML", "audio clip", "jpg", etc. Each Resource type needs somewhat different theming treatment. Categories have only a few fields themselves, such as title, but are used much like Taxonomy is in Drupal currently.

6.Data Models

The team discussed two general models for the construction of a coherent, cohesive Data API, arbitrarily dubbed Model 1 and Model 2. Although the terminology and implementation details differ, both approach the problem in the same way:

1. There is a non-node concept (Thingie, Entity, or Content, depending on the version) to which Fields can be attached.

2. Each Entity has its own primary key-space. That key space may or may not be numeric.

3. An Entity instance (e.g., a specific node, which is an instance of the Entity type "Node") may not have any existence in Drupal's local database.

4. Fields may be added to any Entity type, which is how contrib modules "do their thing".

5. Searching is specific to a Entity type, and is implemented by each Entity type in the most reasonable manner for that Entity type. (e.g., searching nodes involves searching the local database and caching the results. Searching Artworks involves a SOAP call to an existing system. Etc.)

In Model 1, additionally:

1. If an Entity instance is value-added by a contrib module, the key of the Entity instance and the entity's type together form the key to be used by Fields for saving additional data.

2. Users are not an Entity type, as they are not "content". They may, however, tightly bind to an entity.

In Model 2, additionally:

1. If an Entity instance is value-added by a contrib module, a local key is generated. The local key is always numeric. The entity type and local key together form the key to be used by Fields saving additional data. In the degenerate case of Nodes, the primary key-space and the local key space are identical.

2. Users are an Entity type.

3. All Entity types (User, Node, Artwork, Resource) may have sub-types. For Nodes, these are nodes types as we know them. For Entity types that do not really need sub-types (Artwork), there is only one.

The full writeup of each Model is available online:

- Model 1:http://groups.drupal.org/node/8794
- Model 2: http://groups.drupal.org/node/8796

In neither Model was the question of revisions or translation addressed.

7.First step implementation

Although a final architecture has not been established, the team did discuss "first steps" toward implementing such a system. Many involve changes to CCK for Drupal 6 to make it easier to eventually move into Drupal core.

7.1.Core vs. Contrib

The following components of CCK are "core critical", in that without them the system isn't sufficiently functional to be useful.

- field storage engine
- Field CRUD
- widgets / form gen.
- Node CRUD
- formatters
- add more / drag 'n drop re-ordering
- Field validation
- custom multiple value handling

The following components of CCK will remain in contrib, as they are not critical for the system being functionally useful.

- field default value
- field allowed values
- Field CRUD UI (Manage Fields tab)
- Display UI (Display Fields tab)
- Fieldgroups
- Content Copy
- 3rd party Drupal integration (Views, Pathauto, Token)

7.2.Definitions

- A field type is a set of functionality implemented by a module, e.g. text, nodereference, gmap.
- A field is a collection of settings of a field type: e.g. a text field, max length 60, plain text (no input format option for the user).
- A field instance is the binding of a field to a content type with additional settings: e.g. the above text field assigned to

the story node type with the label "Five word summary."

7.3.Changes

First, one of the most complicated parts of CCK currently is the logic to handle the variable schema. At present, CCK fields that are used on only one node type and are single-value only are stored in a single table for the node type. However, as soon as the field is made multi-value or used on a second node type its data is moved to a dedicated field table. Although in theory that design would allow for more optimized queries, in practice it doesn't actually get used that way. The Drupal 6 version of CCK will drop the variable schema and use a dedicated table for each field, including a delta in case the field is multi-value, which should greatly simplify both CCK code and other custom code.

Second, some field properties will be made immutable. Once a field is created, they may never be changed again (without some add-on module that does various complex things to the database schema directly, but that will not be part of CCK core). Immutable properties are, in general, those that affect the storage schema of the field, such as its field name or, for text fields, whether it takes an input filter or not. A full list will be determined by the CCK maintainers.

The D5 and earlier versions of CCK uses a separate formatter for each individual field value, one after the other. That does not always make sense. There should be a possibility to format all values for a field in a single formatter (like a field that contains multiple points on a map or a slideshow of images). This change has gone into the D6 version of the code.

The D5 and earlier versions of CCK presume widgets will handle multiple value forms themselves and pass them back to the Content module. In D6 that is changed so that widgets create a single form value element and it is the Content module that combines them together into a single form element. Some widgets need to opt out of that behavior to handle all values in a single form element (like a select list or checkboxes). Now that the Content module handles multiple values, there is also a way for the widget to opt out of that and handle its own multiple values. The optionwidgets module does this.

7.4.API

There are two ways that modules could add fields to a node/entity type: Declaratively, similar to hook_menu() in Drupal 6, or procedurally, via field_create_field() and similar API calls. As the former will require the presence of the latter anyway, it was determined that the direct API will be implemented first and a declarative mechanism considered later, if it turns out to be feasible.

The API will require, at least, the following operations:

- Create node type / Delete node type
- Create field / Delete field
- Add field to node type / Delete field from node type
- Update field configuration (required, multiple, widgets/formatters, weight)
- Get table name for field
- Get types
- Get fields
- Get field instances for type

Core-based fields will follow a similar node object structure to current CCK fields. That is:

$node->fieldname[n]['column']

It will be the responsibility of module-defined fields to avoid namespace collision. User-defined field namespace collision will need to be handled by the CCK UI (contrib).

7.5.Field types for core

In order for a field type to be included in core, it must have both wide applicability and/or be a requirement for some existing core field type. The team identified the following core concepts that could/should be ported to "fields":

- body: text field
- created: date
- upload: file
- user pic: image
- IDs: number+options

The minimal useful set of field types for core, then, is: text, number, optionwidgets, image, and file. Other current CCK-core field modules can remain in contrib for the time being until they are needed for some core field. While a date field in core would be useful, it is better done as part of a larger overhaul of core's date handling, which is out of scope for this document.

7.6.Next steps and questions

There are several open questions that still remain, and feedback from the community is requested.

1. Nodes as classes. Currently, nodes are represented as a PHP stdClass object. In PHP 4, that has no functional difference from arrays except for a different syntax. In PHP 5, it is essentially "an array that passes by reference". The team agreed the current design is silly. The team also agreed that all first-class entities should use the same data structure, whether an object or an array. Although no definitive decision was reached, some possible functionality, such as lazy-loading, will in practice require the use of classic objects (that is, objects of a specified class, not stdClass).

2. Conversion of Drupal core to all-fields. Having implemented a small number of new-style fields in core, we will need gradually to convert much (all?) of the remaining existing 'fields' to the CCK-style system. Doing so will require new methods and functionality. For example, existing CCK cannot fully handle the current node taxonomy implementation.

3. It has been suggested in follow-up discussion that "fields in core" without the ability to handle remote data is a wrong direction to take, and any "fields in core" implementation must include a design for remote-data handling. No decision has bee reached on this question as of this writing.

8.Design Sprint Lessons Learned

As this was the first such "design sprint" for the Drupal community (as far as we are aware), the attendees offer the following "lessons learned" for those planning future such events:

1. Face-to-face meetings such as this one are very valuable, and greatly improve the development process.

2. A size of about six people is good for such meetings.

3. Three days is a good length for such meetings.

4. A (informal?) facilitator, who is at least somewhat detached from the subject at hand but can help keep the team moving, is very valuable.

5. A note taker / secretary is also critical, who may or may not be the facilitator. The note taker needs to understand

the subject being discussed.

6. Keep the scope reasonable. Our initial scope was too large, which resulted in lots of unnecessary preparation for subjects we never actually addressed.

7. It is helpful when attendees are all at a similar skill/expertise level, but it is also useful to have different backgrounds (CCK experts and not, OOP fans and not, etc.) represented.

8. Sometimes key contributors may need financial assistance to attend. Despite the cost it is a worthwhile investment. The commercial community (Drupal-based companies) understand that value, but employee time is still precious.

9. An 8 hour day is about right. Despite the temptation to go longer, it is no more a good idea for brainstorming and high-level design than it is for programming or any other task. Trying to work part of the day and then move on to the sprint is a poor use of time both for the sprint and for the day job.

10. Nightly wiki summaries are a useful way for both participants and others to analyze and review the results of their work.

11. Take a break and have fun! Dinner out or similar "organized down-time" is important to keep from being overwhelmed.

12. A good facility is essential. A good facility needs
    - A whiteboard
    - WiFi
    - Sufficient space
    - To not be in Chicago during a February snow storm. We recommend Hawaii.

Put everyone up at the same hotel. It avoids wasting time on travel and provides more chances for unscheduled interaction. Lots of good ideas came up while eating at Panera.

9.Useful Links

- Model 1:http://groups.drupal.org/node/8794
- Model 2: http://groups.drupal.org/node/8796

- Day 1 notes: http://groups.drupal.org/node/8660
- Day 2 notes: http://groups.drupal.org/node/8690
- Day 3 notes: http://groups.drupal.org/node/8714