DSpace 2.0 Core Services

The DS2 (DSpace+2.0) core services are the main services that make up a DS2 system. These includes services for things like user and permissions management and storage and caching. These services can be used by any developer writing DS2 plugins (e.g. statistics), providers (e.g. Authn), or user interfaces (e.g. JSPUI).

Core Services

The core services are all behind APIs so that they can be reimplemented without affecting developers who are using the services. Most of the services have plugin/provider points so that customizations can be added into the system without touching the core services code. For example, let's say a deployer has a specialized authentication system and wants to manage the authentication calls which come into the system. The implementor can simply implement an AuthenticationProvider and then register it with the DS2 kernel ServiceManager. This can be done at any time and does not have to be done during Kernel startup. This allows providers to be swapped out at runtime without disrupting the DS2 service if desired. It can also speed up development by allowing quick hot redeploys of code during development.

Caching Service

Provides for a centralized way to handle caching in the system and thus a single point for configuration and control over all caches in the system. Provider and plugin developers are strongly encouraged to use this rather than implementing their own caching. The caching service has the concept of scopes so even storing data in maps or lists is discouraged unless there are good reasons to do so.

ConfigurationService

Manages the configuration of the DSpace 2 system. Can be used to manage configuration for providers and plugins also.

EventService

Handles events and provides access to listeners for consumption of events.

MappingService

Since so many parts of the core data in DSpace 2 may be coming from outside the system (users , groups, etc.) there needs to be a way to map the internal ids (e.g. 111) to external ids (e.g. aaronz). This service also allows the external ids to be changed. This is done so that data can be associated with a static and unchanging internal id while still allowing an external system to provide ids which may change. This service is used by core services to manage and lookup ids in an efficient way (like an indexing system) without requiring each service to do this itself.

RequestService

In DS2 a request is the concept of a request (HTTP) or an atomic transaction in the system. It is likely to be an HTTP request in many cases but it does not have to be. This service provides the core services with a way to manage atomic transactions so that when a request comes in which requires mutliple things to happen they can either all suceed or all fail without each service attempting to manage this independently. In a nutshell this simply allows identification of the current request and the ability to discover if it succeeded or failed when it ends. Nothing in the system will enforce usage of the service but we encourage developers who are interacting with the system to make use of this service so they know if the request they are participating in with has succeeded or failed and take appropriate actions.

SimpleStorageService

This is a basic storage service which allows for key based persistent storage of strings. It is truly just a simple storage service but it provides ways to have flexible and customizable user interfaces (where the UI developer does not have to implement their own storage layer) and also provides a bucket to place things like configuration parameters. The implementation underneath the simple storage service is likely to change based on the DS2 installation but it will match the StorageService in most cases.

SessionService

In DS2 a session is like an HttpSession (and generally is actually one) so this service is here to allow developers to find information about the current session and to access information in it. The session identifies the current user (if authenticated) so it also serves as a way to track user sessions. Since we use HttpSession directly it is easy to mirror sessions across multiple servers in order to allow for no-interruption failover for users when servers go offline.

Storage Service

The storage service is the heart of DS2 and handles interactions with the storage system below it. This service is designed to allow abstraction of other storage systems via a mixin driven system. By default, DS2 core would only include a simple in-memory storage service that is appropriate for demostration only. Deployers would choose to tie DS2 into 1 or many storage systems in order to provide the overall repository experience for their users. The tying together of various storage systems (Fedora, JCR, etc.) is the key to the DS2 concept and the idea that repositories should interoperate out of the box.

UserAuthenticationService

Handles all authentication of the user, this is primarily just handling caching and the authn provider stack, it also allows the authz groups to be controlled when the user logs into the system.

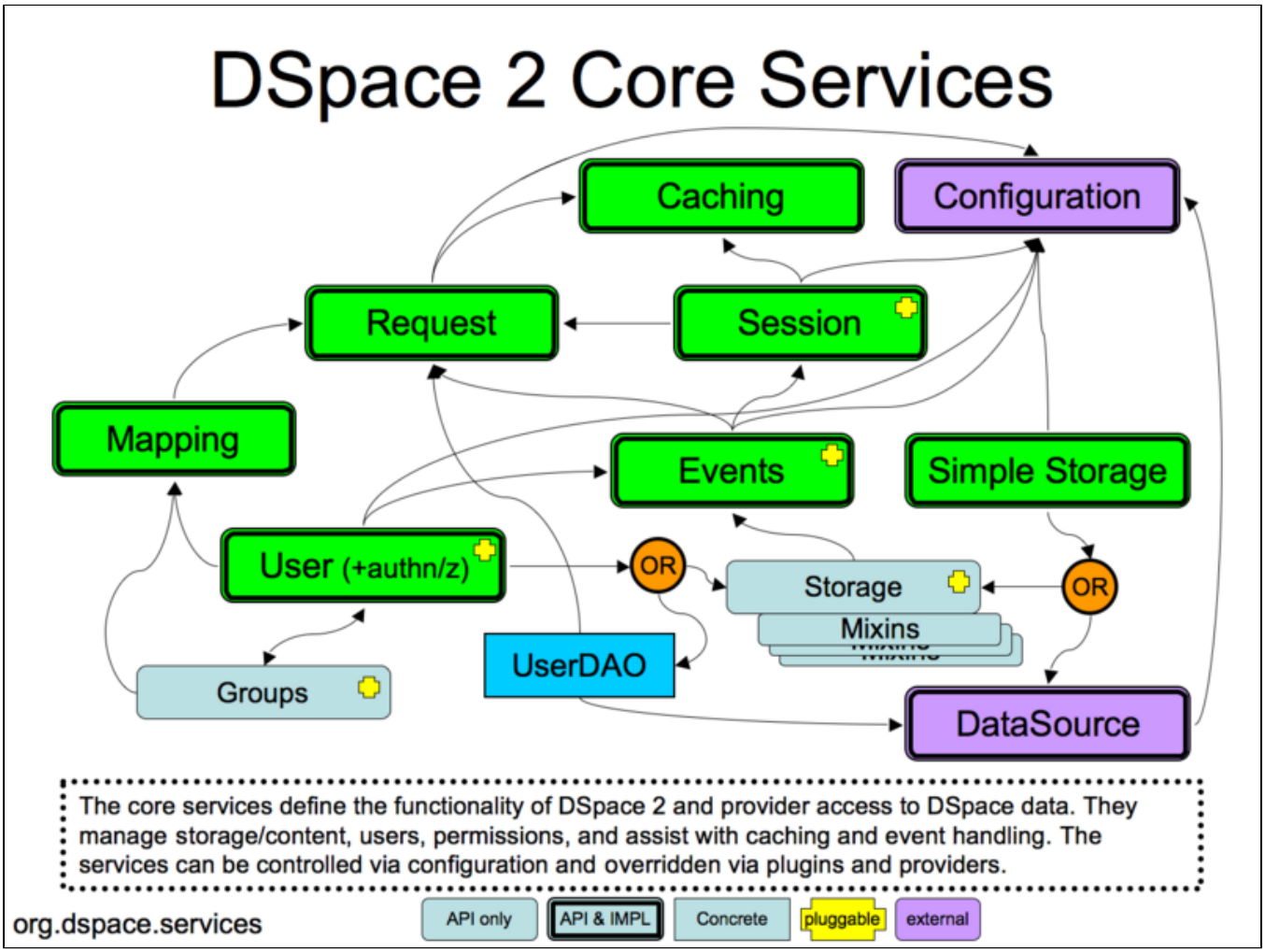
UserAuthorizationService

Handles all Authorization and permissions checks, this also handles the authorization interceptor which allows overrides of all authz checks as desired to provides as much flexibility as is desired when doing permissions checks.

UserService

Provides access to lookup users and their information

This image shows the core services and their dependencies.



Service Layering

Some DS2 core services require additional caching and provider/plugin handling which is managed by stacking implementation of the main service interface. For example, the user service has 3 implementations which are stacked like so:
UserService (API) => CachedUserService => ProvidedUserService => DatabaseUserService

The developer using the service would interface with the UserService API and so would other core services that needed user information. The implementation they would receive would be the caching implementation which doesn't do anything except look into the caches it created at startup with the CachingService. If it does not find anything then it calls down to the next UserService in the chain (it only knows this by the interface). The next one is the ProvidedUserService which only calls out to any registered providers for user lookup. If the providers do not exist or do not have the data requested then it hands the request down the chain again (in some cases it will merge external and internal data but this is a special case). The final piece in the example change is the DatabaseUserService. This looks in the database for the user data which is being requested and returns it. This implementation does not have to worry about providers or caching since this was already handled if needed.

Chaining and separation of concerns

This chaining model keeps the code simpler to understand and concerns are well separated. It also allows for the DatabaseUserService to be swapped out later on for a different type of storage without requiring rewrites of huge parts of the codebase. Some features of Aspect Oriented Programming are preserved here without the complexity and confusion that AOP tends to lead to for many developers.