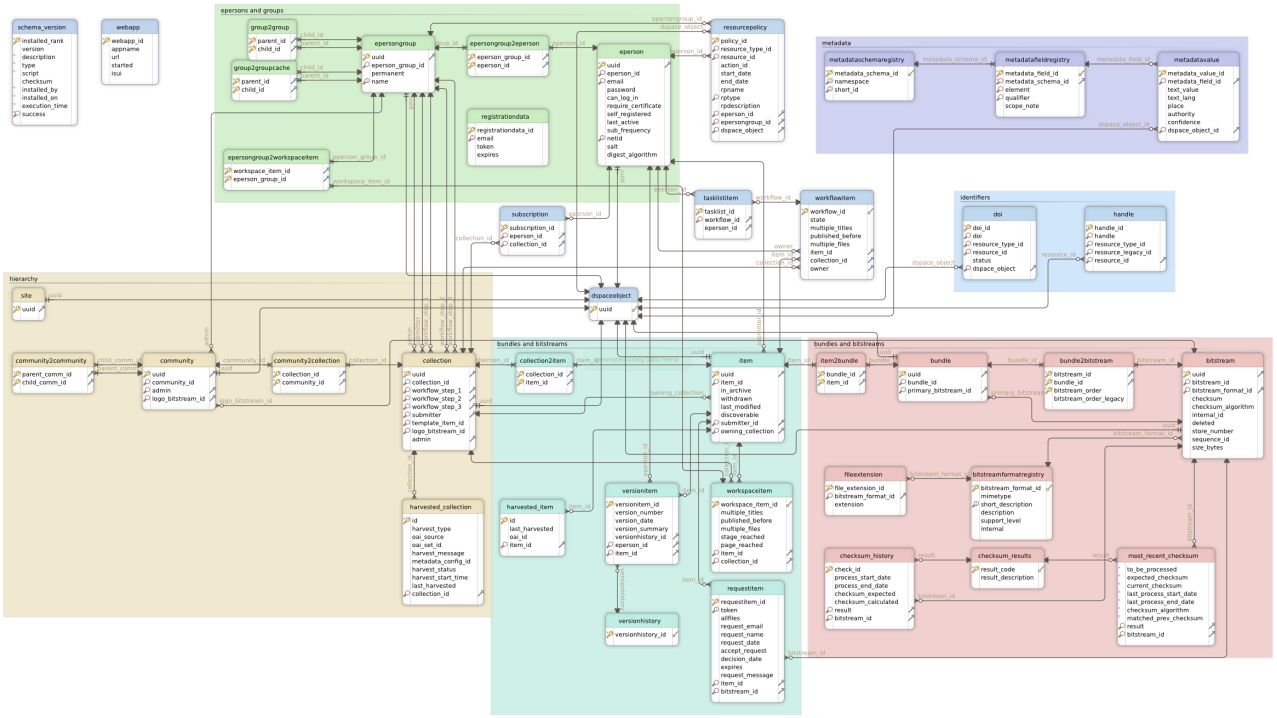
Storage Layer

In this section, we explain the storage layer: the database structure, maintenance, and the bitstream store and configurations. The bitstream store, also known as assetstore or bitstore, holds the uploaded, ingested, or generated files (documents, images, audio, video, datasets, ...), where as the database holds all of the metadata, organization, and permissions of content.

RDBMS / Database Structure

DSpace uses a relational database to store all information about the organization of content, metadata about the content, information about e-people and authorization, and the state of currently-running workflows.

DSpace 6 database schema (PostgreSQL). Right-click the image and choose "Save as" to save in full resolution. Instructions on updating this schema diagram are in How to update database schema diagram.



- DSpace uses FlywayDB to perform automated database initialization and upgrades. Flyway's role is to initialize the database tables (and default content) prior to Hibernate initialization.
  - The `org.dspace.storage.rdbms.DatabaseUtils` class manages all Flyway API calls, and executes the SQL migrations under the `org.dspace.storage.rdbms.sqlmigration` package and the Java migrations under the `org.dspace.storage.rdbms.migration` package.
  - Once all database migrations have run, a series of Flyway Callbacks are triggered to initialize the (empty) database with required default content. For example, callbacks exist for adding default DSpace Groups (`GroupServiceInitializer`), default Metadata & Format Registries (`DatabaseRegistryUpdater`), and the default Site object (`SiteServiceInitializer`). All Callbacks are under the `org.dspace.storage.rdbms` package.
  - While Flyway is automatically initialized and executed during startup, various Database Utilities are also available on the command line. These utilities allow you to manually trigger database upgrades or check the status of your database.
- DSpace uses Hibernate ORM as the object relational mapping layer between the DSpace database and the DSpace code.
  - The main Hibernate configuration can be found at `[dspace]/config/hibernate.cfg.xml`

- Hibernate initialization is triggered via Spring (beans) defined `[dspace]/config/spring/api/core-hibernate.xml`. This Spring configuration pulls in some settings from DSpace Configuration, namely all Database (db.*) settings defined there.
- All DSpace Object Classes provide a DAO (Data Access Object) implementation class that extends a GenericDAO interface defined in o `rg.dspace.core.GenericDAO` class. The default (abstract) implementation is in `org.dspace.core.AbstractHibernateDAO` class.
- The DSpace Context object (`org.dspace.core.Context`) provides access to the configured `org.dspace.core.DBConnection` (Database Connection), which is HibernateDBConnection by default. The `org.dspace.core.HibernateDBConnection` class provides access the the Hibernate Session interface (org.hibernate.Session) and its Transactions.
  - Each Hibernate Session opens a single database connection when it is created, and holds onto it until the Session is closed. A Session may consist of one or more Transactions. Sessions are NOT thread-safe (so individual objects cannot be shared between threads).
  - Hibernate will intelligently cache objects in the current Hibernate Session (on object access), allowing for optimized performance.
  - DSpace provides methods on the Context object to specifically remove (`Context.uncacheEntity()`) or reload (`Context.reloadEntity()`) objects within Hibernate's Session cache.
  - DSpace also provides special Context object "modes" to optimize Hibernate performance for read-only access (`Mode.READ_ONLY`) or batch processing (`Mode.BATCH_EDIT`). These modes can be specified when constructing a new Context object.

Most of the functionality that DSpace uses can be offered by any standard SQL database that supports transactions. However, at this time, DSpace only provides Flyway migration scripts for PostgreSQL and Oracle (and has only been tested with those database backends). Additional database backends should be possible, but would minimally require creating custom Flyway migration scripts for that database backend.

Maintenance and Backup

When using PostgreSQL, it's a good idea to perform regular 'vacuuming' of the database to optimize performance. By default, PostgreSQL performs automatic vacuuming on your behalf. However, if you have this feature disabled, then we recommend scheduling the `vacuumdb` command to run on a regular basis.

```
# clean up the database nightly
40 2 * * * /usr/local/pgsql/bin/vacuumdb --analyze dspace > /dev/null 2>&1
```

Backups: The DSpace database can be backed up and restored using usual PostgreSQL Backup and Restore methods, for example with `pg_dump` and `psql`. However when restoring a database, you will need to perform these additional steps:

- After restoring a backup, you will need to reset the primary key generation sequences so that they do not produce already-used primary keys. Do this by executing the SQL in `[dspace]/etc/postgres/update-sequences.sql`, for example with:

  ```
  psql -U dspace -f [dspace]/etc/update-sequences.sql
  ```

Configuring the Database Component

The database manager is configured with the following properties in `dspace.cfg`:

`db.url`	The JDBC URL to use for accessing the database. This should not point to a connection pool, since DSpace already implements a connection pool.
`db.driver`	JDBC driver class name. Since presently, DSpace uses PostgreSQL-specific features, this should be *org.postgresql.Driver*.
`db.username`	Username to use when accessing the database.
`db.password`	Corresponding password ot use when accessing the database.

Custom RDBMS tables, columns or views

When at all possible, we recommend creating custom database tables or views within a *separate schema* from the DSpace database tables. Since the DSpace database is initialized and upgraded automatically using Flyway DB, the upgrade process may stumble or throw errors if you've directly modified the DSpace database schema, views or tables. Flyway itself assumes it has full control over the DSpace database schema, and it is not "smart" enough to know what to do when it encounters a locally customized database.

That being said, if you absolutely need to customize your database tables, columns or views, it is possible to create *custom Flyway migration scripts* , which should make your customizations easier to manage in future upgrades. (Keep in mind though, that you may still need to maintain/update your custom Flyway migration scripts if they ever conflict directly with future DSpace database changes. The only way to "future proof" your local database changes is to try and make them as independent as possible, and avoid directly modifying the DSpace database schema as much as possible.)

If you wish to add custom Flyway migrations, they may be added to the following locations:

- Custom Flyway SQL migrations may be added anywhere under the `org.dspace.storage.rdbms.sqlmigration` package (e.g. `[src]/dspace-api/src/main/resources/org/dspace/storage/rdbms/sqlmigration` or subdirectories)
- Custom Flyway Java migrations may be added anywhere under the `org.dspace.storage.rdbms.migration` package (e.g. `[src]/dspace-api/src/main/java/org/dspace/storage/rdbms/migration/` or subdirectories)
- Additionally, for backwards support, custom SQL migrations may also be placed in the `[dspace]/etc/[db-type]/` folder (e.g. `[dspace]/etc/postgres/` for a PostgreSQL specific migration script)

Adding Flyway migrations to any of the above location will cause Flyway to auto-discover the migration. It will be run in the order in which it is named. Our DSpace Flyway script naming convention follows Flyway best practices and is as follows:

- SQL script names: `V[version]_[date]__[description].sql`
  - E.g. `V5.0_2014.09.26__DS-1582_Metadata_For_All_Objects.sql` is a SQL migration script created for DSpace 5.x (`V5.0`) on Sept 26, 2014 (`2014_09_24`). Its purpose was to fulfill the needs of ticket DS-1582, which was to migrate the database in order to support adding metadata on all objects.
  - More examples can be found under the `org.dspace.storage.rdbms.sqlmigration` package
- Java migration script naming convention: `V[version]_[date]__[description].java`
  - E.g. `V5_0_2014_09_25__DS_1582_Metadata_For_All_Objects_drop_constraint.java` is a Java migration created for DSpace 5.x (`V5_0`) on Sept 25, 2014 (`2014_09_25`). Its purpose was to fulfill the needs of ticket DS-1582, specifically to drop a few constraints.
  - More examples can be found under the `org.dspace.storage.rdbms.migration` package
- Flyway will execute migrations in order, based on their Version and Date. So, V1.x (or V1_x) scripts are executed first, followed by V3.0 (or V3_0), etc. If two migrations have the same version number, the date is used to determine ordering (earlier dates are run first).

Bitstream Store

DSpace offers two means for storing content.

1. Storage in a mounted file system on the server (DSBitStore)
2. Storage using AWS S3 (Simple Storage Service), (S3BitStore).

Both are achieved using a simple, lightweight BitStore API, providing actions of Get, Put, About, Remove. Higher level operations include Store, Register, Checksum, Retrieve, Cleanup, Clone, Migrate. Digital assets are stored on the bitstores by being transferred to the bitstore when it is uploaded or ingested. The exception to this is for "registered" objects, that the assets are put onto the filesystem ahead of time out-of-band, and during ingest, it just maps the database to know where the object already resides. The storage interface is such that additional storage implementations (i.e. other cloud storage providers) can be added with minimal difficulty.

DSBitStore stores content on a path on the filesystem. This could be locally attached normal filesystem, a mounted drive, or a mounted networked filesystem, it will all be treated as a local filesystem. All DSpace needs to be configured with for a filesystem, is the filesystem path, i.e. /dspace/assetstore, /opt/data/assetstore. The DSBitStore uses a "Directory Scatter" method of storing an asset within 3 levels of subfolders, to minimize any single folder having too many objects for normal filesystem performance.

S3BitStore uses Amazon Web Services S3 (Simple Storage Service) to offer limitless cloud storage into a bucket, and each distinct asset will have a unique key. S3 is a commercial service (costs money), but is available at low price point, and is fully managed, content is automatically replicated, 99.999999999% object durability, integrity checked. Since S3 operates within the AWS network, using other AWS services, such virtual server on EC2 will provide lower network latency than local "on premises" servers. Additionally there could be some in-bound / out-bound bandwidth costs associated with DSpace application server outside of the AWS network communicating with S3, compared to AWS-internal EC2 servers. S3 has a checksum computing operation, in which the S3 service can return the checksum from the storage service, without having to shuttle the bits from S3, to your application server, and then computing the checksum. S3BitStore requires an S3 bucketName, accessKey, secretKey, and optionally specifying the AWS region, or a subfolder within the bucket.

There can be multiple bitstream stores. Each of these bitstream stores can be traditional storage or S3 storage. This means that the potential storage of a DSpace system is not bound by the maximum size of a single disk or file system and also that filesystem and S3storage can be combined in one DSpace installation. Both filesystem and S3 storage are specified by configuration. Also see Configuring the Bitstream Store below.

Stores are numbered, starting with zero, then counting upwards. Each bitstream entry in the database has a store number, used to retrieve the bitstream when required. An example of having multiple asset stores configured is that assetstore0 is /dspace/assetstore, when the filesystem gets nearly full, you could then configure a second filesystem path assetstore1 at /data/assetstore1, later, if you wanted to use S3 for storage, assetstore2 could be s3://dspace-assetstore-xyz. In this example various bitstreams (database objects) refer to different assetstore for where the files reside. It is typically simplest to just have a single assetstore configured, and all assets reside in that one. If policy dictated, infrequently used masters could be moved to slower/cheaper disk, where as access copies are on the fastest storage. This could be accomplished through migrating assets to different stores.

Bitstreams also have an 38-digit internal ID, different from the primary key ID of the bitstream table row. This is not visible or used outside of the bitstream storage manager. It is used to determine the exact location (relative to the relevant store directory) that the bitstream is stored in traditional storage. The first three pairs of digits are the directory path that the bitstream is stored under. The bitstream is stored in a file with the internal ID as the filename.

For example, a bitstream with the internal ID *12345678901234567890123456789012345678* is stored in the directory:

```
[dspace]/assetstore/12/34/56/12345678901234567890123456789012345678
```

The reasons for storing files this way are:

- Using a randomly-generated 38-digit number means that the 'number space' is less cluttered than simply using the primary keys, which are allocated sequentially and are thus close together. This means that the bitstreams in the store are distributed around the directory structure, improving access efficiency.
- The internal ID is used as the filename partly to avoid requiring an extra lookup of the filename of the bitstream, and partly because bitstreams may be received from a variety of operating systems. The original name of a bitstream may be an illegal UNIX filename.

- When storing a bitstream, the *BitstreamStorageService* DOES set the following fields in the corresponding database table row:
  - *bitstream_id*
  - *size*
  - *checksum*
  - *checksum_algorithm*
  - *internal_id*
  - *deleted*
  - *store_number*
- The remaining fields are the responsibility of the *Bitstream* content management API class.

The bitstream storage manager is fully transaction-safe. In order to implement transaction-safety, the following algorithm is used to store bitstreams:

1. A database connection is created, separately from the currently active connection in the current DSpace context.
2. An unique internal identifier (separate from the database primary key) is generated.
3. The bitstream DB table row is created using this new connection, with the *deleted* column set to *true*.
4. The new connection is _commit_ted, so the 'deleted' bitstream row is written to the database
5. The bitstream itself is stored in a file in the configured 'asset store directory', with a directory path and filename derived from the internal ID
6. The *deleted* flag in the bitstream row is set to *false*. This will occur (or not) as part of the current DSpace *Context*.

This means that should anything go wrong before, during or after the bitstream storage, only one of the following can be true:

- No bitstream table row was created, and no file was stored
- A bitstream table row with *deleted=true* was created, no file was stored
- A bitstream table row with *deleted=true* was created, and a file was stored
  None of these affect the integrity of the data in the database or bitstream store.

Similarly, when a bitstream is deleted for some reason, its *deleted* flag is set to true as part of the overall transaction, and the corresponding file in storage is *not* deleted.

Cleanup

The above techniques mean that the bitstream storage manager is transaction-safe. Over time, the bitstream database table and file store may contain a number of 'deleted' bitstreams. The *cleanup* method of *BitstreamStorageService* goes through these deleted rows, and actually deletes them along with any corresponding files left in the storage. It only removes 'deleted' bitstreams that are more than one hour old, just in case cleanup is happening in the middle of a storage operation.

This cleanup can be invoked from the command line via the *cleanup* command, which can in turn be easily executed from a shell on the server machine using `[dspace]/bin/dspace cleanup`. You might like to have this run regularly by *cron*, though since DSpace is read-lots, write-not-so-much it doesn't need to be run very often.

```
# Clean up any deleted files from local storage on first of the month at 2:40am
40 2 1 * * [dspace]/bin/dspace cleanup > /dev/null 2>&1
```

Backup

The bitstreams (files) in traditional storage may be backed up very easily by simply 'tarring' or 'zipping' the `[dspace]/assetstore/` directory (or whichever directory is configured in *dspace.cfg*). Restoring is as simple as extracting the backed-up compressed file in the appropriate location.

It is important to note that since the bitstream storage manager holds the bitstreams in storage, and information about them in the database, that a database backup and a backup of the files in the bitstream store must be made at the same time; the bitstream data in the database must correspond to the stored files.

Of course, it isn't really ideal to 'freeze' the system while backing up to ensure that the database and files match up. Since DSpace uses the bitstream data in the database as the authoritative record, it's best to back up the database before the files. This is because it's better to have a bitstream in storage but not the database (effectively non-existent to DSpace) than a bitstream record in the database but not storage, since people would be able to find the bitstream but not actually get the contents.

With DSpace 1.7 and above, there is also the option to backup both files and metadata via the AIP Backup and Restore feature.

Configuring the Bitstream Store

BitStores (aka assetstores) are configured with `[dspace]/config/spring/api/bitstore.xml`

Configuring Traditional Storage

By default, DSpace uses a traditional filesystem bitstore of `[dspace]/assetstore/`

To configure traditional filesystem bitstore, as a specific directory, configure the bitstore like this:

```
<bean name="org.dspace.storage.bitstore.BitstreamStorageService" class="org.dspace.storage.bitstore.
BitstreamStorageServiceImpl">
        <property name="incoming" value="0"/>
    <property name="stores">
```

```
            <map>
                <entry key="0" value-ref="localStore"/>
        </map>
    </property>
</bean>

<bean name="localStore" class="org.dspace.storage.bitstore.DSBitStoreService" scope="singleton">
        <property name="baseDir" value="${dspace.dir}/assetstore"/>
</bean>
```

This would configure store number 0 named localStore, which is a `DSBitStore` (filesystem), at the filesystem path of `${dspace.dir}/assetstore` (i.e. `[dspace]/assetstore/`)

It is also possible to use multiple local filesystems. In the below example, key #0 is localStore at `${dspace.dir}/assetstore`, and key #1 is localStore2 at `/data/assetstore2`. Note that incoming is set to store "1", which in this case refers to localStore2. That means that any new files (bitstreams) uploaded to DSpace will be stored in localStore2, but some existing bitstreams may still exist in localStore.

```
<bean name="org.dspace.storage.bitstore.BitstreamStorageService" class="org.dspace.storage.bitstore.
BitstreamStorageServiceImpl">
    <property name="incoming" value="1"/>
    <property name="stores">
        <map>
            <entry key="0" value-ref="localStore"/>
            <entry key="1" value-ref="localStore2"/>
        </map>
    </property>
</bean>
<bean name="localStore" class="org.dspace.storage.bitstore.DSBitStoreService" scope="singleton">
    <property name="baseDir" value="${dspace.dir}/assetstore"/>
</bean>
<bean name="localStore2" class="org.dspace.storage.bitstore.DSBitStoreService" scope="singleton">
    <property name="baseDir" value="/data/assetstore2"/>
</bean>
```

Configuring Amazon S3 Storage

To use Amazon S3 as a bitstore, add a bitstore entry `s3Store`, using `S3BitStoreService`, and configure it with `awsAccessKey`, `awsSecretKey`, and `bucketName`. NOTE: Before you can specify these settings, you obviously will have to create an account in the Amazon AWS console, and create an IAM user with credentials and privileges to an existing S3 bucket.

```
<bean name="org.dspace.storage.bitstore.BitstreamStorageService" class="org.dspace.storage.bitstore.
BitstreamStorageServiceImpl">
    <property name="incoming" value="1"/>
    <property name="stores">
        <map>
            <entry key="0" value-ref="localStore"/>
            <entry key="1" value-ref="s3Store"/>
        </map>
    </property>
</bean>
<bean name="localStore" class="org.dspace.storage.bitstore.DSBitStoreService" scope="singleton">
    <property name="baseDir" value="${dspace.dir}/assetstore"/>
</bean>
<bean name="s3Store" class="org.dspace.storage.bitstore.S3BitStoreService" scope="singleton">
    <!-- AWS Security credentials, with policies for specified bucket -->
    <property name="awsAccessKey" value=""/>
    <property name="awsSecretKey" value=""/>
    <!-- S3 bucket name to store assets in. example: longsight-dspace-auk -->
    <property name="bucketName" value=""/>
    <!-- AWS S3 Region to use: {us-east-1, us-west-1, eu-west-1, eu-central-1, ap-southeast-1, ... } -->
    <!-- Optional, sdk default is us-east-1 -->
    <property name="awsRegionName" value=""/>
    <!-- Subfolder to organize assets within the bucket, in case this bucket is shared  -->
    <!-- Optional, default is root level of bucket -->
    <property name="subfolder" value=""/>
</bean>
```

The incoming property specifies which assetstore receives incoming assets (i.e. when new files are uploaded, they will be stored in the "incoming" assetstore). This defaults to store 0.

S3BitStore has parameters for awsAccessKey, awsSecretKey, bucketName, awsRegionName (optional), and subfolder (optional).

- `awsAccessKey` and `awsSecretKey` are created from the Amazon AWS console. You'll want to create an IAM user, and generate a Security Credential, which provides you the accessKey and secret. Since you need permission to use S3, you could give this IAM user a quick & dirty policy of AmazonS3FullAccess (for all S3 buckets that you own), or for finer grain controls, you can assign an IAM user to have certain permissions to certain resources, such as read/write to a specific subfolder within a specific s3 bucket.
- `bucketName` is a globally unique name that distinguishes your S3 bucket. It has to be unique among all other S3 users in the world.
- `awsRegionName` is a region in AWS where S3 will be stored. Default is US Eastern. Consider distance to primary users, and pricing when choosing the region.
- `subfolder` is a folder within the S3 bucket, where you could organize the assets to be in. If you wanted to re-use a bucket for multiple purposes (bucketname/assets vs bucketname/backups) or DSpace instances (bucketname/XYZDSpace or bucketname/ABCDSpace or bucketname /ABCDSpaceProduction).

Migrate BitStores

There is a command line migration tool to move all the assets within a bitstore, to another bitstore. `bin/dspace bitstore-migrate`

```
[dspace]/bin/dspace bitstore-migrate
usage: BitstoreMigrate
 -a,--source <arg>        Source assetstore store_number (to lose content). This is a number such as 0 or 1
 -b,--destination <arg>   Destination assetstore store_number (to gain content). This is a number such as 0 or
1.
 -d,--delete              Delete file from losing assetstore. (Default: Keep bitstream in old assetstore)
 -h,--help                Help
 -p,--print               Print out current assetstore information
 -s,--size <arg>          Batch commit size. (Default: 1, commit after each file transfer)

[dspace]/bin/dspace bitstore-migrate -p
store[0] == DSBitStore, which has 2 bitstreams.
store[1] == S3BitStore, which has 2 bitstreams.
Incoming assetstore is store[1]

[dspace]/bin/dspace bitstore-migrate -a 0 -b 1


[dspace]/bin/dspace bitstore-migrate -p
store[0] == DSBitStore, which has 0 bitstreams.
store[1] == S3BitStore, which has 4 bitstreams.
Incoming assetstore is store[1]
```