WikiCourse

It is proposed that we look into a basic build-time mechanism to incorporate add-ons into DSpace. This would provide us with valuable experience and a step along the road to installing individual modules onto the DSpace platform. The purpose of this is to allow add-on writers to be able to produce a distribution package that will be installable onto a DSpace installation using a standard build procedure. This should make it easier for new features to be added to any instance of DSpace and act as a step along the way to a fully pluggable system. Exactly what would count as an add-on might include: Large chunks of community developed functionality which is not appropriate/mature enough for the core or locally developed custom functionality that may or may not become part of the core later.

At this stage we would like to gather developers' experiences of installing add-ons created locally to understand all the issues involved. Please either create a sub-page (HelpOnEditing/SubPages) of this page with your add-on experiences and installation techniques, or provide a link to somewhere on the web where this information can be found.

- CheckerAddOnTechnique
- [wiki:Self.TapirAddOn The Tapir E\-Theses Add\-On]
- /XmlUiAddOnIdeas

There is an AddOn Prototype in production: Visit /AddOnPrototype for more information

Update 20/01/2006: Initial parts of second prototype available here: attachment:addon-prototype2.tar.gz

Update 27/01/2006: v2.2 of the addon prototype now available here: attachment:addon-prototype_2-2.tar.gz (please ask if you would like access to the SVN space where this is currently housed)

Update 03/02/2006: v2.3 of the addon prototype now available here: attachment:addon-prototype-2.3.tar.gz

Update 24/02/2006: v2.4 of the addon prototype now available here: attachment:addon-prototype-2.4.tar.gz

Below is the start of a proposal for an add-on mechanism. It is not complete, and some points may be controversial, so please comment either on this page or on dspace-devel.

Code Management

Before attacking the technical details, here is an overview of the types of code we're dealing with, and an idea of the direction in which we'd like to head.

Right now, there are two basic types of code in the DSpace community: Code that is 'owned' by DSpace – i.e., it's in Source``Forge CVS, it's part of the package that people download, and responsibility for managing it is down to the DSpace committers; and code that is managed and distributed by other people and organisations. Currently, the only way for the latter type of code to get distributed and have the maintenance burden shared is to submit the code to Source``Forge as a patch and have it put in core DSpace.

A more appealing situation is shown in the diagram below.

attachment:types-of-code.gif

The general intent is initially to define a mechanism whereby external DSpace modifications (C) can be made easier to maintain and distribute independently of the core DSpace code (A). This mechanism can then be used within the DSpace-managed code itself to:

- Make it easier for DSpace instances to configure in and out functionality relevant to them.
- Allow the add-ons to be managed in a delegated fashion, i.e. not all serialised through the core committers. This would enable us to have a more hierarchical committer/developer structure, like Apache, Linux kernel etc.

Both of these will become critical as the code grows and the uses of DSpace continue to diversify. However, note that **a usable 'out of the box' configuration remains critical**.

The idea is to restrict code flowing through arrow 1 to bug fixes and real *core* code changes – things to do with the underlying storage, the data model and so forth.

Most extra functionality would, instead, initially be made into a well-behaved, externally-managed add-on (arrow 2). Then it will be much easier to move such code to a place where the maintenance and development burden is distributed (arrow 3). Once centrally managed by DSpace, the add-on can be included with the core DSpace download or downloaded separately as seems most appropriate.

Once this mechanism is established, we can also start moving some existing DSpace functionality into this add-on structure (arrow 4), enabling delegated maintenance. A good example of a candidate for this is the OAI-PMH data provider code. This also means the core will move towards a real set of core repository services.

Well-behaved add-ons

In order to keep add-ons maintainable, both by developers and by those using the add-ons, certain constraints and practices must be followed. An add-on that does this is called for the purposes of this write-up a 'well-behaved add-on'. For an externally-maintained add-on to become a DSpace-maintained add-on (arrow 3 above) it **must** follow these guidelines.

Must have completely separate code

In almost all cases, bolting code into the core DSpace API classes can and should be avoided. If some change to a core class is absolutely required, an enhancement (e.g. creating a plug-in interface) can be added to the core separately from the add-on. This should be a well-thought enhancement to the core, not just a hack.

Must have separate, 'namespaced' database tables.

This may mean that absolute normalisation is not achieved, but joins can be made trivial and this is a necessary trade-off to make add-ons manageable. If multiple add-ons e.g. add a field to the 'Item' table, numerous problems may result.

By 'namespaced' we mean that all tables for an addon have a prefix to prevent clashes with the core DSpace tables and others. e.g.

{{

Unknown macro: {x_researcher_folder}}

}

For the 'researcher pages' add-on.

We considered having a separate database for all add-ons, or even a separate database for individual add-ons, but that would involve further configuration, maintaining a separate pool of database connections etc.

Are versioned separately from DSpace

Including their schemas. This also means that upgrade tools for add-ons will be separate from the core DSpace upgrade tools.

Versioning add-ons separately means that they can be updated independently of the core DSpace code base.

Each add-on should have a published compatibility matrix – i.e., for each version of an add-on, we should know which versions of core DSpace (and any other required add-ons) that add-on version is compatible with.

This of course means that add-ons will need their own `schema.sql` files and `update_10-11.sql` schema updating files.

On database access

Of course, add-ons need to be able to read + write their own tables. For efficiency's sake, e.g. to do joins etc., they may need to access core tables. This is discouraged; if it's possible, the core DSpace APIs should be exclusive used. This will make the add-on easier to maintain.

In any case, add-ons should not *write* to the core tables; the DSpace APIs must always be used with a valid `Context` object. Otherwise, authorisation may be circumvented, critical operations might not happen, e.g. search/browse index updates etc.

The standard DSpace storage API should be used rather than JDBC directly. This should make managing database compatibility much simpler.

The following describes the allowable access mechanisms.

attachment:db-access.gif

	Direct (using JDBC etc.) (A)	**DSpace storage API (B)**	**DSpace content/core APIs(C)**
DSpace Core tables	**NO**	**READ ONLY; DISCOURAGED**	**YES**
Add-on's tables	**NO**	**YES**	**N/A**

All access to the bitstream store should be via the `org.dspace.content.Bitstream` class.

Configuration

Should we have a strategy whereby all config files of a certain 'type' are concatenated as part of the build process?

e.g. dspace.cfg gets concatenated with dspace-tapir.cfg and all the other dspace-[addon].cfg? Concatenating is easy enough.

Also of concern is 'default' versus 'live' configuration. e.g. if I install add-on X, change a config param, and then update add-on X, I need to be sure that my changed config param hasn't been overwritten.

Web UI

Some XSLT solution for modifying/building web.xml to alter servlet classes / mappings?

libs

Extra jars can be copied in. Do we need a way to prevent duplication? Proposal below should prevent this need – any jars in the core DSpace distribution can be used by any add-on.

JSPs

Force add ons to the use the 'local' customization route? Proposal below removes this need – add-on's JSPs are added into the UI .war directly by DSpace's own `build.xml`.

Package names

DSpace-maintaned packages under `org.dspace`, others as originators think appropriate? Issue: backwards-compatibility when an externally-maintained add-on becomes a DSpace-maintained add-on?

Build process

Currently, the process existing add-ons use generally looks like this:

- The add-on has a `build.xml` file or script which needs to be pointed at a DSpace source tree
- The build file compiles the add-on and modifies the DSpace source tree – inserting classes, possibly patching some files, patching web.xml etc.
- The DSpace source tree can then be used to install a DSpace instance with the add-on activated.

Possible problems with this approach include introducing errors into the DSpace build and clashes/conflicts when adding more than one add-on. It is also likely to make code management with CVS much more difficult. Another approach is proposed which may work better:

- Add-ons have a standard directory structure, for example:

For illustration – this is probably not complete and needs a few revs. It deliberately tries to match the core DSpace tree.
{{

Unknown macro: {add-on-0.2/{panel}

build.xml - add-on's build file. to be controlled by DSpace build.xml (not vice-versa)
config/
addon.cfg - default config for addon, to be added to dspace.cfg
emails/ - email templates the add-on uses
language-packs/Messages.properties - Messages used in UI by add-on
etc/
database_schema.sql - add-on's extra tables
database_schema-1_0-1_1.sql - update between versions (add-on's tables ONLY)
jsp/ - JSPs
lib/ - Any extra required JARs
src/ - Source files

}}}

- Configuration parameters or a configuration file in the main DSpace source tree is used to point to/activate add-ons. For example, a file `addons.properties` could contain:

  {{

  Unknown macro: {/home/johndoe/add-on-0.2 = true/home/johndoe/supercool-1.0.1 = true}}

  }

  Alternatively, this could be done with parameters, perhaps something like:

{{

Unknown macro: {ant -Daddons=/home/johndoe/add-on-0.2,/home/johndoe/supercool-1.0.1 fresh_install}}

}

Or there could be a special directory within the DSpace source tree which is automatically scanned for add-ons.

- The DSpace build.xml file reads the add-on configuration (in whatever form), and because the directory structure of the add-on is known, can incorporate the add-on into the DSpace build without having to actually having to modify the main DSpace source tree in situ.

Updating

In general there are three flavours of code update:

- **Update the DSpace core**

Before updating the DSpace core, you'll need to check whether the add-ons you have are compatible. (Hence the compatibility matrix requirement above). If they aren't, you'll have to either wait for the add-on to be updated, or temporarily disable the add-on.

- **Update an add-on**

Updating an add-on that is compatible with the same version of DSpace should not be a problem – just update the DSpace add-on property to point at the updated add-on's source tree and do an ant -D... update as before. To update the add-on's database schema, either the build process will need to be able to work this out automatically (e.g. by looking for database_schema-XX-YY.sql files) or a manual step will be required.

- **Update both at once**

Updating both at once will work the same as "update an add-on" above, except that it's the updated DSpace source tree that will be pointed at the new add-on. Depending on the changes to DSpace's core database and the add-on's, the add-on's schema update may need to be done before or after the DSpace core database schema update.