DAITSS Operations Manual

Last modified: 04/16/2008

This document covers the operation of the DAITSS digital preservation repository system.

Table of Contents

1. Introduction/Overview

1.1. Purpose & Scope

This document serves as a guide to operating all facets of a DAITSS digital archive. If you need assistance installing DAITSS onto your system, refer to the DAITSS Installation Manual.

1.2. Terms Used In This Document

DAITSS: Dark Archive in the Sunshine State – digital preservation repository software developed by the Florida Center for Library Automation.

Information Package: A complete intellectual entity, comprised of data files and a METS descriptor. Every information package in DAITSS is given a unique ID number, called an IEID.

Data file: A content file present in the archive. Every data file in DAITSS is given a unique ID number, called a DFID.

1.3. DAITSS Overview

DAITSS, or Dark Archive In The Sunshine State, is a digital preservation repository system. It endeavors to preserve digital intellectual entities for the very long term. DAITSS can be described as a 'dark' archive because it is not designed for real time access of its content.

DAITSS is a Java application designed to run in a Unix/Linux environment. More details on system requirements are available in the DAITSS Installation Manual.

DAITSS is based on the OAIS (Open Archival Information System) reference model for digital archives. The full OAIS specification can be viewed at: http://public.ccsds.org/publications/archive/650x0b1.pdf

DAITSS uses METS (Metadata Encoding and Transmission Standard) descriptors for all submission information packages. METS is an XML document format. More information on METS is available from the Library of Congress: http://www.loc.gov/standards/mets/

1.3.1 DAITSS Components

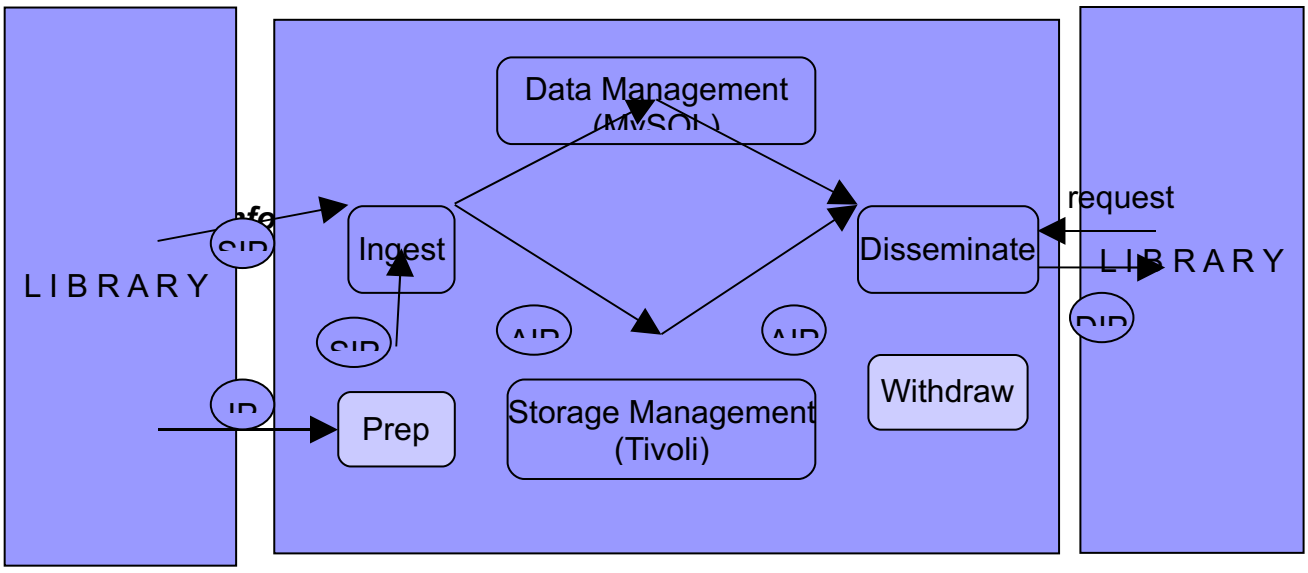
The DAITSS system has three major components: The application itself, a MySQL database, and a storage backend.

The DAITSS application is a Java application which handles all DAITSS functionality. There is a command line interface to run and manage each of the DAITSS functions. These are covered in detail in the chapters dedicated to each of the functions. Also, There is a component of the DAITSS application that runs as a daemon and is used to generate IEIDs and DFIDs for information packages and data files.

DAITSS relies on a relational database to manage its archival collections. As of DAITSS 1.2, MySQL is the only supported relational database management system. For DAITSS to run properly, a MySQL database must be set up and configured. Database setup is covered in the DAITSS Installation Manual; configuration is covered in the DAITSS Configuration File Manual.

The storage backend is where DAITSS stores the information packages. DAITSS can natively write to a Unix filesystem or IBM Tivoli Storage Manager system. Configuration of the storage backend is covered in chapter 3.

The DAITSS daemon, known as OIDServer, runs as a system service in the background and is a necessary part of the DAITSS application. It relies on Java Remote Method Invocation (RMI), so the Java `rmiregistry` must be run as well. There is more detail on the configuration of both OIDServer and `rmiregistry` in the DAITSS Installation Manual.



The primary function of DAITSS is to preserve information packages. An information package is defined as a complete intellectual entity, comprised of data files and a descriptor containing the metadata pertaining to that entity. Information packages take 3 general forms:

Submission Information Package (SIP): A SIP is an information package that is submitted to be ingested into the archive. It contains the data files comprising the intellectual entity, as well as an owner provided METS descriptor.

Archival Information Package (AIP): An AIP is an information package that is created by DAITSS to be archived. It contains the entire originally submitted SIP, any extra metadata and technical information extracted from the information package during Ingest, and transformed versions of the original files.

Dissemination Information Package (DIP): A DIP is an information package that has been disseminated from the archive to be delivered to the authorized requester.

1.3.3. DAITSS Preservation Levels

DAITSS supports two levels of preservation, *bit-level* and *full*. Bit-level preservation includes secure storage, data monitoring, and the creation of multiple masters. Full preservation includes bit-level treatment plus format transformations where applicable.

1.3.4. DAITSS Functions

DAITSS implements four major functions: Prep, Ingest, Dissemination, and Withdrawal.

Prep is an optional first step; preparing information packages for Ingest into the archive.

The Ingest function enters Submission Information Packages (SIPs) into the archive. A

report is generated after a successful Ingest. The core preservation function, Ingest is covered in detail in Chapter 5.

Dissemination is used to access information packages previously Ingested. Essentially, an Archival Information Package (AIP) is converted into a Dissemination Information Package (DIP) and delivered to the owner. The original AIP persists in the archive. A report is created after a successful Dissemination. Dissemination is covered in detail in Chapter 6.

Withdrawal is the permanent removal of an AIP from the archive. Withdrawals can be initiated by either the information package owner or archive management. A report is created after a successful Withdrawal. Withdrawal is covered in detail in Chapter 7.

2. Accounts and Projects

DAITSS can preserve information packages from any number of different submitters. Submitters must establish agreements with the archive indicating the materials to be archived and the preservation level to be accorded them. Each agreement corresponds to one account represented by an Account code. Each Account can designate contact persons for administrative issues, technical issues, and reporting.

Each submitter must define the desired preservation treatment of materials submitted in an SIP for that Account. This is done by specifying for each digital format (MIME type) `full`, `bit` or `none`. For example, a submitter can specify that files of type "application/pdf" receive full preservation treatment but "image/png" receive only bit preservation. A default value is defined per account for any file type not otherwise listed.

Project is an arbitrary code which can be used to designate a publication, collection, or any set of materials. For example, the submitter could specify that PDF files for project "perm" should receive full preservation treatment, and all other PDF files receive bit preservation. Project codes must be predefined in the DAITSS database. They are associated with incoming materials by the `PROJECT` attribute in the `AGREEMENT_INFO` element in the SIP descriptor.

DAITSS includes a provision to allow subaccount codes to be defined, but subaccount codes are not currently used.

2.1. First time configuration

After an initial install of DAITSS, contact, accounts, projects, and preservation rules must be added before use.

Due to foreign key constraints, these must be added in specific order. The correct order for initial DAITSS configuration is:

1. Contact (see 2.5.1)
2. Account (see 2.2.1)
3. Project (see 2.3.1.1)
4. Account/Project mapping (see 2.3.1.2)
5. Correct Contact/Account permissions (see 2.6.1.1)
6. Preservation rule (see 2.4.1)

2.2. Accounts

- A code to identify the account
- The name of the account
- A description of the account.
- An administrative contact.
- A technical contact.
- An email address to receive reports.

2.2.1 Maintenance

Accounts are maintained in the ACCOUNT table in the MySQL database. The following example demonstrates adding an Account.

```
/* add an account for the library of Alexandria with
contacts that have Ids 2 and 3 */
INSERT INTO ACCOUNT (CODE, NAME, DESCRIPTION,
ADMIN_CONTACT, TECH_CONTACT, REPORT_EMAIL) VALUES ('LOA',
'Library of Alexandria', 'Archives of the Library of
Alexandria in case something happens', 2, 4,
'archive@loa.org.eg');
```

2.3. Projects

- A code to identify the project.
- A description of the project.

2.3.1. Maintenance

Projects are maintained in the ACCOUNT table in the MySQL database. The following example demonstrates adding an Project.

2.3.1.1. Adding Projects

```
/* add an project for the antiquities project */
```

```
INSERT INTO PROJECT (CODE, DESCRIPTION) VALUES ('ATQ',
'Antiquities of the World');
```

2.3.1.2. Relating Accounts to Projects

```
/* relate the antiquities project with the library of
alexandria */

INSERT INTO ACCOUNT_PROJECT (ACCOUNT, PROJECT) VALUES
('LOA', 'ATQ');
```

2.4. Preservation

Preservation Level describes to what extent a Digital Object is preserved. A Preservation Level of a Digital Object is determined by:

- The Account listed in SIP that contains the Digital Object.
- The Project listed in SIP that contains the Digital Object.
- The Level of preservation: full, bit, or none.
- The Format of the Digital Object.
- An interval of time for which the preservation rule is in force.

The format the preservation level applies to is determined by MIME type. There is a "default" MIME type recognized by DAITSS, which is useful for specifying a default preservation level for file types that do not have an explicit preservation rule. If a default preservation level is to be used, one must be added for every account/project combination.

2.4.1. Maintenance

Preservation Levels are maintained in the ARCHIVE_LOGIC table in the MySQL database. The following examples demonstrate adding Preservation Levels.

```
/* add preservation level FULL for as default for Account/
Project ID 5, which will apply to all file types in this
project that do not have explicit preservation levels of
their own. */

INSERT INTO ARCHIVE_LOGIC (ACCOUNT_PROJECT, START_DATE,
MEDIA_TYPE, PRES_LEVEL) VALUES (5,  NOW(), 'default',
'FULL');


/* add preservation level FULL for video/mpeg in the
Account/Project of ID 5 starting immediately with no
expiration */

INSERT INTO ARCHIVE_LOGIC (ACCOUNT_PROJECT, START_DATE,
```

```
MEDIA_TYPE, PRES_LEVEL) VALUES (5,  NOW(), 'video/mpeg',
'FULL');


/* add preservation level BIT for video/avi in the Account/
Project of ID 4 starting immediately with no expiration */

INSERT INTO ARCHIVE_LOGIC (ACCOUNT_PROJECT, START_DATE,
MEDIA_TYPE, PRES_LEVEL) VALUES (4,  NOW(), 'video/avi',
'BIT');


/* add preservation level NONE for video/quicktime in the
Account/Project of ID 3 starting immediately with an
expiration of 6 months */

INSERT INTO ARCHIVE_LOGIC (ACCOUNT_PROJECT, START_DATE,
END_DATE, MEDIA_TYPE, PRES_LEVEL) VALUES (3,  NOW(), NOW()
+ INTERVAL 6 MONTH, 'video/quicktime', 'BIT');
```

2.5. Contacts

A contact within DAITSS is defined as:

- A name for the contact.
- A street address.
- An email address.
- a phone number.

2.5.1. Maintenance

Contacts are maintained in the CONTACT table in the MySQL database. The following examples demonstrate adding Contacts.

```
/* add contacts */

INSERT INTO CONTACT (NAME, ADDR_L1, ADDR_L1, EMAIL, PHONE)
VALUES ('Ptolemy II',  '1123', 'Alexandria, Egypt',
'123-4321', 'ptolemy2@loa.org.eg');

INSERT INTO CONTACT (NAME, ADDR_L1, ADDR_L1, EMAIL, PHONE)
VALUES ('Demetrius of Phaleron',  '5813', 'Athens, Greece',
'987-6789', 'dphaleron@loa.org.eg');
```

2.6. Permissions

Withdrawal and Dissemination require that permissions be set up in the DAITSS database. This requires that an entry be inserted into the `OUTPUT_REQUEST` table defining:

- A contact to grant permission to.

- An Account to grant permission on.
- Withdraw permission, true if and only if the contact can disseminate from this account.
- Disseminate permission, true if and only if the contact can disseminate from this account.

2.6.1. Maintenance

Permissions to invoke DAITSS functions are defined in the `OUTPUT_REQUEST` table in the MySQL database.

2.6.1.1. Adding Initial Permissions

Initial Permissions must be granted before any authorization of withdrawal or disseminate may be granted. Initial permissions are restricted to not allow dissemination or withdrawal. The following example grants initial permissions to a user:

```
/* grant initial restrictive permissions to a contact with
id of 5 on account LOA */

INSERT INTO OUTPUT_REQUEST (ACCOUNT, CONTACT) VALUES (5,
'LOA');
```

2.6.1.2. Changing Permissions

The following examples demonstrate how to enable or disable permissions of a user on an account to withdraw or disseminate:

```
/* enable withdrawal for the contact with id of 5 on
account LOA */

UPDATE OUTPUT_REQUEST SET CAN_REQUEST_WITHDRAWAL = 'TRUE'
WHERE CONTACT = 5 AND ACCOUNT = 'LOA';


/* disable dissemination for the contact with id of 5 on
account LOA */

UPDATE OUTPUT_REQUEST SET CAN_REQUEST_DISSEMINATION =
'FALSE' WHERE CONTACT = 5 AND ACCOUNT = 'LOA';


/* enable withdrawal and disseminate for the contact with
id of 5 on account LOA */

UPDATE OUTPUT_REQUEST SET CAN_REQUEST_WITHDRAWAL = 'TRUE',
CAN_REQUEST_DISSEMINATION = 'FALSE' WHERE CONTACT = 5 AND
ACCOUNT = 'LOA';
```

3. Archival Storage System

DAITSS data files are stored in the Archival Storage System. Archival Storage is a set of Storage Instances accessible to DAITSS.

3.1. Storage

A Storage Instance identifies a container for data files. They are defined in the DAITSS database table STORAGE_INSTANCE. All Storage Instances are used to store data files. Your archive should be set up to store data files into multiple storage instances for redundancy.

Two types of storage instances are supported. `TIVOLI` for IBM Tivoli Storage Manager Backup and Archive and `FILE` for filesystems.

A Storage Instance is defined as:

- A storage instance name that identifies the Storage Instance relative to the storage method.
  - *FILE*: the storage instance name is treated as a path to a directory where all stored digital objects are rooted.
  - *TIVOLI*: the storage instance name matches a pattern `${`*node*`}@${`*server*`}:${`*management_class*`}` where *node* is the Tivoli node name of the machine running DAITSS; *server* is the Tivoli Storage Management Server; *management_class* is the Tivoli management class for the Storage Instance.
- A state describing if the Storage Instance is enabled or disabled. A DAITSS function will use a Storage Instance if and only if the Storage Instance is enabled. All enabled storage instances will be used to store data files.
- An optional description of the storage insance.

The following URL identifies a Storage Instance of type *FILE* rooted at `/var/daitss/storage_1`: `FILE:///var/daitss/storage_1`.

3.2. Maintenance

Storage Instances are maintained in the `STORAGE_INSTACE` table in the MySQL database.

3.2.1. Adding Storage Instances

Adding a Storage Instance is done through a MySQL query. The following example will add a storage instance of type *FILE* to the system:

```
INSERT INTO STORAGE_INSTANCE (METHOD, INSTANCE,
DESCRIPTION)

VALUES ('FILE', '/var/daitss/storage_1', 'An example
filesystem storage instance');
```

3.2.2. Enabling and Disabling Storage Instances

Adding a Storage Instance is done through a MySQL query. The following examples will enable and disable a storage instances:

```
/* enable the storage instance */

UPDATE STORAGE_INSTANCE SET ENABLED = 'TRUE' WHERE METHOD =
'FILE' AND INSTANCE = '/var/daitss/storage_1';


/* disable the storage instance */

UPDATE STORAGE_INSTANCE SET ENABLED = 'FALSE' WHERE METHOD
= 'FILE' AND INSTANCE = '/var/daitss/storage_1';
```

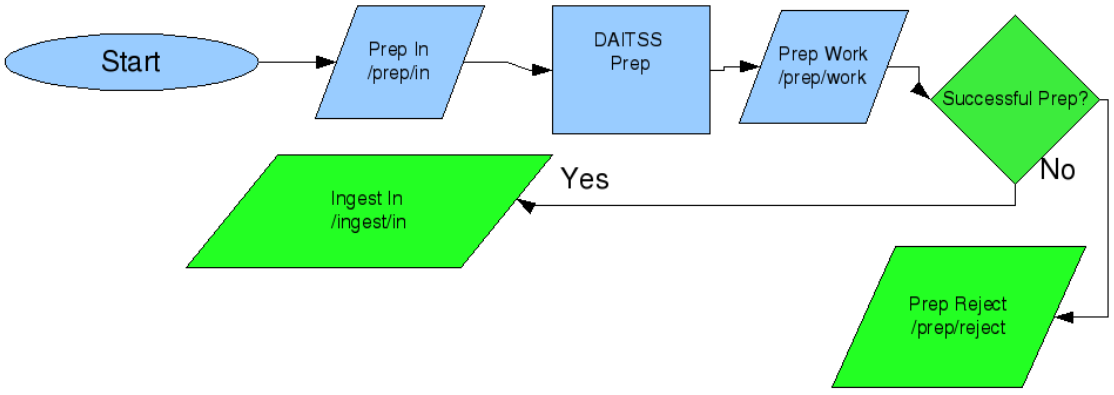
4. Prep

4.1. Prep Overview

4.1.1. Introduction to Prep

Below is a high level diagram depicting information package flow on Prep.

Prep is a function that serves to ensure that an information package is ready for Ingest.



Prep will:

- Verify that an information package is not empty.
- Verify that an information package contains a METS descriptor. In specific cases, a METS descriptor can be automatically generated.
- Verify that the information package only contains the files described in descriptor .
- Verify that the descriptor has proper account information.
- If applicable, extract the information package from a zip file.

Prep also ensures that the provided descriptor conforms to the Metadata Encoding and Transmission Standard (METS). METS is an XML application. In the event that an MXF descriptor is provided, Prep is able to convert the MXF descriptor into METS.

Once the presence and content of the descriptor is verified, Prep can use the descriptor to perform an initial audit of the information package – ensuring that all files therein are actually described by the descriptor. In addition, Prep checks the message digests of all included files to ensure that they match up.

Finally, Prep ensures that there exists proper DAITSS account and project information. More information on DAITSS accounts and projects is provided in Chapter 2.

4.1.2. Prep and Zip Files

Prep is capable of processing information packages within zip files. The zip file must share the name of the package it contains. For example, a zip file containing a package named “foobar” should be named “foobar.zip.”

Packages contained in zip files are placed in the `prep/in` directory for processing. Prep will automatically extract and place the contents inside `/ingest/in` and delete the zip file.

4.2. Prep Usage

The general command to start Prep is:

```
prep start
```

By default, the executable is located in `/opt/daitss/bin`. There are a number of arguments that can be passed to Prep from the command line. The following parameters must immediately follow the command name:

start: Begins execution of Prep.

kill: Terminates execution of Prep, assuming that it is running

status: Displays the status of Prep – running, or not running

zap: Removes the lock file. Useful when Prep has been stopped by means other than prep kill.

clean: Removes any files present in the Prep Work directory

help: Displays usage information on the console

There are additional optional command line parameters that can be passed to Prep. The following optional parameters can only be used with the `start` argument.

-d: Instructs Prep to automatically generate a METS Descriptor for the information package. When using this parameter, the account and project parameters must also be specified.

-r: Instructs Prep to prune undescribed files from the package. Any file that is removed from the package as a result will be noted in the program output.

-a acct: Specifies the account code as acct. The account code must have been defined in the DAITSS database. In using this parameter, the project parameter must also be used.

-p prj: Specifies the project code as prj. The project code must have been defined in the DAITSS database. In using this parameter, the account parameter must also be used.

Note: Prep will not overwrite existing account/project metadata from an information package descriptor. The -a and -p parameters are only used if the information package has no account/project information specified.

4.3. Prep Usage Examples

4.3.1. Starting Prep

The following example illustrates starting Prep:

```
prep start
Prep start: Starting Prep processing...
```

4.3.2. Stopping Prep

The following example illustrates stopping Prep:

```
prep kill
Prep kill: killing Prep...
```

4.3.3. Prep Run Status

The following example illustrates checking the Prep run status

```
prep status

Prep status: not running
```

4.3.4. Starting Prep, generating descriptors and pruning

The following example illustrates starting Prep, generating descriptors for undescribed information packages with the provided account and project codes, and pruning undescribed files.

```
prep start -d -r -a LDA -p ATQ
```

5. Ingest

5.1. Ingest Overview

The Ingest function processes information packages into the repository. Ingest extracts and validates metadata, then stores all data files in the package. If necessary, data files are migrated and normalized.

5.2. Ingest Workflow

Below is a high level diagram depicting information package flow on Ingest.

When Ingest is run, information packages are copied from the Ingest In directory for processing. Information packages are placed into Ingest In manually by the Operator.

As Ingest runs, it copies the information package it is currently working on to the Ingest Work directory. When the information package is successfully processed, then the information package is copied to permanent archival storage as well as placed in the Ingest Out directory. Should the Ingest function fail, the information package is placed in the Ingest Reject directory.

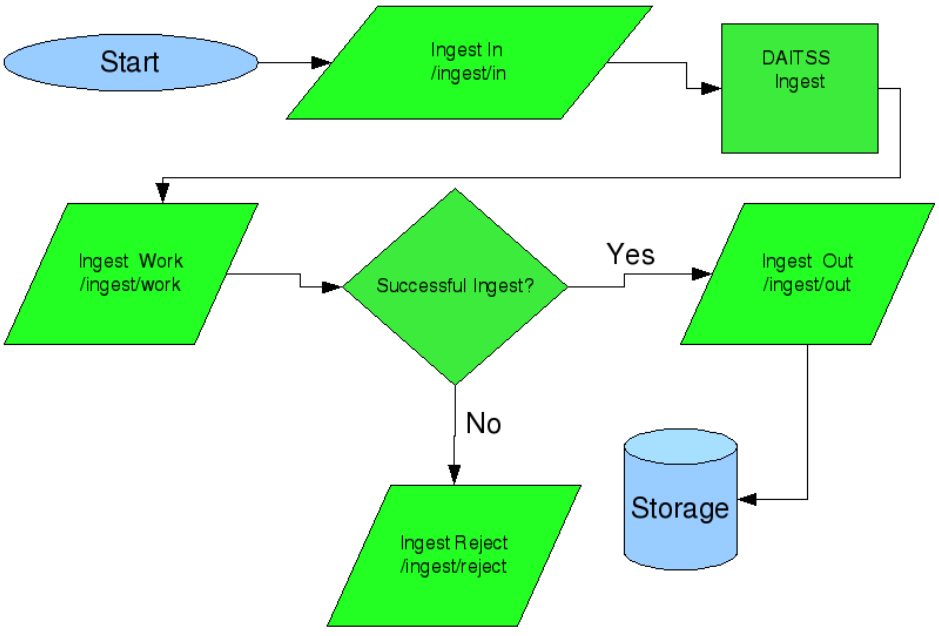
Please note that for certain errors encountered during Ingest, an error report may not be sent out to the reporting e-mail address for the given account. In this case, an error report will be sent to the administrative reporting e-mail account set during installation. This generally happens when an error Ingesting a package occurs that is known to be caused by a system problem.

5.3. Ingest Usage

The general command to start Ingest is:

```
ingest start
```

The executable is located in the DAITSS application bin directory. There are a number of arguments that can be passed to Ingest from the command line. The following parameters



must immediately follow the command name:

start: Beings execution of the Ingest.

kill: Terminates execution of Ingest, assuming that it is running

status: Displays the status of Ingest – running, or not running

zap: Removes the lock file. Useful when Ingest has been stopped by means other than prep kill.

clean: Removes any files present in the Ingest Work directory

help: Displays usage information on the console

Ingest frequently produces a generous amount of output to the console. It is often prudent to redirect the output of Ingest to a log file, and then utilize the Unix `tail` command to display only the most recent output.

5.4. Ingest Usage Examples

5.4.1. Starting Ingest

The following example illustrates starting Ingest:

```
ingest start
Ingestor start: Starting Ingestor...
```

5.4.2. Stopping Ingest

The following example illustrates stopping Ingest:

```
ingest kill
Ingestor kill: killing Ingestor...
```

5.4.3. Ingest Run Status

The following example illustrates checking the run status of Ingest:

```
ingest status

Ingestor status: running
```

5.4.4. Ingest Clean

The following example illustrates cleaning the Ingest work directory:

```
ingest clean
Ingestor clean: cleaning work directory
Work directory files deleted
```

6. Dissemination

6.1. Dissemination Overview

Dissemination is the process by which an authorized user can retrieve an information package from the repository. Both the original and any transformed versions are delivered.

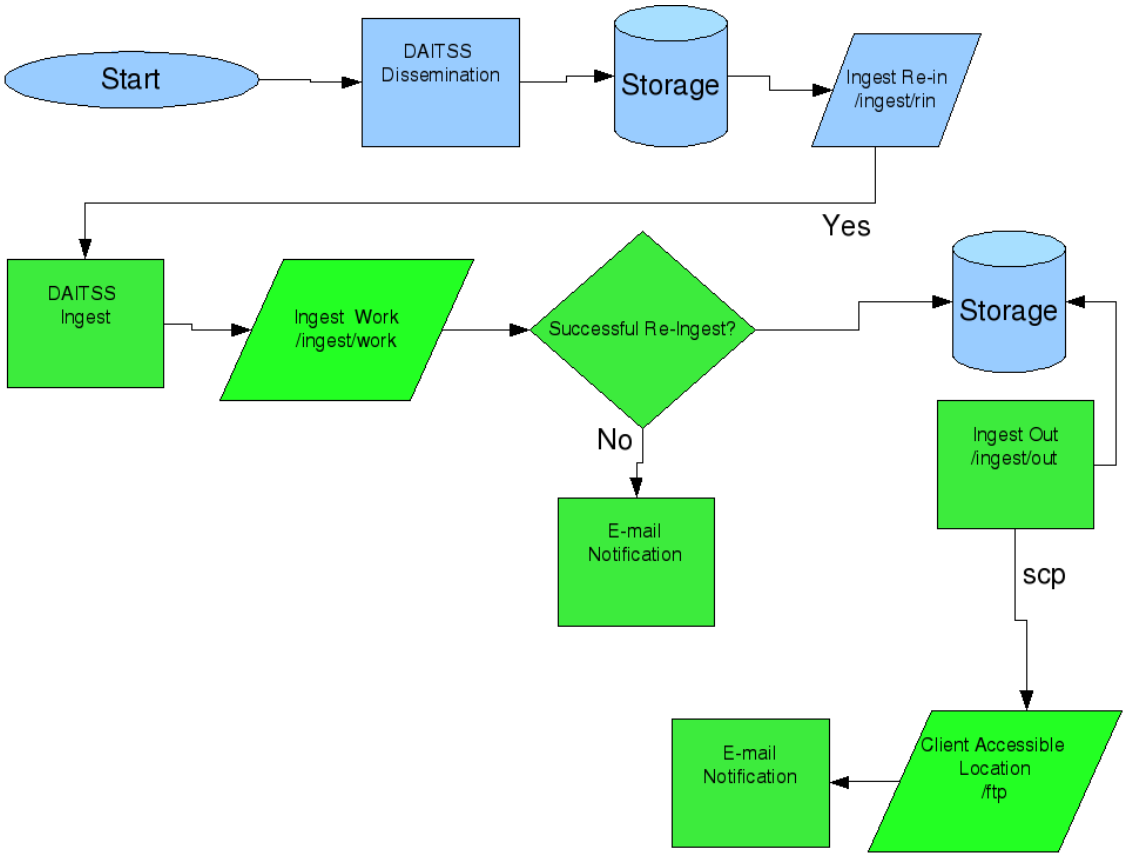
During Dissemination, the information package requested is copied from storage and placed in the `REINGEST_IN` directory. Then, at the next Ingest runtime, the information package is re-ingested into the archive to apply the most recent transformations. After re-ingest, the information package is placed in the `INGEST_OUT` directory and copied to the FTP directory defined in the `daitss.properties` file for the account. In addition, a Dissemination report is delivered to the report contact for the account associated with the information package.

6.2. Dissemination Workflow

Below is a high level diagram depicting information package flow on Dissemination.

The goal of Dissemination is to provide the information package owner with the 'last best' version of the originally submitted information package. Relevant format transformations are applied to the information package. In all cases, the original files provided will also be returned.

When a Dissemination request is received, the relevant information package is pulled from storage and placed into the reingest directory. The next time that an Ingest function is run, the information package is re-ingested. The Re-ingested information package is then zipped and copied into the package owner's FTP directory, and written back to storage. In the event of a rejection during re-ingest, notices are sent to the report e-mail for the account.



6.3. Dissemination Usage

The general command to run Dissemination is:

```
disseminate -c contactID -i IEID -s storageInstance
```
or
```
disseminate -c contactID -f FILE -s storageInstance
```

Command line arguments:

-c: Specifies the contact ID of the individual requesting the Dissemination. In order for a Dissemination to succeed, the requester must have the appropriate permissions. Setting Dissemination permissions is described in detail in section 2.5.

-i: Specifies a single IEID to disseminate. The -i argument cannot be used in conjunction with the -f argument.

-f: Specifies a comma delimited file of IEIDs to disseminate. A small report of which information packages were successfully extracted follows the operation, allowing the operator to identify any problem information packages. The -f argument cannot be used in conjunction with the -i argument.

-s: Specifies a particular storage instance to pull information packages from. This argument is optional. If a storage instance is not specified, DAITSS uses the first active storage instance enabled.

6.4. Usage Examples

6.4.1. Disseminating a Single Package

The following example illustrates disseminating a single information package from the archive.

```
disseminate -c 1 -i E20061108_AAAAAA
```

6.4.2. Disseminating Multiple Packages

The following example illustrates disseminating multiple information packages from the archive. All IEIDs to be disseminated must be listed in a comma delimited text file.

```
disseminate -c 1 -f textfile.txt
```

6.4.3. Disseminating a Single Package from Specified Storage Instance

The following example illustrates disseminating an information package specifying a storage instance.

```
disseminate -c 1 -i E20061108_AAAAAA -s  FILE:///var/
daitss/storage_1
```

7. Withdrawal

7.1. Withdraw Overview

Withdrawal is a mechanism to permanently remove an information package from a DAITSS archive. The result after a Withdrawal operation is that only provenance that the information package once existed in the archive remains.

There are two types of Withdrawals, `account` and `archive`. An `account` Withdrawal is requested by an authorized contact associated with the project. An `archive` Withdrawal is requested by the archive operators. In practice, both types of Withdrawals are executed by the archive operators, but the Withdrawal types help differentiate between a Withdrawal carried out at the request of the owner versus a Withdrawal done by administration.

After a Withdrawal operation, the information package is removed from storage, and all data except for trace data indicating that the information package was once present is deleted. The information package is not re-ingested or otherwise disseminated. A Withdrawal report is generated and delivered to the report contact for the account, whether an `account` or `archive` Withdrawal was performed.

7.2. Withdrawal Usage

The command to run Dissemination is:

`withdraw -t TYPE -c contactID -i IEID` or

withdraw -t TYPE -c contactID -f FILE

Command line arguments:

-t: Specifies the type of Withdrawal to be performed. Valid values are

`archive` or account.

-c: Specifies the contact ID of the individual requesting the Withdrawal. In order for a Withdrawal to succeed, the requester must have the appropriate permissions. Setting Withdrawal permissions is described in detail in section 2.5.

-i: Specifies a single IEID to Withdraw. The -i argument cannot be used in conjunction with the -f argument.

-f: Specifies a comma delimited file of IEIDs to Withdraw. A small report of which information packages were successfully extracted follows the operation, allowing the operator to identify any problem information packages. The -f argument cannot be used in conjunction with the -i argument.

7.3. Withdrawal Usage Examples

7.3.1. Account Withdrawal of a Single Package

The following example illustrates a contact-requested withdrawal of a single package.

```
withdraw -t account -c 1 -i E20061108_AAAAAA
```

7.3.2. Archive Withdrawal of Multiple Packages

The following example illustrates an archive administrative withdrawal of multiple information packages. All IEIDs to be withdrawn must be listed in a comma delimited text file.

```
withdraw -t archive -c 1 -f textfile.txt
```

8. DAITSS Log Files

8.1. Log File List

Name	Location	Notes
Ingest Log	`/var/log/daitss/ingest`	Logs all action taken by Ingest

Dissemination Log	/var/log/daitss/disseminate	Logs all action taken by Dissemination
Withdrawal Log	/var/log/daitss/withdraw	Logs all action taken by Withdrawal
Prep Log	/var/log/daitss/prep	Logs all action taken by Prep
OIDServer Log	/var/log/daitss/service	Logs all action taken by DAITSS OIDServer

8.2 Log File Structure

All log files have the same general structure. Each line in the log file designates a discrete action taken by the DAITSS component that created it. The first space-delimited field is a timestamp with four digit year. Following the timestamp is information about the action taken.

8.3 Sample Log Output

Below is sample output from the Ingest log:

```
20070118213513 L    0    edu.fcla.daitss.Ingestor init()
     Ingestor  Initializing Ingestor

20070118213514 L    0    edu.fcla.daitss.Ingestor process()
     Ingestor  Begin Ingestor processing

20070118213514 L    0    edu.fcla.daitss.entity.SIPList
     build()   IPList    Begin building of IPList

20070118213514 L    0    edu.fcla.daitss.entity.SIPList
     build()   WF00014660     Adding /var/daitss/ingest/rin/
WF00014660 to IPList

20070118213514 L    0    edu.fcla.daitss.entity.SIPList
     process() SIPList   Begin processing of IPList

20070118213514 L    0    edu.fcla.daitss.entity.SIPList
     init()    SIPList   Initializing objects that are not
shared among packages

20070118213515 L    0
     edu.fcla.daitss.database.MySqlConnection
     retrieve(Vector, String, String)   Query: SELECT
```

```
GLOBAL_FILE.DFID, USE_COUNT, VALUE, IEID, PACKAGE_PATH,
FILE_TITLE, FILE_EXT, PRES_LEVEL FROM GLOBAL_FILE,
MESSAGE_DIGEST, DATA_FILE where GLOBAL_FILE.DFID =
MESSAGE_DIGEST.DFID AND CODE = 'sha-1' AND GLOBAL_FILE.DFID
= DATA_FILE.DFID    SQL query

20070118213526 L    0    edu.fcla.daitss.entity.AIP
     ingest(SIP)    SIP: WF00014660     SIP ingest began at
2007-01-18 21:35:26

20070118213526 L    0    edu.fcla.daitss.entity.SIPList
     rejectPackage(InformationPackage)  WF00014660     SIP
rejected
```

Below is sample output from the Prep log:

```
20070102205641 L    0    edu.fcla.daitss.prep.PreProcessor
     process   Preprocessing  Starting preprocessing of
package: /var/daitss/prep/in/CF00000005

20070102205641 L    16
     edu.fcla.daitss.prep.filter.DescriptorExistenceFilter
     process   Checking if packages have a descriptor. No
package descriptor found.:
edu.fcla.daitss.exception.PackageException

20070102205642 L    8    edu.fcla.daitss.prep.PreProcessor
     process   Preprocessing  Rejecting package: CF00000005
```

9. Fixity Check Utility

The DAITSS fixity check utility maintains the fixity of Data Files across all enabled copies in storage.

9.1. Usage

```
% fixitycheck  [DFID...]
```

9.2. Description

The fixity check utility will check and possibly recover corrupted files specified by list of Data File IDs (DFID).

The fixity check utility will calculate the message digest of each stored copy of the Data File for every message digest algorithm supported by the system: md5 and sha1. It will then compare those message digests with the message digests calculated when the data file was ingested.

If all copies of the Data File have the same message digest values that was calculated at ingest the fixity check is passed. A successful Event of type `FC` is recorded.

If some copies' message digests don't match the digests recorded at ingest and at least one copy does, the fixity checker utility will overwrite the corrupted copies with the uncorrupted copy. A successful Event of type FC is recorded, with the details of the recovery recorded as a not.

If all copies in storage are corrupt, recovery is impossible and the DAITSS operator is notified via email. An unsuccessful Event of type `FC` is recorded, with the details of the recovery recorded as a note.

9.3. Logs

The fixity check utility keeps logs of its activities in files contained within the directory specified by the DAITSS property `FIXITY_CHECK_LOG`. The log format is consistent with all other DAITSS logs.

9.4. Exit Status

0	Successful fixity check	The fixity check utility successfully: checked the fixity; passed with possible recovery or failed.
1	Invalid fixity check	The fixity check could not complete due to invalid parameters, i.e. A DFID that does not exist in the archive.
2	Fixity check could not complete	The application crashed, details will be in the fixity check logs.

9.5. Examples

To check a single data file with the DFID F20070611_AAABBB:

```
% fixity check  F20070611_AAABBB
```

To check a multiple data files:

```
% fixity check  F20070611_AAABBC F20050611_AAABBD
F20060611_AAABBE
```