DSpace 2.0 Design Proposal

Robert Tansley, Hewlett-Packard Laboratories

Version 0.2 / 28-October-2004

1 Introduction

The DSpace open source digital asset management system has received unanticipated widespread interest and adoption. Two of the system's main strengths which have enabled this are:

- The system provides the full breadth of functionality required to set up and operate a digital asset management system 'out of the box'
- The open source approach allows organisations to tailor the system to local requirements, as no two organisations have exactly identical needs

However, there is still work to be done. The system was developed 'breadth first', with finite time and resources. In addition, the system is being put to a variety of novel uses not previously envisaged. The diversity of the community interested in DSpace is remarkable. Naturally, this is not a case where 'one size fits all'.

This document outlines some general areas where the DSpace architecture needs to be improved. Then a proposal for an architecture addressing these needs is described.

This document assumes familiarity with the DSpace 1.x architecture, information model and implementation. Also, basic familiarity with the Open Archival Information System reference model is assumed.

One more important point: This is a proposal for DSpace version 2.0 – the choice of the number '2.0' is important. There will (if DSpace continues to be successful!) versions 2.1, 2.2, 3.0, and so forth. This proposal is focussed on developing the architecture to a point where more step by step, evolutionary development is possible. It is not presented as an attempt at a 100% complete or perfect design; nor does it attempt to answer every requirement of a digital asset management system. Rather, the proposal is focussed on creating the DSpace 2.0 platform as one that will enable future development.

2 Areas for Improvement

2.1 Modularity

Given the number and diversity of individuals and organisations interested in using DSpace, one size will certainly not fit all. In addition, the area of digital preservation (as well as other aspects of digital asset management) is very much in the research arena. One of the drivers behind the open source approach is to promote and facilitate experimentation; some of these approaches may turn out not to be great, and never production quality.

Although DSpace 1.x is divided into a number of components, modifying or extending functionality could still be simplified further in the long term. Once modifications have been made to the system, it can be difficult to update your modified system in line with the main DSpace source distribution. For instance, if you've made a large number of modifications to version 1.1.1, to update your code base to DSpace 1.2 can involve quite a complex merge operation. There is no easy way to 'unplug' the default implementation of a component and replace it with your customised version. Additionally, the user interface is currently one single component. Customised or extended functionality normally materialises in the user interface. It is difficult to 'plug in' extra functionality into the user interface.

The proposed approach here is to make the system more modular. In common with most modular architectures, the principal aims here are:

- Allow extra functionality to be developed and 'plugged in' optionally, so that people who don't need that functionality aren't affected
- Allow modifications of one piece of the system to be made without worrying about the rest of the system
- Allowing an existing component of the system to be replaced without having to modify other parts of the system
- Allow easier integration with other existing systems an organisation may have invested in

The main changes that will have to occur to achieve these aims are:

- More disciplined definition and maintenance of interfaces between modules
- No direct sharing of data between modules (for example, database tables). This is important to enable modules to be transparently plugged in and out. For example, I may want to be able to replace the default e-person module with an LDAP-backed implementation. If other modules are using the e-person database tables from the default module directly, this will be problematic. This approach does come with some extra development cost, but this is the price one pays for true modularity.
- Breaking the UI up into separate, independently replaceable modules

2.2 Internationalisation

The original intention with using Java Servlets and JavaServer Pages (JSPs) was to enable, as far as possible, separation of processing code and display, following the Model/View/Controller (MVC) model. However, this hasn't worked out as well as we'd hoped. The JSPs necessarily contain rather more Java code that we'd like. This means that in order to make localisations or translate the UI, you have to copy and modify the JSPs, including the Java code within them, which effectively forks the code, complicating maintenance. If the core DSpace UI changes at all, then the Java code in the JSPs often changes.

The localisation (adapting the UI to a particular organisation) and internationalisation (translating the UI) processes are both the same; this means if you've 'localised' the system at all, your translation of the UI isn't really sharable (people will get your localisations too).

Also, even if you've translated the DSpace UI, it's still mono-lingual – there's no capacity for DSpace to speak one language to user X and another to user Y.

There is definitely a requirement for improvement in this area.

2.3 Preservation

In the DSpace 1.x design, all metadata is in a relational database, and all content bitstreams are in the file system on the server. This is not an ideal situation from a preservation point of view.

- You need to be careful about backups. If for some reason the database becomes corrupt, all you have is a jumble of bitstreams.
- In any sort of backup restoration, disaster recovery, or situation where an organisation is taking over the contents of a DSpace from another organisation, one must have the hardware and software stack available that runs a compatible version of the relational database software so that the metadata can be restored. In order to be truly long-term, data needs to be independent of the software and hardware stack in which it is stored.
- Another essential digital preservation process is basic auditing; i.e. periodically ensuring that the content in the archive is all present and correct, to ensure that storage systems are not failing, and content has not become corrupt. In DSpace 1.x, this is relatively simple in the case of bitstreams (sizes and checksums are stored for each), but the all-important data in the relational database is not easily auditable in this way.
- Additionally, the existing DSpace 1.x approach forces you to go through hoops to actually share content, an essential component of many digital preservation strategies involving keeping multiple copies of content in a variety of geographic locations. To share an object, you need to know how to retrieve the appropriate bits and pieces from the database and file system. However, the content management API does some of this work for you, so this is not as critical a concern here as the previous points.

In order to improve a DSpace instance's resilience, it is proposed that storage of DSpace content is reworked such that both the metadata and content associated with an archival object are stored together, with the metadata in a standardised format that can be read without the need for a particular hardware or software stack. This means the Archival Information Package (AIP) is a more tangible concept in the system, as opposed to being a purely logical concept ("an AIP consists of these files here, together with this row from this database table and these rows from that database table" etc.)

This approach has advantages, from both digital preservation and scalability viewpoints:

- No reliance on a particular set of hardware or software to be able to read the AIP metadata
- Don't need to understand how DSpace system works to be able to transfer or rescue data
- Hence easier to reconstruct an archive in the event of disaster recovery, or receiving stewardship of assets from another organisation

- These ready-packaged AIPs are easy to move around, making mirroring and distributed storage strategies easier to implement

In this approach, these AIPs represent the real ‘archive’. For performance or other reasons, various indices or copies of the metadata may be stored in a database or other mechanism, but these are considered a cache of the information in the asset store.

3 Proposed Architecture Overview

The new modularity and reworked storage are reflected in the four-layer architecture shown in Figure 1 below.

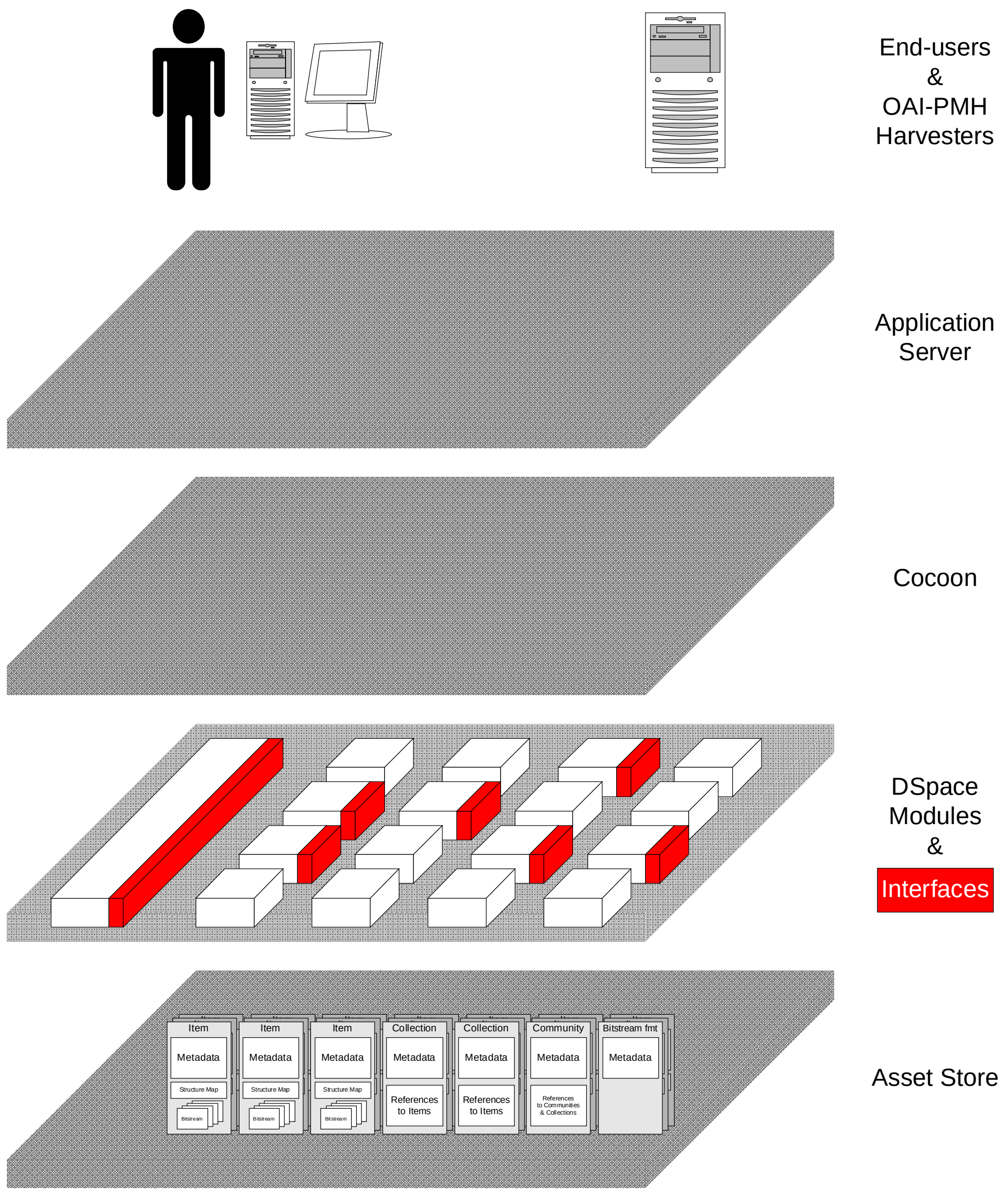

Figure 1: DSpace 2.0 Four-Layer Architecture

Each layer is summarised below, starting bottom-up from asset store:

- The **Asset Store** layer is the reworked storage of DSpace content. It consists of self-contained AIPs, each containing the metadata and bitstreams for a DSpace Item, Collection or Community. Even if the three other layers all 'blow up', through hardware or software failure, becoming obsolete or some natural disaster, it will still be possible to reconstruct the archive of content and metadata, provided you understand the format in which the AIPs are stored. Naturally, the implementation of the asset store can vary from DSpace instance to DSpace instance.
- The **DSpace Modules** layer is where modules can be 'plugged in or out'. These modules communicate via interfaces and do not share data directly (e.g. relational database tables). Modules can interact with the asset store via a defined asset store interface. Modules may also make use of indices and caches of content held in the system maintained by other modules, e.g. a Lucene-based indexer module; it is not the case that every module needs to maintain its own indices.
- **Cocoon** provides a means for modules to present their UIs in such a way that the end-user feels like they're interacting with a single application as opposed to a set of individual applications. It will provide the 'glue' necessary to do this, tying together separate modules for navigation tools, the look and feel (site 'skin'), and so forth. It should also address issues like internationalisation, enabling the look and feel and language used in the UI to be changed independently of the modules in the DSpace Module layer.
- The **Application Server** is the interface between a DSpace instance and the outside world. Any standard Java Servlet/JSP server such as Apache Tomcat or Jetty should fulfil this role.

You may have noticed that a relational database is not present in the above diagram. However, many DSpace modules may use a relational database; however, they should not share tables, and only communicate via defined APIs. Additionally, any indexed or information duplicated from the asset store is essentially a cache; the asset store holds the authoritative contents of the archive.

The following sections describe each layer.

4 The Asset Store

This layer holds the authoritative contents of the DSpace instance. These are divided into separate Archival Information Packages (AIPs). Each AIP is a self-contained package containing the metadata and bitstreams of a particular object.

Note that the asset store is not a direct parallel to the storage layer in DSpace 1.x. In the DSpace Module layer described in the next section, there will be a relational database which modules can use to store such things as e-person records, authorisation permissions, indices, and workspaces for creating and modifying content. Hence the asset store represents the archival storage component of DSpace 2.0, as opposed to the entire data storage function.

Another important point is that the asset store need not necessarily be the only means for modules to access information. For example, there may be an index module, which maintains indices or caches of information in the asset store. This index module could provide a query interface to other modules in the DSpace instance. Many of those modules might not need to access the asset store directly at all.

There may be many different implementations of the asset store available. A simple file system based implementation could be the default. Other implementations based on Grid-style or other storage systems will be possible. The asset store is itself a DSpace module with a defined interface, and as such the implementation can be changed.

4.1 DSpace Archival Information Packages

This section needs more work – especially regarding representation information.

The asset store contains only AIPs. These AIPs may have different types however, and the underlying information model is not hugely different from DSpace 1.x. There are still Communities, Collections and Items. Communities may form a hierarchy, and contain collections. Collections contain Items. Items contain Bitstreams, though these Bitstreams are not AIPs in their own right, they are part of an AIP.

The model is evolved slightly from the 1.x model in terms of representation information. In DSpace 1.x, Items contained Bundles of Bitstreams, and each Bitstream is associated with one Bitstream Format. This is not enough information for many digital preservation purposes, for example:

- Structural information is often required. For example, to make use of a sequence of page images one needs information about their ordering
- Bitstreams may encapsulate other Bitstreams with different formats, e.g.:
    - Compressed 'zip' file. To make use of the contents, the formats of the files within the zip file must also be known
    - 'AVI' video file. As well as knowing that the Bitstream is of the '.avi' format, one also needs to know which encoding algorithms are used in the audio and video streams. It is not optimal to have a separate Bitstream Format entry for every possible combination of audio and video algorithm.
- In the case of many more complex objects, representation information over and above the format of the Bitstream may be required to make use of the object. For example, if the object is source code, information about the build environment and required libraries and so forth is required. For executable objects, or ROM images and so forth, a description of the hardware/software environment in which the object will operate is required.
- Format registries (and environment registries) are likely to become increasingly important for digital preservation concerns.

The above factors point to the need for a more generalised approach to representation information.

An additional requirement here is a more flexible approach to metadata in general – not just representation information, but descriptive, administrative and other kinds of metadata.

This document proposed an object-oriented style inheritance approach:

- A 'base class' of DSpace AIP has some basic information common to all AIPs. All DSpace modules can expect this information to be there.
- Each 'subclass' of DSpace AIP (Community, Collection, Item etc) may have additional mandatory information. E.g. every Item may be required to have a Dublin Core record
- Beyond that, the model is extensible. Items may have other kinds of metadata.
- Applications may subclass further for specific application needs.

This approach is shown in Figure 2. Note that the objects do not have methods; this is important.

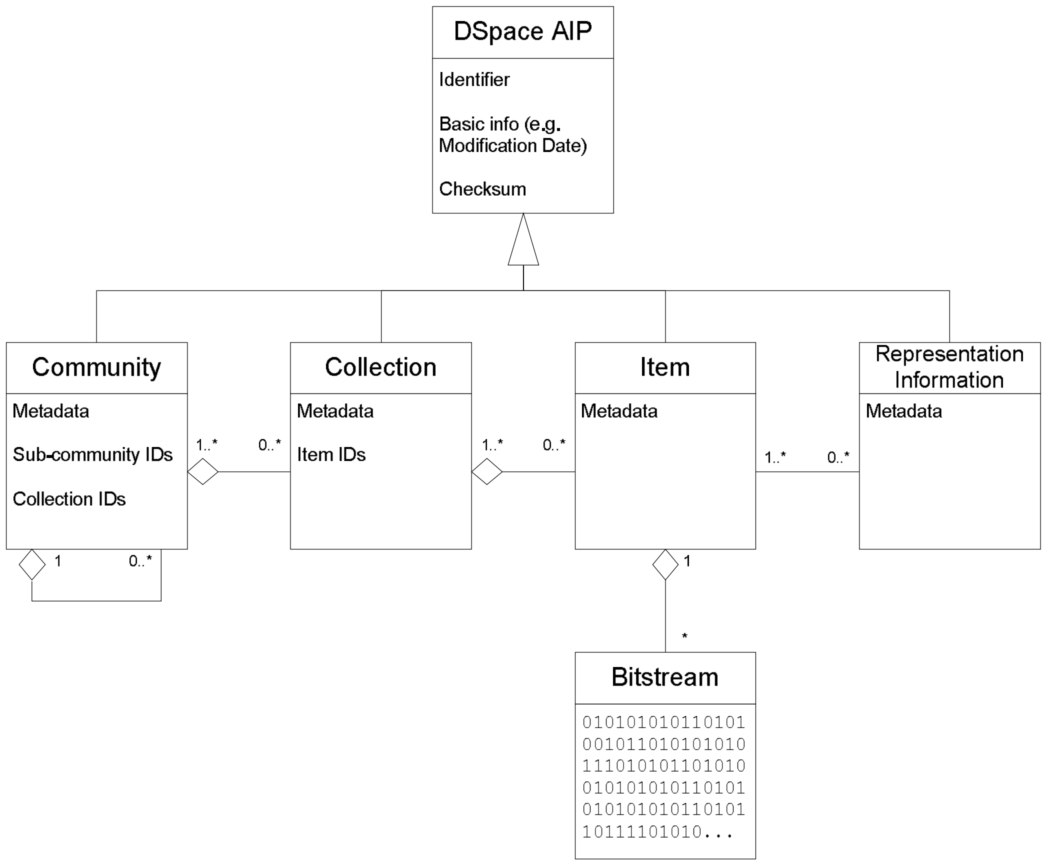

Figure 2: Archival Information Package Model

4.2 Asset Store Interface

An important notion here is that the asset store is not required to understand the contents of the AIPs other than the basic information of the DSpace AIP class. Or rather, an implementation of the standard asset store interface does not need to understand the contents of AIPs. It is the modules in the layer above that understand the contents of the AIPs. This separation of storage and semantic concerns is important to keep DSpace modular.

Hence, the asset store interface needs to be fairly general. Once again, it is important to note that DSpace modules that have a deeper understanding of the contents of AIPs may index and/or cache the metadata in AIPs, and provide more convenient interfaces to other modules in the system.

This document proposes that the asset store interface itself presents an AIP in two parts: An XML serialisation of the metadata AIP, and the bitstreams. There should be methods for:

- Creating a new AIP
- Getting, retrieving and updating the XML metadata serialisation
- Storing and retrieving Bitstreams in an AIP

Identifiers for AIPs should be treated as opaque text strings; identifiers are another area where one size does not fit all, and so another, independently replaceable module should be responsible for allocating identifiers. However, the asset store will need to be able to retrieve things by ID efficiently.

4.3 Versioning

TODO: Simple approach

4.4 Push/Pull

One topic of debate has been how those modules that maintain indices or caches of information in the asset store keep those indices and caches up to date. There are two principal approaches, each with pros and cons:

Push: In this approach, the asset store is responsible for 'pushing' the fact that something has changed up to those modules maintaining indices and caches. There are a variety of ways of achieving this. In general, the asset store sends out a message, event signal or notification whenever something has changed, indicating what has changed. Those modules 'listening' for these can then obtain and update themselves accordingly.

Pull: In this approach, those modules who maintain indices and caches are responsible for keeping their indices and caches up-to-date; the asset store is a passive agent in this model. In general, modules would periodically pull (poll) the asset store to find out what has changed since the last poll, then obtain the changed metadata and/or content and update their indices and caches accordingly.

There is nothing in the DSpace 2.0 architecture that is intrinsic to either approach. The main issue is whether the core asset store interface is geared to one approach or the other.

A neutral approach might be to have an agnostic core asset store interface; then both approaches could be used. There could be a defined 'pull-model' interface which asset stores can implement; in addition, some sort of notification, event/listener or messaging module could be implemented. This way, both approaches can be tested to see which works best in which situations. However, this may mean that indexing modules would have to either support both approaches, or only be usable in certain DSpace instances where only one of the models is supported. Actually, it should be pretty easy to have an index module support both models; in terms of how the actual indexer works, the difference is probably superficial, it's just a matter of timing.

5 DSpace Modules

The DSpace module layer consists of DSpace modules, which communicate via defined interfaces. This follows the 'component-oriented programming' paradigm. Essentially, a number of interfaces are defined. A module may:

- implement zero, one or more of these interfaces
- invoke other interfaces
- declare the interfaces they need to invoke to function (dependencies)

A module may not:

- Invoke classes/methods in other modules that are not part of the defined interface. Otherwise, one would not be able to transparently replace a module implementing the defined interface.
- Write to database tables (or other data storage) under control of another module. That represents an assumption about how the other module functions. Again, one would not be able to replace that module transparently, as it may use different database tables or another form or storage.

The above restrictions may result in some extra development effort, but this is the necessary price to pay for modularity.

5.1 Modular User Interface

Most changes of functionality to the system are reflected in the UI in some shape or form; perhaps with some new pages, or some existing pages are changed or replaced. In DSpace 1.x, the Web user interface is a single component in the system, which makes it hard to make and maintain such changes over time. In general one does not want to replace the entire Web user interface component. Hence there is a need to modularise the user interface itself.

Cocoon is proposed as a tool that provides the 'glue' to allow a modular user interface to be executed; see section 6.

A potential path forward is to separate those modules that present interfaces to other modules with those that are UI components. Separating of the user interface and business logic concerns is generally good design practice.

However, there are a variety of situations where enforcing that a module can be either a user interface module, or a business logic module but not both, may be restrictive.

- A user interface module might have an admin API that other modules can invoke.
- Consider the case where a module that implements a commonly-used and defined API (interface X). This implementation of that API may well have an

associated admin UI, for example to manage and configure connection to an LDAP server. One could enforce the design pattern depicted in Figure 3. However, the esoteric API (admin interface) that the module developer must create could be very specific to that implementation. In such a case, rather than declare the API to the DSpace system and have three things to worry about distributing and configuring (module, admin UI module and API in between) it may be preferable to follow the design pattern depicted in Figure 4. Naturally, the design pattern in Figure 3 is often preferable and should be encouraged, and nothing about the design prevents that pattern; however, *excluding* the possibility of the design pattern shown in Figure 4 does not seem to convey any advantage.

- As well as being a production system, DSpace is also intended to be a platform for experimentation. Allowing a single module to present a UI as well as perform business logic lowers the barrier to prototyping tools.
- Integrating third-party tools which cannot themselves be divided into separate modules.

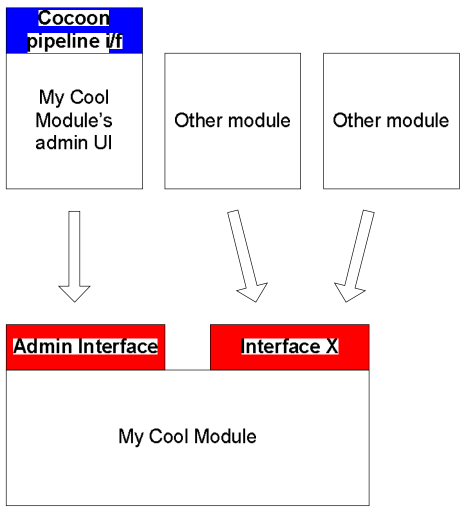

Figure 3: Separate Admin UI Module

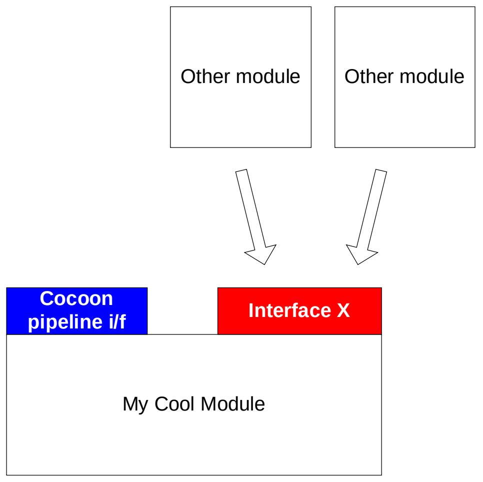

Figure 4: Module With Inbuilt Admin UI

Hence, it is proposed that modules may be a user interface module, and provide an API, but not both. From a broad DSpace architecture perspective, this makes all modules peers. However, other design patterns (such as separation of UI and other concerns as far as is possible and appropriate) may be overlaid and implemented within the DSpace 2.0 framework. If one particular design pattern proves to work extremely well for all of the uses of DSpace, then DSpace 2.1 or 3.0 can evolve to take that into account.

5.1.1 Navigation

Each component of the user interface will need to be bound together, however, in order to make a DSpace instance, composed of a set of modules, feel like a single application, as opposed to a group of individual applications. Maintaining a common look and feel should be relatively straightforward, as there are many mature technologies for achieving this, such as cascading style sheets and XSLT transforming style sheets. A greater challenge is providing navigation between different module's UIs, and allowing integration where necessary.

For example, the site navigation bar could allow modules to 'plug in' links. More complex examples include authentication screens, such as username/password entry screens.

5.1.2 Internationalisation

Additionally the DSpace Web user interface should allow modules to be internationalised'. Internationalisation refers not only to allowing translations of the UI to be made without affecting the underlying DSpace module, but also allowing DSpace instances to be 'multi-lingual', for example to allow a French user to see the UI in French and an English user to see it in English.

Language 'catalogues' or 'dictionaries' are the established way of achieving this. Instead of the module providing the UI with prose, the UI is provided in terms of tags, which the UI framework can then look up the relevant translation for before actually sending to the user.

This becomes a little tricky in a modular environment, since each module will require its own set of translations. Hence a fairly complex matrix of DSpace modules against available translations is likely to emerge. However, an active and motivated open source community should make this a manageable problem.

5.2 DSpace Distributions

The above framework provides good flexibility; it provides a 'canvas', within which modules can be plugged in and out to create a wide variety of flavours of DSpace.

This approach may seem to push DSpace more into the middleware domain. However, we should not lose sight of one of the strengths of DSpace – that it provides a breadth of functionality 'out of the box'. Hence, the default distribution of DSpace should contain a complete, ready-configured set of interfaces and modules providing that functionality with a minimum of additional effort to the organisation deploying the system.

As communities develop around particular uses of DSpace, it is likely that alternative 'distributions' of the DSpace system with particular modules included and pre-configured will emerge to suit the set of functionality common to that community.

6 Cocoon

The Apache Cocoon system provides mechanisms to address the above concerns. Its pipeline-based approach to creating Web pages and sites allows the concerns of 'plug-in' module user interfaces, site look and feel ('skinning'), internationalisation and so forth. Figure 5 shows an extremely simple potential Cocoon site map demonstrating how this could be achieved. This is just a simple illustration; Cocoon offers richer functionality that DSpace can exploit, particularly in the area of multi-page interactions, such as the submission user interface.

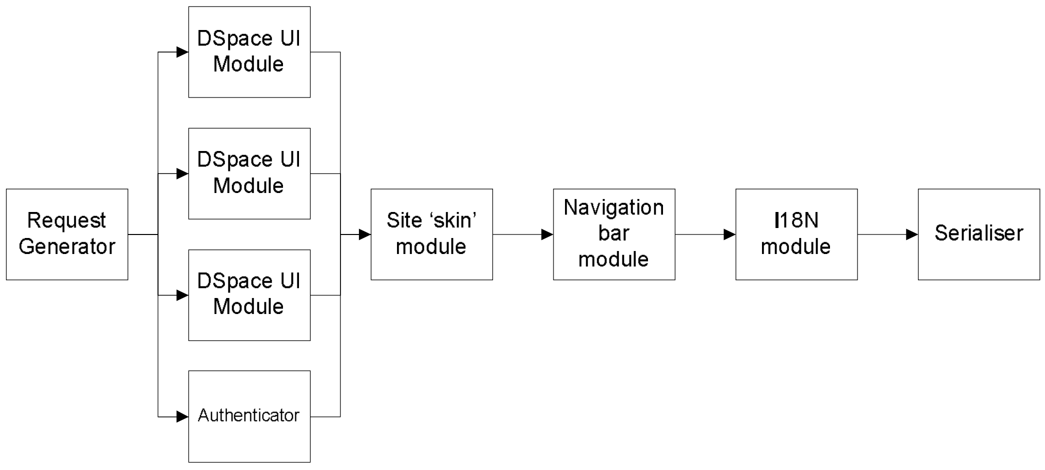

Figure 5: Simple DSpace Cocoon pipeline

7 Application Server

The application server is just a standard servlet container, such as Tomcat, Jetty, Caucho Resin and so forth. This should help enable DSpace to be deployable on a variety of platforms and take advantage of advanced application server features like clustering, load balancing and so forth.

8 Example DSpace Deployments

More detail; step-through of operations

This section describes some example DSpace 2.0 instances. Hopefully these should illustrate how the architecture works as a whole, conveys the required improvements over the 1.x architecture outlined in the introduction.

8.1 Default Distribution

Figure 6 depicts the DSpace 2.0 modules that might be present and configured in a default distribution of DSpace. This would be an ‘out of the box’ installation with roughly the same functionality as DSpace 1.x. Figure 6 depicts only the ‘DSpace modules’ layer of the four-layer architecture; it’s as if we have ‘zoomed in’ to that layer in Figure 1.

The arrows depict some principal interactions between modules, though of course they’re not exhaustive. Hopefully this makes it clear that not all modules will be interacting with the asset store directly, in fact it may be a minority of modules that do.

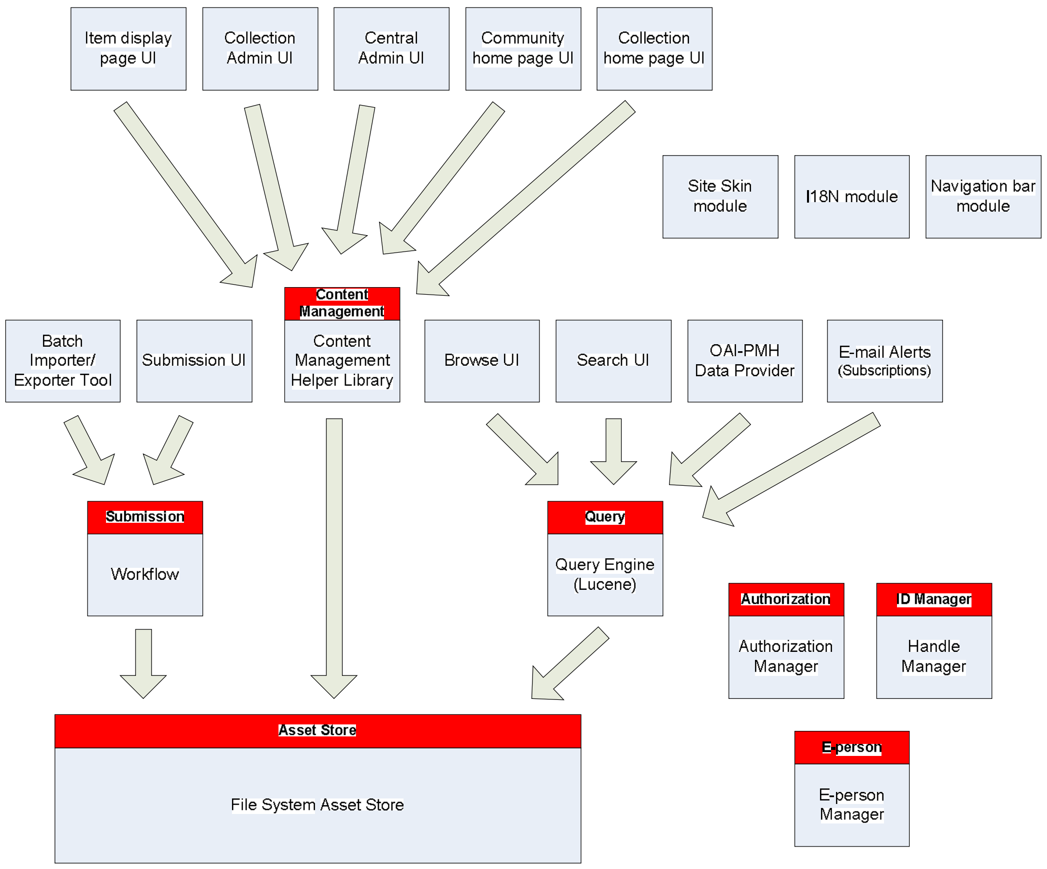

Figure 6: Default DSpace 2.0 Instance

8.2 Shared Asset Store

This example deployment shows how two DSpaces can share an asset store. In this example, the DSpace instances are specialised; one is a DSpace for submitting or batch importing content. The second DSpace instance is a ‘read only’ access deployment.

Figure 7 depicts the specialised deposit-only DSpace instance. Notice how many modules are not present – this DSpace is streamlined to perform ingest well, without the added load of end users querying and retrieving content from the same server.

Figure 8 depicts a streamlined read-only DSpace, which shares the same Grid-based asset store as the instance in Figure 7. This instance can be optimised to handle indexing, large numbers of queries and downloads by end users. This ‘read only’ deployment could be repeated on different servers and in different geographies.

The design of the asset store makes it very easy to share content between DSpace instances, and in this way the DSpace instances form a robust, scalable and distributed system.

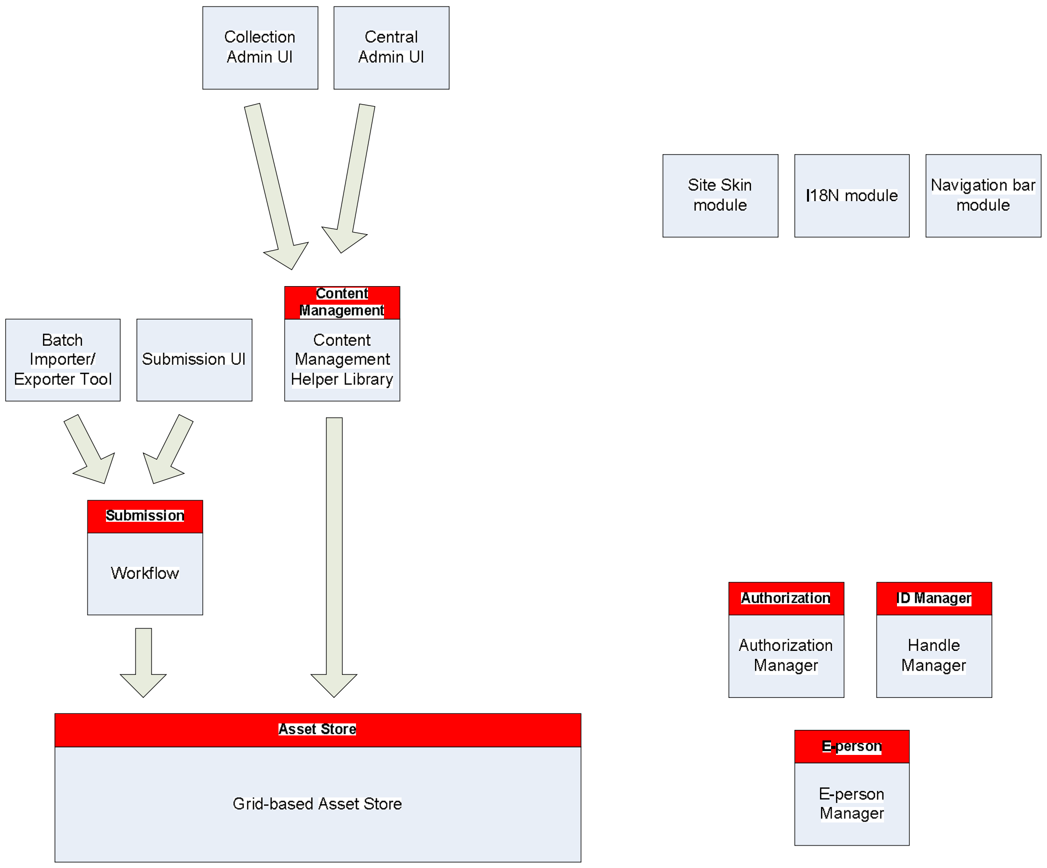

Figure 7: Specialised Submission and Batch Import DSpace Instance

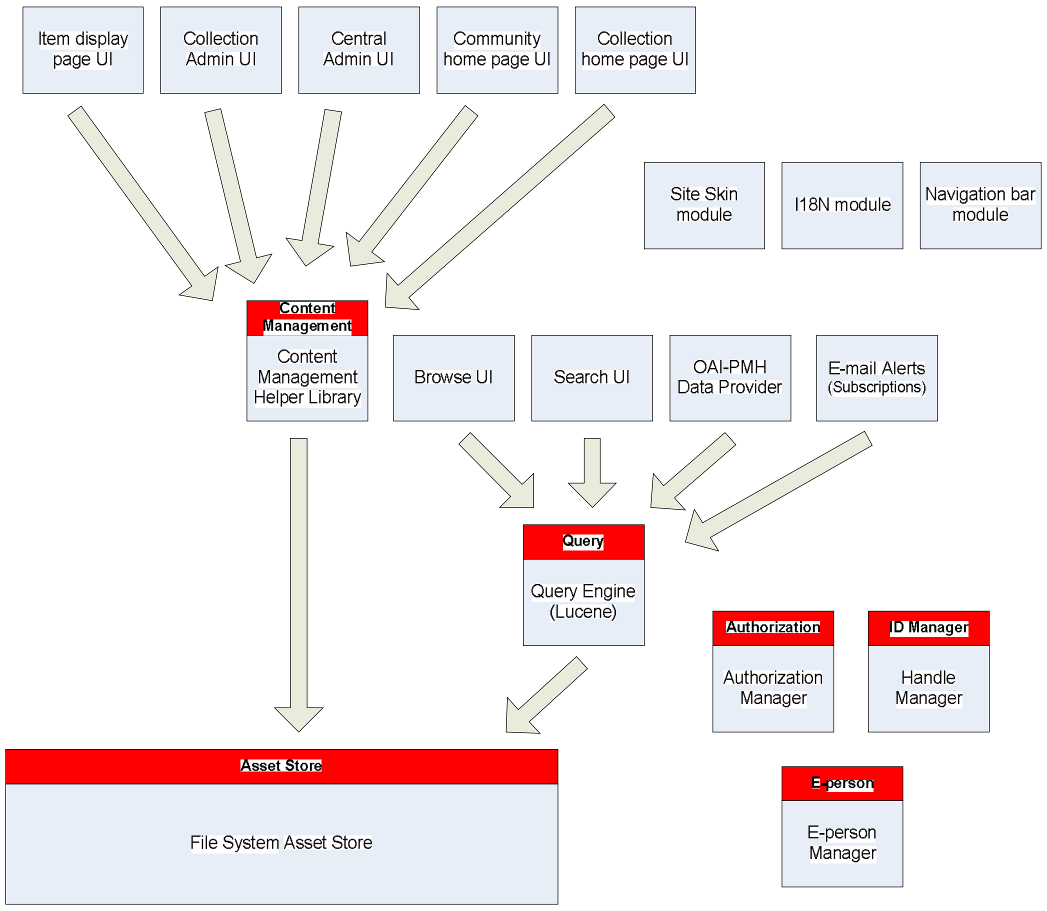

Figure 8: Specialised Read-Only DSpace Instance

9 Security Model

Needs some work J haven't had time to work on this for a while…

Here's a quick and dirty run-down of the proposed, simple security model for 2.0. There are some basic assumptions:

- DSpace modules running within a single instance are trusted. You don't plug in a DSpace module into your DSpace unless you trust it to respect the authorisation API and to authenticate end-users or other external systems appropriately. For systems where security is a big deal, you don't want to be plugging in modules which might be malicious.
- Authentication mechanisms will vary significantly depending on the method of accessing DSpace. E.g. X509 certificates are probably not going to cut it for a Web Services interface, especially since Web UI logins are likely to have a UI component.
- The Authorization API has a performant way of answering the question "can e-person X do action Y (to object Z)"
- Provided that the asset store implementation is appropriately 'blessed', i.e. has authenticated and is authorised with the storage mechanism, the storage

mechanism trusts that the asset store manager is only doing things that it has deemed are authorised in the overall environment. (i.e. if there are organisation-level user authentication/authorisation standards, the DSpace JVM has modules that use/enforce that.)

So the responsibilities go like this:

- Modules that interact with clients (external users or machines) are responsible for securely authenticating those clients (working out which e-person they are)
- The module implementing the authorisation API has ultimate responsibility for deciding what an e-person can or cannot do
- The asset store implementation is responsible for invoking the authorisation API to ensure that the asset store API is used 'legally'.

So I guess I'm putting a stake in the ground saying that I DON'T think the DSpace 2.0 architecture should have to worry about malicious code running in the DSpace instance (e.g. a module invoking the asset store API pretending to be a privileged user to get at restricted information or do something destructive.)

As far as something like Shibboleth goes, I would expect it's possible to implement a DSpace module or modules that implement UI authentication, the e-person API and the authorisation API in a Shibboleth-compliant way. It may also be that what sits behind the e-person API has no local records at all (other than a cache) but all records are on some remote server.

Figure 9 shows how the basic security model works in a step-by-step fashion. The bright green arrows show the how process goes for the end-user of a UI performing an operation. The purple ones show the process for some external process interacting via some Web Services interface.

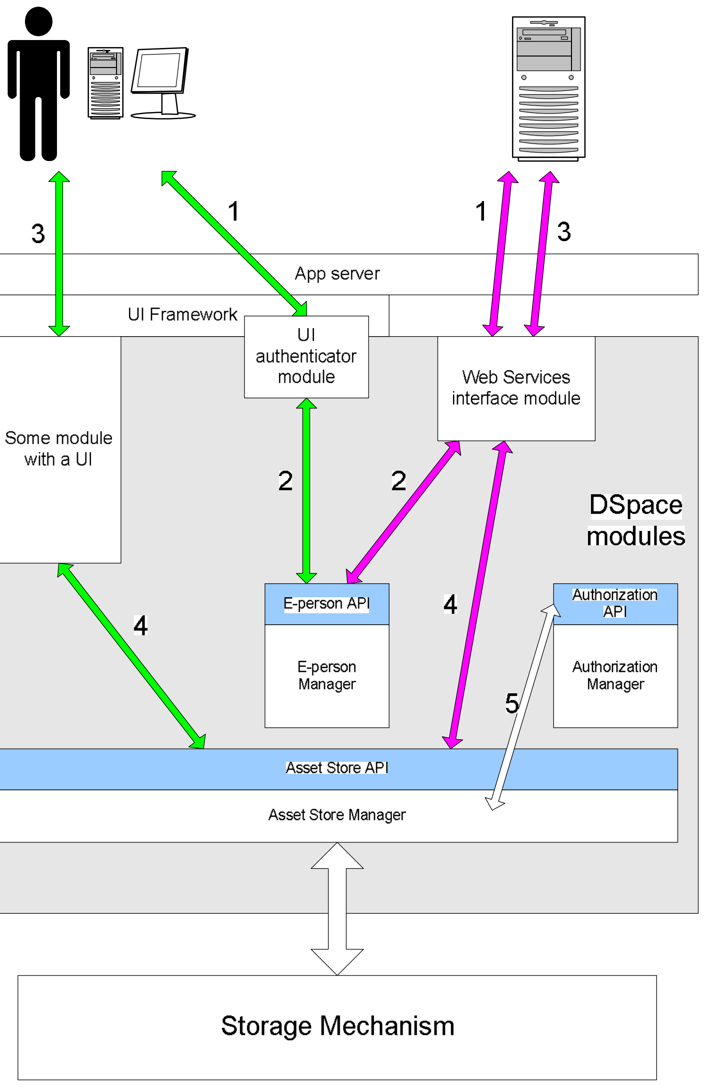

Figure 9: DSpace 2.0 Security Model

FIXME: Is this diagram confusing? May need reworking

For the UI:

1. Before the user can do anything (apart from anonymous read) they must authenticate with the UI framework. This is a UI-specific process.
2. Whatever module is responsible for authenticating users of the UI must decide whether the end user corresponds to one of the e-people in the system. (In certain circumstances, e.g. it has a verified X509 certificate but there is no e-person record, it may create a corresponding e-person record.) If this is correct, the end user is given some credential. Currently, in DSpace 1.x, what happens is that the session cookie the end-user has is 'blessed', i.e. we say 'operations performed using this session cookie are being done by X' and this is trusted throughout the DSpace instance. This is how most Web applications work currently, as far as I am aware.
3. The user can then interact with the UI they are using to perform the operation.

4. In order to perform the operation for the user, the DSpace module invokes the asset store API, indicating that it is user X attempting this. The asset store API trusts the module to be telling the truth about who is attempting this.
5. The asset store manager then invokes the authorisation API to ensure that user X is allowed to do what they are attempting. If all is OK, it performs the operation, otherwise it throws an exception.

The Web Services interface follows a similar pattern:

- The external system/caller must first authenticate
- Just as the UI authentication module above, the Web Services interface module (WSim) decides whether the user is who they say they are, and which e-person record they correspond to. They then bless this connection however they see fit; this may involve giving some magic number/credential of some description to the external system. That is specific to the Web Services interface.
- Now the external system can invoke a Web Services method.
- The WSim now invokes the asset store API as required, indicating that it is e-person X trying to do this.
- The asset store invokes the authorisation API to ensure it's allowed, proceeding as appropriate, as above.