ModularityMechanism

`needsupdate`

Modularity Mechanism

The http://www.dspace.org/conference/presentations/architecture.ppt DSpace+2.0 proposal mentions some key changes that I think need to occur for a modular system. These are informed by various other efforts and design patterns, such as Separation of Concerns and Inversion of Control (TODO add references)

Modules are responsible for their own data.

In order for a module to be independently 'pluggable', i.e. dropped in, removed or modified without the need to modify other modules, a module needs to be responsible for its own data. This basically means not sharing database tables or data files. For example, take the e-person table in DSpace 1.x – this is directly used by several modules, and if you modify it, you need to alter several 1.x modules/packages (e-person, workflow, web UI).
However, we don't want a situation where module X requires PostgreSQL, but module Y requires MySQL or Oracle – I think we should strive towards some uniformity there. This means allowing modules to connect to a common database (although using separate tables), and ideally allow different databases to be the backend, which probably requires disciplined use of standard SQL.
I also wonder if we really need the DMS 'wrapper' that we have in 1.x (org.dspace.storage.rdbms) – it may be responsible for some of the performance issues we're seeing with DSpace with large amounts of data. It doesn't allow you to use PreparedStatements or do much optimisation in terms of D query. Although suggesting having components use JDC directly seems to contradict the previous paragraph, actually the wrapper doesn't help because modules calling org.dspace.storage.rdbms still have to use raw SQL queries for the most part. (Problems with porting to other databases with 1.x seem to mainly relate to the org.dspace.browse code rather than the org.dspace.storage.rdbms code.) Perhaps using JDC directly (with some centrally defined parameters /methods for connecting to the database etc.), and having some requirement/test that ensures the SQL a module uses is standard might work better.

Using a good persistence layer such as hibernate (or JDO) would mean we didn't have to worry about getting dirty with SQL/JDC.
I'd also recommend that we (or the persistence layer) use the DataSource interface, and leave managing connection pools, caching PreparedStatements etc. to one of the several high quality OS implementations around. JimDowning

---

Modules interact only via APIs

As a side-effect/corollary to the above, the proposal is that modules interact only via defined APIs. This API is essentially a contract between the two modules, and also allows either module to change (or be replaced) independently of the other.
A module may require an implementation of API X to be present in order to function. It shouldn't need to know what module is implementing that API or how.

This decoupling is vital, but in my experience really very difficult to achieve. One approach is to add a method for every query a client wants to do on the module. This means that every query can be optimised in one location, and that all schema specific code is localised. It also means that it's very slow if you're developing the consumer and implementation at the same time (this isn't a problem if the system is **completely** specified up front, but that's rare!), and it increases the expense of developing an implementation of the module interface. Another approach would be to use a query language as the api, which means that each client can optimise their own calls. The downside is that to all intents and purposes it locks all implementations of the module interface to the same persistence mechanism (and probably schema). A tricky one.
Anyone with any solutions? JimDowning

Modules may present a UI via the UI framework

Having the Web UI as a separate module in 1.x, using the APIs of business logic modules, doesn't seem to work from a modular perspective. Any functionality change usually has an impact in both the business logic layer and the Web UI layer, and hence requires a change to the API in between them. Hence, the proposal is that modules can present their own UI via a UiFramework, the UiFramework being the 'glue' that holds them together.
Finding out just how attainable a goal this probably requires some experimentation.
As an example scenario, consider a custom workflow module for theses. This involves changing the submission workflow code, the data stored by that code, and the corresponding Web UI changes. These three aspects don't make sense independently – the functionality involves all three aspects. The Web UI aspect is useless without the updated submission workflow code (or API), and that in turn requires the relevant data stored.
In other words, to enable truly pluggable modules, I think the boundaries between modules need to be changed from how they are in 1.x – i.e. a module is responsible for data, code and the UI (at least, the Model and Controller parts of MVC – the View part should be customisable to some extent via the UiFramework).

Choosing a mechanism

The only really fundamental requirement for us are, I think:

- APIs can be defined and kept stable
- DSpace instances are free to 'plug in' different implementations of each API, such that other modules in their DSpace invoking that API will actually invoke the chosen implementation.
  There are lots of other potential 'nice' features of course, such as dependency specification (module X needs an implementation of API Y to be present), and 'hot' swapping in and out of modules. However, I'm a fan of keeping things simple – I'd hate to see things get complex without a compelling reason.

I am strongly of the opinion that for 2.0 we should think just in terms of in-process Java APIs. In the 2.0 proposal I've presented ways of scaling DSpace up without the need for networked APIs (Web Services etc.) That may come later, but for 2.0 lets keep things tractable in a reasonable time frame and stick with Java APIs.
Possibilities here include leveraging an existing tool, or 'piggybacking' Cocoon's modular mechanism, based on Avalon, I believe, if we do go with Cocoon.
/obertTansley
(Note - just in case this comes into question later - contents of this and referenced pages are not under GPL license)