# BitstreamFormat Renovation

This page proposes a set of changes to improve the representation of file formats in the content model, in order to better support preservation activities. The design applies to DSpace 1.5, or later. This work is to be done as part of the FACADE project but it is designed and intended as a generally useful extension to the platform.

Please use the Discussion tab for your comments on and reactions to this proposal, since comments mingled with this much text would be too hard to find.

**See Also** these related wiki pages:

- About Data Formats - background and philosophy
- BitstreamFormat Conversion Instructions  - how to convert a DSpace archive
- BitstreamFormat Workbench  - manual for the command-line admin utility
- Paper presented at OR08: *BitstreamFormat+Renovation_DSpace Gets Real Technical Metadata*
- Slides from BSF Renovation presentation at Open Repositories 2008, Southampton, UK

**Downloads:**
Download Patch to DSpace 1.5 Source

Contents

## Introduction

In *Automatic Format Identification using PRONOM and DROID*, Adrian Brown defines a "data format" as:

> *The internal structure and encoding of a digital object, which allows it to be processed, or to be rendered in human-accessible form.*

Note that this implies more than just knowing the common name of a Bitstream's format, e.g. "Adobe PDF". That name actually describes a family of formats. In order to know *exactly* how to recover the intelligence in a particular Bitstream, you'd want to know which specific *version* of PDF it is: later versions have features not found in earlier ones. The "internal structure and encoding" imposed by a data format is usually defined in exacting detail by a format specification document, and/or by the software applications that produce and consume that format.

For additional, extensive, background, see About Data Formats , which serves as a manifesto of sorts for this project.

Overall, this proposal affects:

- The way data formats are represented in the DSpace content model.
- Clarification and rationalization of the use of `BitstreamFormat`.
- Mechanisms for identifying the data format of a `Bitstream`.
- Integration of standards-based technical metadata about data formats which can be effectively shared with other applications.

The original DSpace design intentionally avoided the issue of describing data formats in such detail because there were already other efforts underway to thoroughly catalog data formats – and DSpace would eventually leverage their work. As of June, 2007, the most sophisticated data format registries are *still* in development, but some usable systems are operating in production. We propose to integrate external data format intelligence through a flexible plugin-based architecture to take advantage of what is currently available but leave a clear path for future upgrades and changes. It also lets each DSpace installation choose an appropriate level of complexity and detail in their format support.

## Purpose and Principles

- Enable accurate, meaningful, fine-grained, and globally-understood identification of a `Bitstream`'s data format.
- Maintain backward compatibility with most existing code, and existing archives.

- Introduce the binding of persistent, externally-assigned data format identifiers to `BitstreamFormats`.
- Integrate tightly with "standard" data format registries, using a plugin framework for flexible configuration:
    - Anticipate that the Global Digital Format Registry (GDFR) will be the registry of choice, but allow free choice of other metadata sources.
    - Recognize references to entries in "standard" data format registries in ingested content (e.g. technical MD in SIPs) to facilitate exchange of SIPs and DIPs.
    - The DSpace data model directly includes *only* the subset of format metadata it has an immediate use for, and references entries in an external format registry for the rest.
    - Refer to formats by the external data format registry's identifiers so format technical metadata is recognized outside of DSpace.
- Improve the automatic identification of data formats in batch and non-interactive content ingestion.
- Help interactive users identify formats easily and with accuracy during interactive submission.
- Rationalize use of `BitstreamFormat` object:
    - Eliminate the overloaded use of the "License" format and "Internal" flag in BitstreamFormats to mark and hide deposit license bitstreams.
    - Attempt to accurately describe the data format of *every* Bitstream, even the ones created for internal use.
- Create pluggable interface to external data format registries, to encourage experimentation and track developments in this highly active field.
- Add a separate pluggable format-identification interface to allow a "stack" of methods to identify the format of a Bitstream by various techniques.

## Use Cases

See BitstreamFormat Renovation for the sketches of the anticipated use cases that drove this design. The text grew too large for one page.

## Overview of Changes to DSpace Core and API

Here is a summary of all of the proposed changes, by section.

### Content Model

#### `Bitstream`

Each `Bitstream` still refers to a `BitstreamFormat` object to identify its data format. In addition, the `Bitstream` gains two new properties:

1. A *format confidence* metric, which indicates (on a coarse symbolic scale) the certainty of the identification of its format, reflecting both accuracy and precision.
2. The *source* of the format identification, indicating the tool or mechanism responsible for the format technical metadata in this Bitstream.

#### `BitstreamFormat`

Although outwardly similar and largely backward-compatible, the `BitstreamFormat` has been completely gutted and re-implemented. It now serves as a local "cache" of format technical metadata and holds one or more *external format identifiers*, each of which refers to a complete technical metadata record in an external *data format registry*.

These external registries (as described below) are the authoritative source of format technical and descriptive metadata about data formats. This fundamental change lets DSpace take advantage of the extensive work and recognized standards offered by external format registries such as PRONOM and the GDFR.

### Core API

#### External Format Registry

We add a plugin interface to provide access to external data format registries. Each registry is modeled as an implementation of the `FormatRegistry` interface. It is fairly simple; it only supports "importing" a format description into the local `BitstreamFormat` cache, updating an existing format, and a few queries.

A single DSpace archive may be configured to access many external format registries. It usually will be, since no one registry currently has all the answers.

Backward-compatibility is provided by a built-in "DSpace" registry which contains all of the DSpace-1.4 formats.

#### Automatic Format Identification

The old `org.dspace.content.FormatIdentifier` is replaced by a configurable, extensible, plugin-based format identification framework. It is *not* part of the format registry plugin, because while some format recognition services live in a registry's software suite, others are independent of any registry.

Format identification is one of the most important improvements in this project. It is explained in complete detail below.

#### Administration

There are additions to the configuration and administration APIs to support configuration and maintenance of the new registry and format-identification frameworks.

# Changes to Content Model

Here are detailed explanations of the planned changes to content model classes.

## Exceptions

The following exceptions are thrown when a fatal error is encountered in the format registry and identification framework. They are similar in meaning to existing exceptions in the DSpace API, such as `AuthorizeException` – signalling a fatal error with enough context and explanation to communicate the cause to the user or administrator.

### `FormatRegistryException`

Sent when there is a fatal error while accessing an external format registry or updating the local cache of format metadata in the DSpace RDBMS. Can be caused by incorrect or missing configuration entries, network problems, filesystem problems, etc.

#### `FormatRegistryNotFoundException`

A subclass of `FormatRegistryException`, this exception is sent in particular cases when looking up an external identifier fails although it *should* have been found (e.g. since it had been found before). In the common case of looking up an identifier for the first time, e.g. through `BitstreamFormat.findByIdentifier()`, no exception gets thrown because failure is a possibly-expected result.

## `FormatIdentifierException`

Thrown when a format identification method encounters a fatal error which would cause it to return a false negative result. For example, if its configuration is missing or incorrect, the method throws this exception rather than silently failing. Simply failing to identify a format is in the realm of expected results and does not cause an exception.

## `FormatIdentifierTemporaryException`

Thrown by the format identification method when it fails because of a "temporary" problem, e.g. when a network resource is not available. This subclass of `FormatIdentifierException` tells the identifier manager that it may succeed when retried later.

## `Bitstream`

The most significant change is that the `Bitstream` now remembers the *confidence* of its format identification, an enumerated value which indicates the certainty and source of its format identification. There is also a convenience method to access the automatic format identification: since it almost always used to set the `Bitstream`'s format anyway, this improves code clarity.

Here is an interface view of the API additions:

### `FormatConfidence`

```
// Ordered symbolic values of format-identification confidence:
// (See Automatic Format Identification section for details)
package org.dspace.content;
public enum FormatConfidence
{
// No format was identified. The unknown format is assigned.
UNIDENTIFIED,
//
// A format was found but it was based on "circumstantial evidence", i.e. external properties of the Bitstream such as its name or a MIME type attached to it on ingest.
CIRCUMSTANTIAL,
//
// The data format was determined by coarse examination of the contents of the Bitstream and comparison againt the known characteristics of generic formats such as plain text, or comma-separated-values files.
HEURISTIC,
//
// The format was identified by matching its content positively against an internal signature that describes a "generic" (supertype) format or family of formats.
POSITIVE-GENERIC,
//
// The format was identified by matching its content positively against an internal signature that describes a specific, precise, data format.
POSITIVE-SPECIFIC,
//
// Contents of Bitstream are validated as conforming to the identified
// format, this is the highest confidence reached by automatic identification.
VALIDATED,
//
// Format is derived from reliable technical metadata supplied at the time
// the Bitstream was ingested; if applied it is given a priority
// that overrides automatic identifications. Ingest-derived
// formats with a low level of confidence should be assigned CIRCUMSTANTIAL.
INGEST,
//
// Format was identified interactively by a user, which acts as an override of automatic format identification.
MANUAL
}
```

// ONLY includes new methods

```
public class Bitstream extends DSpaceObject
{
// Returns the "confidence" recorded when the format of this Bitstream was identified.
public FormatConfidence getFormatConfidence()
//
// Sets the value returned by getFormatConfidence()
public void setFormatConfidence(FormatConfidence value)
//
// Returns the source that identified the format of this Bitstream.
public String getFormatSource()
//
// Sets the value returned by getFormatSource()
public void setFormatSource(String value)
}
```

## `BitstreamFormat`

The `BitstreamFormat` class, which we will abbreviate as **BSF**, is essentially gutted and replaced with a new implementation. As described above, it now serves as a local "cache" of technical metadata that comes from external *data format registries*. Every BSF is bound to *at least one* format identifier in an external registry so its format technical metadata can be expressed in a way that is recognized outside of DSpace.

### Rationalization of Usage

In the current (version 1.4.x) codebase, the `BitstreamFormat` object has acquired uses and meanings beyond simply describing a Bitstream's data format – but these interfere with intended purpose in preservation activities. For example, the BSF has an **internal** flag which directs the UI to hide Bitstreams of that format from casual view. In an unmodified DSpace installation, the `"License"` BSF is the only one for which **internal** is true, to keep deposit license files from appearing in the Web UI. Unfortunately, this usage cripples `"License"` as an actual *format* descriptor, since it gets applied to all sorts of Bitstreams that contain licensing information no matter what their actual format. XML, plain-text, and RDF files are all tagged with the "License" BSF to make them disappear, yet it says nothing about the format of their contents.

We rationalize the function of the BSF so that it *only* describes the data format of a Bitstream's contents. There are other ways to mark Bitstreams as "internal", e.g. Bundle membership, which is a better fit with the semantics of the content model anyway.

### Formal Definition

This is the new formal definition of a `BitstreamFormat`:

- Each BSF represents a description of a single, unique data format; there is exactly one BSF for each distinct data format referenced by Bitstreams in the DSpace archive.
- A BSF is *bound* to one or more *entries in external data format registries.*** The identifiers are logically all peers, although the metadata cached in the BSF is only imported (or updated) from one of them.
    - All external format identifiers which describe the *equivalent format* must be bound to the same DSpace BSF – in other words, there should never be two BSFs describing the same conceptual format, such as "PDF Version 1.2"; one BSF encompasses all synonym external identifiers.
- The BSF's function is to describe the data format of the contents of a Bitstream, and *nothing more*.
    - Application code must not "overload" a BSF with additional implicit meanings, such as marking Bitstreams invisible in a UI or indicating a function such as the deposit license.
- One special BSF, the *unknown* format, represents the unknown or unidentified data format.
- Every Bitstream refers to exactly one BSF:
    - If its format has not been assigned or identified, it is the *unknown* format.
    - This allows an application to assume every Bitstream has a valid BSF with all of its attendant properties, so e.g. it can get a valid MIME type.

In the content model, a `BitstreamFormat` aggregates all the format-related technical metadata for Bitstreams of its type. Not only does this save space, it lets an administrator make changes and adjustments to that metadata easily in one place.

It also holds DSpace-specific format metadata, which currently consists only of the one administrative metadata item, the "support-level", which controls the preservation support policy for all Bitstreams of that format.

This new implementation makes the `BitstreamFormat` a local cache for the relevant format metadata, but mainly it acts as a reference to the full technical metadata found in one or more external *format registries* (like the GDFR). It only caches the metadata immediately needed by DSpace, such as MIME type, name, description. This is adequate for everyday operation of the archive; DSpace never has to go to the external format registry for metadata.

This organization makes it much easier to take advantage of developments in the rapidly evolving field of data format technical metadata. Instead of trying to import all of the various schemas and data models of of every external data format registry, we can just maintain a reference into the external registry.

## External Format Identifiers

External data format *identifiers* are introduced in this architecture. They link DSpace's internal BSF objects to entries in external format registries such as the GDFR. Format identifiers consist of a *namespace* name, and the identifier, an arbitrary string. They possess these properties:

*__Globally unique:__ each pair of namespace and identifier (except for the "Provisional" namespace) is unique in the world.
*__Persistent:__ Identifiers in external format registries are expected to be persistent, that is, bound forever to the same format description.
*__Resolvable:__ There must exist some automated means to retrieve the metadata bound to an external identifier.

- 
    - It may require special software and local data files or contacting network resources.
      *__Opaque:__ The content of the identifier itself is assumed meaningless and is not interpreted.
      *__Cardinality:__ A single external identifier may only belong to one BSF, but a BSF may be bound to multiple format identifiers.
      *__Required.__ A BSF *must* be bound to at least one external identifier.

External identifiers are the key to naming data formats in a way that can be recognized outside of the DSpace system; they allow the BSF to be meaningfully crosswalked to and from technical metadata schemas such as PREMIS.

### The Namespace

We add a DSpace-specific *namespace* to the external format identifier to:

1. Mark which external registry it belongs to
2. Ensure every external identifier is unique even if two registries have identical identifiers

The *namespace* is a short string belonging to a controlled vocabulary (represented by a table in the DSpace configuration or the RDBMS). Each namespace describes a source of data format information.

### Initial Set of Well-Known Namespaces

Here are the suggested initial namespace names for existing registries. This is *not* an exhaustive set; since the registry is a plugin, a new registry can always be added as a DSpace add-on.

The **DSpace-Internal**, **DSpace**, and **Provisional** registries are implemented in the core. We exepect a **PRONOM** registry plugin to be available with the initial reference implementation.

| Namespace | Description |
|---|---|
| **DSpace-Internal** | Contains only the **Unknown** identifier, a placeholder for the *unknown format* which represents an unidentified Bitstream format. This is the only *mandatory* namespace which is automatically configured. |
| **DSpace** | Contains most of the original generic formats defined by DSpace, for backward-compatibility and for archives which do not care about precise data format descriptions |

| | |
|---|---|
| **Provisional** | For custom data formats local to the archive. Provisional extensions to the "DSpace" format registry are put in their own namespace so there is no chance of a conflict with formats added later to "DSpace", and also to make their status as local extensions obvious. |
| **PRONOM** | PUIDs from PRONOM's format registry |
| **GDFR** | Persistent identifiers from the Global Distributed Format Registry. |
| **LOC** | Library of Congress Sustainability of Digital Formats project format descriptions. |

The standard namespace values are available as public static fields on the `FormatRegistryManager` class. The LOC namespace is not really a registry yet but it makes sense to reserve the namespace since it is a significant source of format technical metadata.

### MIME Types excluded

Note that MIME types *cannot* be BSF identifiers because they violate the rule that only one BSF may be bound to each identifier. MIME types are imprecise, many BSFs have the same MIME type; e.g. a lot of XML-based are tagged `"text/xml"`.

### Apple UTIs

Apple Computer has developed what is essentially an alternative to MIME types called Uniform Type Identifiers (UTIs). It is an interesting development, although not directly relevant. Although the UTI database is, in a sense, a registry of format identifiers, it is not a good candidate for use in DSpace for several reasons:

- No accompanying metadata of any sort, it is just an identifier space.
- Ad-hoc extensions by vendors, no online central registry.
- Varying levels of granularity, since it is primarily intended to match files to application programs.

## Removing the "Internal" Flag

This renovation removes `BitstreamFormat`'s "internal" flag, which was originally intended to guide UI applications in hiding certain classifications of Bitstreams.

Was the "Internal" flag ever truly necessary? Apparently it was only ever used to make the Bitstreams containing deposit-license and Creative Commons license invisible in the Web UI.

Its presence is actually harmful: not only does it have nothing to do with describing the format of the data, it actually encouraged usage that obscures the data format. The one `"License"` BSF was applied to all Bitstreams containing an Item's deposit license and Creative Commons licenses, no matter what their actual data formats. The Creative Commons license consists of three Bitstreams of distinct actual formats – e.g. one is RDF. It is misnamed with the `"License"` format so it will not be properly preserved.

In the DSpace@MIT registry, I have determined that `"License"` is the only `BitstreamFormat` for which "internal" is true, and that Bitstreams whose format is "License" appear *only* in Bundles named "LICENSE" and "CC_LICENSE". Therefore, we can determine the internal-ness (i.e. invisibility) of a Bitstream by its its owning Bundle as accurately as by the bogus BSF. It makes **no** *practical* difference which technique is used, but the Bundle-name cue is a better fit with the current content model. It works just as well under the DSpace+2.0 content model, since bundles evolve into "manifestations".

By modifying UI code to judge "visibility" of a Bitstream by the Bundle it belongs to rather than a cryptic property of its format, we can get rid of the "internal" bit without any user-visible changes. The content model and API are improved, since license Bitstreams may now have meaningful data formats assigned so they can be preserved and disseminated correctly.

## BitstreamFormat Properties

Here is a list of all the BSF's *properties*, i.e. fields of the object. **Source** is where the property originally comes from; **Mod** means whether or not it may be changed by the archive administrator.

**BitstreamFormat Properties**

| Property | Source | Mod | Description |
|---|---|---|---|
| Name | Registry, Can override | Yes | Brief, human-readable description of this format, for listing it in menus.<br>Used to be "short description". |
| Description | Registry, Can override | Yes | Detailed human-readable explanation of the format including its unique aspects. |
| Identifier | Registry | No | List of all namespaced identifiers linking this BSF to an entry in an<br>established data format registry. A BSF *must* have at least one identifier.<br>This list is *ordered*; the first member names the external registry entry<br>that was originally imported to create this BSF. |
| Support-level | User-entered | Yes | Encoding of the local DSpace archive's policy regarding preservation of Bitstreams encoded in this format. Value must be one of:<br>1. Unset - Policy not yet initialized, flags format entries that need attention from the DSpace administrator.<br>2. Unrecognized - Format cannot be identified.<br>3. Known - Format was identified but preservation services are not promised.<br>4. Supported - Bitstream will be preserved. |
| MIME-type | Registry, Can override | Yes | Canonical MIME type (Internet data type) that describes this format. This is where the `Content-Type` header's value comes from, when delivering a Bitstream by HTTP. |
| Extension | Registry, Can override | Yes | The canonical filename extension to apply to unnamed Bitstreams when<br>delivering content over HTTP and in DIPs. (NOTE: Some format<br>registries have a *list* of filename extensions is, used to help<br>identify formats, but we only need the canonical extension in the BSF model. |
| LastUpdated | System | No | Timestamp when this BSF was imported or last updated from its home registry. |

## `BitstreamFormat` API

Here is the new API for `BitstreamFormat`.

### ***Add*** the following methods:

```
// Returns all external identifiers bound to this BSF
public String[] getIdentifiers()
throws SQLException, AuthorizeException;
//
// Add a binding to an external identifier
// Versions to accept separate namespace and identifier, or namespaced identifier.
public void addIdentifier(String nsIdentifier)
throws SQLException, AuthorizeException;
public void addIdentifier(String namespace, String identifier)
throws SQLException, AuthorizeException;
//
// remove a binding to an external identifier.
public void deleteIdentifier(String nsIdentifier)
public void deleteIdentifier(String namespace, String identifier)
throws SQLException, AuthorizeException
//
// Find BSF bound to an external identifier, returns null if none found.
// Versions to accept separate namespace and identifier, or namespaced identifier.
public BitstreamFormat findByIdentifier(Context context, String nsIdentifier)
throws SQLException, AuthorizeException, FormatRegistryException
public BitstreamFormat findByIdentifier(Context context, String namespace, String identifier)
throws SQLException, AuthorizeException, FormatRegistryException
//
// Advanced version with extra "import" param, says to NOT look
// for format in registries, but to return 'null' if there is no
// existing BSF matching the indicated format. (Mainly for internal use.)
public BitstreamFormat findByIdentifier(Context context, String namespace, String identifier, boolean import)
throws SQLException, AuthorizeException, FormatRegistryException
//
// Return true if this BSF conforms to the target identifier, i.e. if
// would be acceptable to a service that accepts the given format.
public boolean conformsTo(String nsIdentifier);
throws SQLException, AuthorizeException, FormatRegistryException
//
// Return/set the canonical filename extension (without ".").
public String getCanonicalExtension();
public setCanonicalExtension(String extension);
//
// Get and set the Name (takes place of ShortDescription)
public String getName();
public void setName(String s);
//
// Flags that show whether to override the values picked up from external registry
// Set to false to remove override.
public boolean isOverrideName();
public void setOverrideName(boolean val);
public boolean isOverrideDescription();
public void setOverrideDescription(boolean val);
public boolean isOverrideMIMEType();
public void setOverrideMIMEType(boolean val);
public boolean isOverrideCanonicalExtension();
public void setOverrideCanonicalExtension(boolean val);
//
// return true if this BSF is the unknown format.
public boolean isUnknown()
//
// return date this external-id was last imported to the BSF.
public Date getLastImported(String namespace, String id)
public Date getLastImported(String nsidentifier)
throws SQLException, AuthorizeException
//
// set the date this external-id was last imported to the BSF.
public void setLastImported(String namespace, String id, Date newdate)
public void setLastImported(String nsidentifier, Date newdate)
throws SQLException, AuthorizeException
```

### Remove these methods:

```
findByShortDescription()
findByMIMEType()
findNonInternal()
//
isInternal()
setInternal()
//
getShortDescription() // renamed to getName()
setShortDescription() // renamed to setName()
//
getExtensions()
setExtensions()
```

### Existing Methods that Remain:

These existing methods are retained, just mentioned here for completeness.

```
static BitstreamFormat create(Context context);
void delete();
static BitstreamFormat find(Context context, int id);
static BitstreamFormat findUnknown(Context context);
static BitstreamFormat[] findAll(Context context);
String getDescription();
int getID()
String getMIMEType();
int getSupportLevel();
static int getSupportLevelID(String slevel);
void setDescription(String s);
void setMIMEType(String s);
void setSupportLevel(int sl);
void update();
```

## `FormatRegistryManager`

Following DSpace coding conventions, the factory and static class for a service is named with the suffix `-Manager`. The `FormatRegistryManager` class gives access to instances of `FormatRegistry`. Since a format identifier is directed to a `FormatRegistry` implementation by its *namespace*, the Manager also takes care of selecting the right instance for a namespaced identifier. This lets applications use namespaced identifiers without worrying about taking them apart to choose a registry instance.

Since all of the `FormatRegistryManager`'s state is effectively managed by the Plugin Manager, it does not need any state itself and only has static methods.

## API

Here is a sketch of the API:

```
public class FormatRegistryManager
{
// Namespaces for internal format registry - contains only "Unknown"
public static final String INTERNAL_NAMESPACE = "Internal";
//
// Name of the unknown format:
public static final String UNKNOWN_FORMAT_IDENTIFIER = "Unknown";
//
// Applications should use this as default mime-type.
public static final String DEFAULT_MIME_TYPE = "application/octet-stream";
//
// returns possibly-localized human-readable name of Unknown format.
public static String getUnknownFormatName(Context context);
//
// Returns registry plugin for external format identifier namespace
public static FormatRegistry find(String namespace);
//
// Returns array of all Namespace strings, even "artifacts" no longer configured.
public static String[] getAllNamespaces(Context context)
throws SQLException, AuthorizeException
//
// Returns array of all currently Namespaces of external registries.
public static String[] getRegistryNamespaces();
//
// Calls apropriate registry plugin to import format bound to a namespaced identifier.
// Returns null on error.
public static BitstreamFormat importExternalFormat(Context context, String namespace, String identifier)
throws FormatRegistryException, AuthorizeException
//
// Calls apropriate registry plugin to update format bound to a namespaced identifier.
// When force is true, update even when external format has not been modified.
public static void updateBitstreamFormat(Context context, BitstreamFormat existing, String namespace, String identifier, boolean force)
throws FormatRegistryException, AuthorizeException
//
// Calls apropriate registry plugin to compare two namespaced
// identifies (which must be in the same namespace).
public static boolean conformsTo(String nsIdent1, String nsIdent2)
throws FormatRegistryException
//
// Creates a namespaced identifier out of separate namespace and registry-specific identifier.
public static String makeIdentifier(String namespace, String identifier);
//
// Returns the namespace or identifier portion of a namespaced identifier.
public static String namespaceOf(String nsIdentifier)
public static String identifierOf(String nsIdentifier)
}
```

## FormatRegistry

The `FormatRegistry` interface models an external data format registry. We define *data format registry* as any formally organized and administered collection of technical metadata about data formats. This *may* include a collection published mainly for human consumption such as the Library of Congress Sustainability of Digital Formats format catalog, as well as those accessible through public APIs such as the GDFR and DROID. The only requirement is that the data formats are named by unchanging, unique *identifiers*.

A format registry becomes available to DSpace through a Named Plugin implementing the `FormatRegistry` interface.

A registry has these functions, explained in detail below:

- Resolve a reference to a format *identifier*, and create a new `BitstreamFormat` representing the external format.
- Update the local cache of metadata in a `BitstreamFormat` to the current state of the external registry.
- Answer "conformance" queries - judge whether a Bitstream in one format would also conform to another format (i.e. former is a subtype of latter).

### Format Registries are Tightly Coupled

Format registry implementations are tightly coupled with DSpace. By this we mean they must be able to respond to frequent queries quickly and with low latency, and high reliability. The format registry must be available to complete some common operations such as ingestion and selection of applications like `MediaFilter`.

This is only likely to be an issue at all with registries that are attached through the network. Registries that exist in local data files or RDBMS tables can share server resources with the DSpace archive.

Network-based registries might use a local "cache" server sharing the DSpace host to increase reliability. The GDFR architecture explictly encourages this sort of configuration. Otherwise, it might be necessary for the `FormatRegistry` implementation to add caching of its own to increase performance.

## Format Identification is Separate

Although some of the tools to automatically identify formats are tied to format registries, this registry interface does *not* have anything to do with format identification. The identification tools are accessed through a separate plugin interface, discussed below.

## `FormatRegistry` API

Here is the API of the `FormatRegistry`. The plugin's name is also the DSpace string value representing its namespace. It is implemented as a self-named plugin, so that the instance itself knows its namespace without depending on each DSpace administrator to get it right. The namespaces must be consistent between DSpace installations so that format technical metadata (i.e. PREMIS elements in AIPs) can be meaningfully exchanged.

```
// implementing classes should extend SelfNamedPlugin
package org.dspace.content.format;
public interface FormatRegistry
{
// Typically returns 1 element, the Namespace name of the implementation's registry
String [] getPluginNames();
//
// Returns the DSpace namespace of this registry.
public String getNamespace();
//
// Return an URL needed to configure the underlying registry service;
// this allows the registry to configure itself from the DSpace
// configuration.
public URL getContactURL();
//
// Returns all external identifiers known to be synoyms of the
// given one, in namespaced-identifier format. (Because one registry
// may know about synonyms in other registries.)
public String[] getSynonymIdentifiers(Context context, String identifier)
throws FormatRegistryException, AuthorizeException
//
// Import a new data format - returns a BitstreamFormat. There
// not be any existing BSF with the same namespace and identifier.
public BitstreamFormat importExternalFormat(Context context, String identifier)
throws FormatRegistryException, AuthorizeException
//
// Compare existing DSpace format against registry, updating anything that's changed.
// NOTE: it does not need to check last-modified date, framework does that.
public BitstreamFormat updateBitstreamFormat(Context context, BitstreamFormat existing, String identifier)
throws FormatRegistryException, AuthorizeException
//
// Return date when this entry was last changed, or null if unknown.
public Date getLastModified(String identifier)
throws FormatRegistryException
//
// Predicate, true if format named by sub is a subtype or
// otherwise "conforms" to the format defined by fmt.
public boolean conformsTo(String sub, String fmt)
throws FormatRegistryException
//
// Free any resources associated with this registry connection,
// since it will not be used any more.
public void shutdown()
}
```

### Registry Name

Typically the name of the registry is bound to some well-known public constant so it can be referred to in a program without a "magic string" that is easily misspelled to disastrous effect. E.g.:

```
FormatRegistryManager.INTERNAL_NAMESPACE ... "Internal"
DSpaceFormatRegistry.NAMESPACE .... "DSpace"
ProvisionalFormatRegistry.NAMESPACE .... "Provisional"
```

Implementors of new registries should follow this convention.

### The Import Operation

*Importing* an entry from an external format registry creates a new local BSF. In order to create a new BSF, there must not be an existing BSF already bound to that entry's identifier (or any identifiers it lists as synonyms).

So, first check for a BSF bound to *any* of the namespaced identifiers attached to the external registry entry. If one is found, add the new identifier to that BSF and return it instead of creating a new one.

If no existing BSF is found, create a new one, and initialize its properties from appropriate values of registry entry's technical metadata:

- The entry's identifier becomes the BSF's first external identifier.
- Any synonym identifiers in the entry are added to the BSF
- Initialize the BSF's name, description, MIME type etc.

Binding *all* equivalent (synonym) identifiers in the remote entry ensures that a scenario like this does not create extra local BSFs:

1. Import format "PRONOM:foo" as a BSF.
2. When asked to import format "GDFR:bar", discover that the GDFR entry lists "PRONOM:foo" as one of its synonyms.
    - Creating a new entry for *that* would result in two entries bound to "PRONOM:foo", which is illegal.
    - Adding "GDFR:bar" as a synonym identifier to the existing BSF solves the problem.

### The Update Operation

This describes the way a single BSF is updated from its corresponding entry in an external data format registry – the choice of which BSF to update and which registry to query are separate issues covered in the section on administrative actions.

Updating the BSF is a lot like the import operation:

- For metadata items such as Name, Description, etc. replace the value in the BSF if the remote registry's value is different.
- If the remote entry has any synonym identifiers which are not already listed, add them.
- Do not delete synonym identifiers that *are not* listed since they may have come from another registry.

### The ConformsTo Operation

Supplied with the identifiers of two entries in the registry, this predicate function returns true if the the first format *conforms to* the second. That means, any Bitstream identified as the first format would pass the tests to be identified as the second as well. For example, if the first format is a specific version of a format while the second identifier names a format family which includes it, *conformsTo* would be true.

The registry plugin should implement this operation efficiently, since it may be called many times, e.g. when choosing applications to match the format of a Bitstream.

## The DSpace-Internal "Registry"

The *unknown* BSF is installed with the system, but for consistency, it is also derived from an entry in an "external" format registry. Since it is the only BSF which is absolutely mandatory, this registry must always be available, so it is a hard-coded registry that is always configured.

The `FormatRegistryManager` maps the namespace `DSpace-Internal` to a special registry object which only recognizes the "Unknown" format identifier. The first reference to that format identifier, e.g. by the method `BitstreamFormat.findUnknown()`, "imports" it to create the *unknown format* BSF.

Although it might appear that the "Unknown" format really belongs in the **DSpace** namespace and registry, that would force the **DSpace** registry to be configured all the time. Putting "Unknown" in a separate built-in registry lets the administrator remove the **DSpace** registry from the configuration if she wishes to.

## The Built-In Format Registries

The initial implementation also includes *built-in* format registries for the **DSpace** and **Provisional** registry namespaces. Unlike the **DSpace-Internal** registry, they are *optional*. By itself, the **DSpace** registry reproduces the release 1.4.x behavior to offer the option of backward-compatibility. The **Provision al** registry offers a separate place to put formats local to the archive, safe from namespace collisions and future updates to the **DSpace** registry. (It is not always the *recommended* way to handle new formats, more on this later.)

These two registries share an implementation class since their operation is exactly the same. The plugin manager creates one instance for each registry (i. e. namespace). They use the plugin/namespace name to select a configuration file. The registry's contents come from that file, which is read on startup.

To add entries to the **Provisional** format registry, the DSpace administrator edits its configuration file (in a documented XML format similar to the current `b itstream-formats.xml` initialization file) and restarts any relevant DSpace processes. Since changes should be *very* infrequent this should not be a burden.

### "DSpace" (Traditional) Format Registry

The "DSpace" registry includes most of the traditional, loosely-defined, format names, like `"Text", "Adobe PDF", "HTML"`. It offers a simple solution for DSpace administrators who do not need precise and detailed format identification, nor the digital preservation tools that require it. Since it includes most of the formats from previous DSpace releases under their same names, it also gives a degree of backward-compatibility.

It is *not* necessary to include this registry in the configuration. It can be left out if, e.g., the administrator only wants to use PRONOM formats.

The contents of the "DSpace" registry are controlled by DSpace source code releases and *must not* be altered locally. See the next section to add formats to your archive.

### "Provisional" Format Registry

The **Provisional** format registry lets a DSpace administrator add data formats which are *not* available in any other external registry to her DSpace archive. The contents of the "Provisional" registry are strictly under the control of the administrator. It starts out empty.

Using formats from the **Provisional** namespace carries some risks: the format identifiers are meaningless (and useless for preservation) outside of their own DSpace archive. Even another DSpace might not have the same **Provisional** formats configured. Of course, a Provisional format should only be added when it is not available in any shared registry, anyway.

As soon as a new format *does* become available in some external registry, you can add the new external identifier to its `BitstreamFormat`, perhaps updating the BSF's local metadata from its external registry.

Ideally, you will *only* employ Provisional formats when there will eventually be an entry in a globally-recognized registry for the format. For example, if you are adding a format to the GDFR but need to apply it to a Bitstream immediately, before the GDFR editorial process accepts it, you could create it in the **Pr ovisional** registry to have it available immediately. Later, once the GDFR has an entry for it, add the GDFR identifier to the `BitstreamFormat` you already created. Then, DIPs of objects in that format will bear the GDFR format identifier that is recognizable to other archives, and your Bitstreams will also have linkage to any preservation metadata in the GDFR.

This registry is implemented the same way as the "DSpace" registry, reading format information from an XML document that lists all the "Provisional" format, at startup. However, its identifiers occupy a separate namespace so there is no chance of collisions with the data formats provided by the DSpace release.

## Database Schema Changes

Here are the tables which were added or changed:

```
------------------------------------------------------
– ExternalFormatNamespace table, for efficient storage and comparison of External Format Identifier namespaces
------------------------------------------------------
CREATE TABLE ExternalFormatNamespace
(
namespace_id INTEGER PRIMARY KEY,
namespace VARCHAR(128)
);
------------------------------------------------------
– BitstreamFormat table, a cache of references to technical metadata in external format registries
------------------------------------------------------
CREATE TABLE BitstreamFormat
(
bitstream_format_id INTEGER PRIMARY KEY,
name VARCHAR(128),
description TEXT,
mimetype VARCHAR(128),
support_level INTEGER,
canonical_extension VARCHAR(40),
override_name BOOL DEFAULT FALSE,
override_description BOOL DEFAULT FALSE,
override_mimetype BOOL DEFAULT FALSE,
override_canonical_extension BOOL DEFAULT FALSE
);
------------------------------------------------------
– ExternalFormatIdentifier table, map of BitstreamFormat entry to namespaced external identifier
------------------------------------------------------
CREATE TABLE ExternalFormatIdentifier
(
external_format_id INTEGER PRIMARY KEY,
bitstream_format_id
INTEGER REFERENCES BitstreamFormat(bitstream_format_id),
namespace_id INTEGER REFERENCES ExternalFormatNamespace(namespace_id),
external_identifier VARCHAR(128),
last_imported TIMESTAMP WITH TIME ZONE
);
ALTER TABLE ExternalFormatIdentifier ADD CONSTRAINT identifier_uniqueness
UNIQUE (namespace_id, external_identifier);
------------------------------------------------------
– FormatSource table, normalized values of format_source column since we can expect much repetition.
------------------------------------------------------
CREATE TABLE FormatSource
(
format_source_id INTEGER PRIMARY KEY,
format_source VARCHAR(256) UNIQUE
);
-------------------------------------------------------
– Bitstream table
-------------------------------------------------------
CREATE TABLE Bitstream
(
bitstream_id INTEGER PRIMARY KEY,
bitstream_format_id INTEGER REFERENCES BitstreamFormat(bitstream_format_id),
name VARCHAR(256),
size_bytes BIGINT,
checksum VARCHAR(64),
checksum_algorithm VARCHAR(32),
description TEXT,
user_format_description TEXT,
source VARCHAR(256),
internal_id VARCHAR(256),
deleted BOOL,
store_number INTEGER,
sequence_id INTEGER,
format_confidence INTEGER DEFAULT 0,
format_source_id INTEGER REFERENCES FormatSource(format_source_id)
);
```

## Automatic Format Identification

Experience has shown that even the most knowledgeable submitters rarely understand or care about identifying the data formats of materials they upload. Also, many submissions are done in batch and non-interactive transactions where human intervention is not possible. Thus, we promote automatic format identification as the primary method of assigning formats to Bitstreams, and strive to make it accurate, reliable and efficient.

We propose a configurable and extensible framework for integrating external automatic format identification services. This is the best approach because:

- There are already several powerful, open-source, format identifiers available for integration.
- Format identification services are distinct from data format registries (though some are related), so they demand a separate plugin framework.
- Since the science and practice of format identification is still evolving rapidly, DSpace must be flexible to take advantage of new developments.
- Many DSpace sites have unique needs dictated by the types of digital objects that get submitted, perhaps demanding completely custom format identification techniques.

## Format Identification Framework

The framework is a common API to which format identification services conform. This lets DSpace treat them as a "stack" of plugin implementations, trying each one in turn and choosing the best of all their results. The API consists of:

- `FormatHit`
  A class encapsulating the information returned by a single potential format identification "hit"
- `FormatConfidence`
  A set of enumerated values that quantifies the "confidence" (certainty, accuracy) of format identifications for comparison on a common basis.
- `FormatIdentifier`
  Interface of the plugin class that actually identifies formats.
- `FormatIdentifierManager`
  Static class to operate the plugin stack and return a format identification verdict.

### The `FormatHit` Object

A `FormatHit` is a record of the results of one format-identification match. It contains the following fields:

```
package org.dspace.content.format;
public class FormatHit implements Comparable<FormatHit>
{
// Namespace and identifier of the external format
// (which may not yet exist as a BitstreamFormat)
public String namespace = null;
public String identifier = null;
//
// Human-readable name of the format, if any (may be null), to aid user in selection
public String name = null;
//
// Measure of the "confidence" of this hit, one of the constants
// described below.
public FormatConfidence confidence;
//
// Record of the method which identified the format.
public String source;
//
// Any warning from the identification process (to show to user)
public String warning = null;
//
// Constructor
public FormatHit(String namespace, String identifier, String name, FormatConfidence confidence, String warning, String source);
//
// Utility to add this hit in the correct place on a results list,
// implementing the DEFAULT method (not the ONLY method)
public void addToResults(List<FormatHit> results);
//
// Convenience method to return a single string containing the namespaced format identifier:
public String getIdentifier();
//
// Convenience method returning resulting BSF
public BitstreamFormat getBitstreamFormat(Context context)
throws SQLException, AuthorizeException, FormatRegistryException
//
// for Comparable<FormatHit>
public int compareTo(FormatHit o);
//
// Set the relevant fields in Bitstream with contents of this hit.
public void applyToBitstream(Context c, Bitstream bs)
throws SQLException, AuthorizeException, FormatRegistryException
}
```

Each attempt at automatic identification of a Bitstream's format returns a Collection of `FormatHit` objects, representing the possible matches. The list is sorted by accuracy and confidence of hit.

### Confidence of Format Identification Hits

The `FormatHit` includes a *confidence* metric, which represents the accuracy and certainty of the identification. It is an enumerated type of ordered, symbolic values implemented as a Java 5 enumeration.

The specific values are described above, under the description of the `FormatConfidence` object.

`FormatHit` includes a confidence rating so hits can be compared on the basis of confidence, and so it can be stored in the `Bitstream` object whose format was identified.

The confidence values have a greater range and granularity than seems possible given DSpace's simple format model; i.e. DSpace does not distinguish betwen "generic" and "specific" formats. However, the actual automatic format identification is done by plugin implementations, some of which are driven by external format registries. These have access to more sophisticated format models and data, including notions of format granularity, so the confidence metrics reflect that.

## `FormatIdentifier` Interface

Automatic format identification is accomplished by plugins implementing the `FormatIdentifier` interface. Each plugin applies its own technique toward identifying the format of the Bitstream. There is no direct relationship between external data format registries and format identifying plugins: a single plugin can utilize several registries or none, and different plugins can use the same external registry.

Note that the `FormatHit` returned by the identification process contains an external format identifier, *not* a `BitstreamFormat`. The archive administrator is responsible for ensuring that all external format identifiers returned by automatic identification methods can be imported, i.e. that the relevant registries are configured.

Here is the Java interface:

```
package org.dspace.content.format;
public interface FormatIdentifier
{
// identify format of "target", add hits to "results"
public List<FormatHit> identifyFormat(Context context, Bitstream target, List<FormatHit> results)
throws FormatIdentifierException, AuthorizeException;
}
```

The `identifyFormat()` method attempts to identify the data format of the given Bitstream, and delivers its results by adding a new `FormatHit` at the appropriate point on the *results* list. It returns the resulting list (possibly either modified or replaced) as its value; the caller must anticipate that it may be a different Object.

An identifier method can add results to anyplace in the result list, or use the default algorithm implemented by `FormatHit.addToResults()`. It is described in the next section, *Implementing Automatic Format Identification*.

The identifer method should throw `FormatIdentifierException` when it encounters a fatal error that prevents it from properly identifying the format of the Bitstream. Otherwise, there would be no way to tell the difference between a Bitstream that does not match any of the formats this method identifies, and a fatal error in the identifying code (e.g. configuration problem), since the results list is simply returned unmodified in both cases.

When the identifier cannot return valid results because of a temporary condition that may be cleared up later – e.g. a network resource that is temporarily unavailable – it can throw the {{FormatIdentifierTemporaryException}}to indicate that the results may change in the future.

A note on the object lifecycle: One instance of `FormatIdentifier` is created per JVM; it gets cached and reused. The `identifyFormat()` method is assumed to be thread-safe. If it is not, the implementing class should have it call an internal method which is synchronized on itself.

Typically, only the internal `FormatIdentifierManager` code ever calls identification methods.

## `FormatIdentifierManager`

This is a static class to operate the plugin stack and return a format identification verdict. Applications use it instead of calling the `FormatIdentifier` plugin stack directly.

```
package org.dspace.content.format;
public class FormatIdentifierManager
{
// identify all formats matching "target", returning raw hit list.
public static List<FormatHit> identifyAllFormats(Context context, Bitstream target)
throws AuthorizeException
//
// identify format of "target", returning best hit (never null).
public static FormatHit identifyFormat(Context context, Bitstream target)
throws AuthorizeException
//
// identify format of "target", AND set results in the Bitstream
public static void identifyAndSetFormat(Context context, Bitstream target)
throws SQLException, AuthorizeException, FormatRegistryException
}
```

The `identifyFormat` method always returns a hit. If the Bitstream was not successfully identified, it makes up a hit containing the unknown format.

## Controlling the Format Identification Process

The format identification framework is based on a sequence plugin, which gives administrators complete freedom to add and rearrange identification methods.

The `FormatIdentifier.identifyFormat()` method is very powerful; it actually controls the entire process of automatic format identification, even though it is called from deep within the framework. The `FormatIdentifierManager` only calls the stack of identifier methods in order and collects the results they provide. Each DSpace administrator has complete control of the methods run and the order of their execution, and the methods determine the results. The format identification API was designed to be very flexible, and also to make it easy to implement new identification methods.

Each implementation of `FormatIdentifier.identifyFormat()` can do whatever it wants with the Bitstream and list of results it is given. It might be a "filter" method that prunes the results of any below a certain level of confidence. It could look at other results and try to refine them, or reorder them.

An archive administrator can even insert a different framework into this one by configuring just one method that calls out to the other framework, and translates its results.

## Implementing Automatic Format Identification

Many different techniques and software products have been developed to identify the format of a data file; it is still a somewhat mysterious art. The best choice of methods and their ordering depends on your archive's users and the materials they most often submit.

Plugins with the best chance of precisely identifying a format should usually be first on the list, if they use the default result-ordering method that gives priority to the first hits among others of equal confidence.

Note that some plugins may depend on other identification methods running before they do because they *refine* an identification already found on the *results* list. Special relationships like that must be well documented so the administrator is aware of them.

Each `FormatIdentifier` plugin applies its special knowledge or resources to attempt to identify the format of the Bitstream; it is not responsible for solving the whole problem. For example, take a plugin that executes a heuristic to detect comma-separated-values files. It might collaborate with another method that detects plain-text files, so that it only applies its algorithm to *refine* the format identification if it sees from the results that the file is plain text.

### Random Access to Bitstream Contents

One problem that has not yet been completely addressed by this design is that many format-identification methods require *random access* to the contents of a Bitstream, but the Bitstream API only offers serial access through a Java `InputStream`. *Random* access means reading a sequence of bytes from the Bitstream starting at any point in its extent; this is very helpful when looking for an *internal signature* to identify the file, since the signature may be located relative to the end of the file or at some larged offset into it.

There are techniques to compensate for the lack of random access, although they sacrifice efficiency. It may also be necessary to add a method to `Bitstream` to retrieve a random-access stream when the underlying storage implementation supports it.

### Format-Hit Comparison Algorithm

Follow these steps when comparing two format identification hits to determine which has priority. This is implemented as the method `FormatHit.compareTo()`.

1. If the Namespace of the identifier cannot be resolved (looked up), i.e. because there is no FormatRegistry configured for it, that hit loses. See `FormatRegistryManager.find()`.
2. Order hits by their FormatConfidence index. Thus, hits based on the content of the Bitstream rate more highly than ones based on external attributes like the name.
3. Between two equal hits, if one has a non-null warning it is ranked lower.
4. When a hit has a *conflict* (that is, there is a lower-ranked hit which disagrees with it because the MIME type is different or similar), it is ranked below hits of the same confidence.

### *Default* Recommended Format Identification Algorithm

This is the *default* algorithm that is implemented by `FormatIdentifier.identifyFormat()` methods that simply call the `FormatHit`'s `addToResults()` method on each hit they develop.

NOTE: It is *not necessary* to use this algorithm. As described above, the format identification process is completely under the control of the `identifyFormat()` method implementations.

1. Start with an empty *results* list.
2. Call the `FormatIdentifier.identifyFormat()` method of each plugin in the sequence in turn:
   - Passing it the Bitstream and list of accumulated results so it can add new results.
   - If it has a *better-confidence* match than the current head of the list, that hit becomes the new head of the list.
   - Otherwise the hit gets appended to the end of the list.
3. When finished, the head of the list is the best format match.

To select a `BitstreamFormat` from the results, follow these steps:

1. Starting with the first result, take the first format identifier and namespace that can successfully be resolved into a `BitstreamFormat` (importing a new one if necssary).
2. If no `BitstreamFormat` is available, result is the *unknown* one, and set the confidence to `UNIDENTIFIED`.

This logic is encapsulated in the `FormatIdentifierManager.identifyFormat()` method.

If an application wants to generate a dialog showing all of the results of an automatic format identification (e.g. to give an interactive user the chance to second-guess the automatically-chosen format) it could call the plugins and process the results according to the algorithm above. We don't anticipate anyone wanting such a service, but if it comes up, we can always add another method to `FormatIdentifierManager`.

### Applying the results to a `Bitstream`

The properties of a `Bitstream` describing its format and the confidence of its identification have analogues within the `FormatHit` structure. The logic to map between them is encapsulated within the `FormatHit.applyToBitstream()` method, so there is only one piece of code to update if either of those objects changes in the future.

### Implementing a `FormatIdentifier` plugin

As mentioned before, each `FormatIdentifier` implementation only has to do part of the job, so it can be very narrowly focused. It can also look at the results of previous methods to decide if it has anything to add to the overall solution. For example, a method that heuristically identifies text-based formats would only proceed if it saw that a previous method had identified the data
as generic plain text.

Each method may add several {{FormatHit}}s to the result list, or none at all.

As an example of the power and flexibility of this approach, imagine a site that accepts many different kinds of XML formats, and needs precise identification of a few of them (e.g. METS documents of various profiles).

- A signature-based format identifier notices that the file starts with `"<?xml"` and adds a hit for the generic format "XML", and MIME type `text/xml`.
- The text heuristic identifier methods don't bother running because there is already a higher-confidence positive identification (of the generic XML format).
- A table-driven specific XML identifier method notices the hit for the generic XML format, and parses enough of the file to match one of the XPath specifications in its configuration. This identifies the file as an IMS Content Package manifest, MIT OCW version 1 profile.
    - If the XML parse had failed, the plugin could add a warning to the generic "XML" hit since it was obviously not well-formed XML.
- Results include the "OCW-IMSCP" format first, "XML" next, and perhaps other generic hits after it.

### Handling Conflicts

A *conflict* arises when the automatic identification process returns hits for incompatible formats. This is commonly caused by contradictory clues in a Bitstream, for example, a filename extension that a different format than the one indicated by internal signature matches. Consider a Bitstream containing a well-formed XML document; the "internal signature" method correctly identifies it as XML. However, its name ends with `".txt"` which is only listed as an external signature for other kinds of formats.

We may wish to record a warning (e.g. in the server log) when the results include such a conflict. There is no place to record it in the data model, but since warnings are mainly of interest to an archive administrator tuning identification or diagnosing problems, the log should be good enough.

## User Interface Elements Related to Data Formats

As further detailed in the use-case document, here is the minimum set of UI functions that the data format infrastructure has to provide:

1. Displaying a human-readable description of the format to the end-user.
2. Choosing from among all available data formats:
    - Just the formats in the `BitstreamFormat` table.
    - All formats available in selected external registries.
3. Selecting a `BitstreamFormat` from among the results of an automatic format identification, with the option of choosing freely as in Case #2.
4. Administrative interface to add and update `BitstreamFormat` objects.
5. Administrative interface modify chosen format of `Bitstream` (existing UI can be used with minimal changes).

### Display

The UI needs to display the data format of a Bitstream to the user in a meaningful way. Historically, this has been accomplished with the `name` (formerly "short description") property of the BSF, which is a short human-readable label such as "Adobe PDF 1.2". In some contexts it may be helpful to also cite the *confidence* property of the Bitstream to indicate how the format was discovered so the user can tell how much to trust it.

There are also some contexts - e.g. when offering a selection among Bitstreams to download - when it is helpful to include the MIME type of each Bitstream as well. Although MIME types are not meaningful to all end-users, they do tell the more technically sophisticated ones what the browser (or other HTTP client) is likely to do with a Bitstream.

To summarize:

- The BSF's *name* property is the primary user-visible identity of a Bitstream's format.
- The BSF's MIME type is sometimes helpful, some users find it more familiar than arbitrary format names.
- In some contexts, also indicate the *confidence* of format identification.
- Perhaps make the BSF's *description* available e.g. through a mouseover.

### Choosing A Format

First, what range of formats do you want to be able to choose from?

1. All `BitstreamFormat` names? This typically includes only formats of objects that have already been imported into the archive.
2. All of the formats in a given external registry, or set of them? This is likely to be more complete, but brings with it the problem of handling a very large list, perhaps thousands of choices.

In the first case, the problem is easily solved, but not so useful in most applications; if you are choosing a new format for a Bitstream, why limit your choice only to formats of Bitstreams already in the archive? It is mostly useful when your *purpose* is to choose among {{BitstreamFormat}}s, e.g. picking one to edit.

The second case is more helpful when actually selecting a format for a real Bitstream, but brings with it other problems - e.g. how to obtain and navigate formats from an external registry:

#### Choosing Formats from an External Registry

For good reasons, we propose to sidestep the whole problem of interfacing DSpace internals to an external registry to choose a format, and leave it to be settled between the DSpace user interface
application and the external registry. Here is why:

Current data format registries are large and growing – PRONOM has over 400 already, the TrID proprietary registry has over 2,600 entries, and the GDFR stands to acquire thousands when it becomes operational. They each have distinctive taxonomy and relationship metadata to help navigate the format space; for example, GDFR is developing a faceted classification system. Although GDFR may emerge as the standard, there is currently no effective standard for data format metadata.

Also, there are potentially several interchangeable DSpace UIs, each with differing capabilities and styles.

Given the complexity of implementing solutions for the cross product of format registries and DSpace UIs, we think it is more productive to let each UI negotiate with the format registry of its choice to produce a navigible display of formats. For example, the UI can transfer control to a popup or dialog encapsulating the registry's UI or a registry-specific extension. All it has to do is return a format identifier that can be resolved or imported to a `BitstreamFormat`.

The alternative is to force all registries into a common model, which would probably deprive them of the metadata most helpful to generating a good navigation interface. Each registry has unique features in its data model to facilitate browsing.

### Choosing / Confirming Automatic Identification

Since automatic format identification is powerful and fairly reliable, it makes sense to use it to assist users in identifying the format of their submissions, at least by narrowing down the choices. The automatic process yields a list of hits, which should be presented differently from a list of available formats:

- Indicate default choice of format and its MIME type.
- Ordering is significant, hits closer to the head of the list take precedence.
- The *confidence* metric on each hit is highly significant in helping the user evaluate them, so include it prominently and in a way that makes its values easy for naive users to understand.
- Offer an "escape route" to choose a format from all available formats. This becomes the default if automatic format identification failed.

### Editing `BitstreamFormat` Table

See the next section for details, this is classed as an administrative operation.

# Administrative Operations Relating to {{BitstreamFormat}}s

The DSpace administrator manages data formats with these operations:

## Initial Setup

### Configuration

The external registries are chosen by adding their names to a plugin interface as shown here. Note that the plugin name, which is also the namespace it covers, gets supplied by plugin itself through the `getPluginNames()` method. The order is **not** significant. This configuration example includes registries implementing the PRONOM, DSpace, and Provisional namespaces (guessing from the classnames).

```
plugin.selfnamed.org.dspace.content.format.FormatRegistry = \
org.dspace.content.format.PRONOMFormatRegistry, \
org.dspace.content.format.DSpaceFormatRegistry, \
org.dspace.content.format.ProvisionalFormatRegistry
#

    1. initialization files configured as "contact URIs":
       formatRegistry.DSpace.document = /dspace/config/registries/dspace-formats.xml
       formatRegistry.DSpace.validate = true
       formatRegistry.DSpace.schema = /dspace/config/registries/formats.xml
       formatRegistry.Provisional.document = /dspace/config/registries/provisional-formats.xml
       formatRegistry.Provisional.validate = true
       formatRegistry.Provisional.schema = /dspace/config/registries/formats.xml
       formatRegistry.PRONOM.contact = http://www.nationalarchives.gov.uk/PRONOM/Format/proFormatSearch.asp?status=listReport#
```

Format identifiers are configured in a sequence plugin, as in this example:

```
plugin.sequence.org.dspace.content.format.FormatIdentifier = \
org.dspace.content.format.DROIDIdentifier, \
org.dspace.content.format.UnixFileIdentifier, \
org.dspace.content.format.DSpaceFormatRegistry
```

### Conversion from Previous DSpace Version

To convert a DSpace 1.5 archive to the renovated BitstreamFormat code, see BitstreamFormat Conversion Instructions for instructions on running the conversion script. It will:

- Alter tables for the new data model.
- Check for alterations to the standard released BitstreamFormat registry and preserve them in the Provisional registry.
- Convert all referenced BitstreamFormats to the new model.
- Optionally use automatic format identification to set any formats that could not be directly converted.

The conversion process alters the archive as little as possible. If the backward-compatible **DSpace** namespace is configured, existing formats are simply mapped to that registry. Otherwise, every Bitstream has to be automatically re-identified.

## Reports

The following reports are helpful to administrators to check and validate format-related configuration options, and to plan preservation activities. They are generated by command-line administrative applications.

### Format Identification Testing: Success Rate

This report is intended to let administrators see the effect of changes to the format-identification configuration by testing how existing Bitstreams would now be identified. Given a group of DSpace objects to work against, this test runs the automatic format identification against each member Bitstream but *does not* change anything.

It reports the number of cases where the re-identification reaches a different result than the existing identification, and optionally shows each one in a detailed report. The report includes:

- Total number of Bitstreams processed
- Count of results with same format but a different confidence metric ("Unknown" failing again counts here).
- Count of results with a different format.
- Count of failures to make any identification.
- Optionally, details about each Bitstream with a different format result:
    - Bitstream identification
    - Previous format and confidence
    - Newly identified format and confidence

### Histogram of Formats In Use

Operating over a range of selected Bitstreams, this report shows the number of Bitstreams identified as each format. This lets the administrator see what formats are in use, and the relative proportion of each one, for the selected Bitstreams. It can be helpful when planning for preservation, since it immediately shows the number of Bitstreams in problematic formats.

The report includes, for each `BitstreamFormat` in use,

- BSF's name
- External identifier(s)
- Count of Bitstreams referring to it.

## Maintenance

### Edit `BitstreamFormat` Metadata

Some administrators will undoubtedly have a need to make local customizations to the descriptive and technical metadata for data formats. These attributes of a `BitstreamFormat` may all be customized by overriding the values imported from the remote registry – and the overrides persist even when the BSF is updated from its external registry.

- Name
- Description
- MIME Type
- Canonical File Extension
- Add or Delete External Format Identifiers
- Change the primary External Format Identifier

#### Update BSFs From Remote Data Format Registries

Update the local copies of technical metadata from the originals in remote format registries. Options are:

- Update a single BSF
- Update all of the BSFs whose metadata comes from a particular registry (i.e. Namespace).
- Update all BSFs.

A timestamp of last update is maintained for all BSFs. When performing a group update, use the timestamp farthest in the past as the limit when searching for changed formats in the remote registry. After a group update, set the time of last update of *all* relevant BSFs to the time of this operation.

### Edit `Bitstream` Technical Metadata

Here are the cases where a Bitstream's format technical metadata must be modified:

#### Retry Format Identification

Rerun the automatic format identification, perhaps after configuration changes or improvements to a remote registry. This is the same operation as the automatic format identification performed on ingest.

It can be done both interactively, for a single Bitstream, or in batch for a set of selected Bitstreams. The interactive UI should offer a choice of viewing and accepting the new identification choice.

#### Override Format Choice

Manually (interactively) force the choice of a new data format, chosen from either:

- The existing set of `BitstreamFormat` entries.
- An explicitly specified namespace and identifier referencing an external format, which is imported if necessary.
  - This may be a simple text-entry box since it doesn't have to be user-friendly.

### Maintain "DSpace" and "Provisional" Format Registries

The *DSpace* registry is updated by modifying or replacing its XML configuration file. The new contents will be loaded automatically when DSpace is next started.

Similarly, the *Provisional* registry is maintained by editing its XML configuration file. Its contents depend entirely on the local administrator, however.

### Remove an External Format Registry

Before removing an external format registry from the configuration, all references to it must be removed from BSFs. There is a utility administrative tool to manage this automatically, with proper checks, i.e. so it does not delete the last (primary) external format identifier from a BSF which is in use.

In a typical scenario, the archive administrator decides to get rid of one of the external format registry plugins, e.g. the "DSpace" registry. By removing it from all of the BSF entries first, she ensures there will not be any reference failures after it is removed from the plugin configuration – although theoretically, the only normal DSpace operation that would fail is an update from the remote format registry.

# Omissions and Future Work

## Your Comments

Please use the Discussion tab for your comments on and reactions to this proposal, since comments mingled with this much text would be too hard to find.

## Preservation Applications

Although adequate support for data format representation is a necessary foundation for preservation activities, this work does not include any actual preservation functions. These are all subjects for other projects:

- Detecting and notifying administrators of obsolete formats in the archive.
  - Use Policy Engine to manage reaction to obsolescence.
- Format migration and *normalization* (migration on ingest).
  - Control normalization requirements within Communities and Collections with Policy Engine.
- Data format validation (some of which is already implemented in pending JHOVE integration work).

## Container Formats

This proposal does not include any explicit support for features of *container formats*, that is, data formats such as "tar" and "Zip" which serve primarily as "wrappers" for other data objects. Containers typically implement some or all of the following functions:

- Bundle other files together into a unit.
- Apply data compression to the contents.
- Apply encryption and/or digital signature to the contents.
- Encode the contents as alphanumeric text (e.g. "base64").
- Add metadata to content file(s).

Reasons not to consider explicit support for containers at this time:

1. Container formats *are* supported as opaque formats without any special properties. They are identified and disseminated with technical metadata. This should be adequate for formats to be preserved as single Bitstreams, e.g. JAR.
2. In some contexts, e.g. package-based ingestion, containers are unwrapped upon ingestion so the container itself is not relevant as a preservation issue.
   - The Packager mechanism can be extended with new plugins to ingest and disseminate additional types of packages, so the DSpace can just store the contents and metadata.
3. Putting a container into a digital archive is antithetical to the principles of preservation: it increases the difficulty and risk in recovering the original data by adding another layer of format to interpret.
4. There is currently no compelling use case for generalized container support in the DSpace content model.

## Documentation of Data Formats

Attaching thorough documentation about the interpretation of a data format (i.e. standards documents) is an important preservation tool. We do not need to make any provisions for this within DSpace if we trust external data format registries to maintain it as format technical metadata.

In particular, the GDFR architecture allows for a locally-administered GDFR node, with full local copies of all data, to be integrated into the DSpace software. It includes provisions for documentation on interpreting each format. We believe we can rely on the GDFR to solve this problem when it is fully developed.

---

Please use the Discussion Page for your comments on this page.

# Other Documentation

- A presentation on the work described in this wiki at Open Repositories 2008, Southampton, UK - [ Slides|http://mit.edu/sands/www/bfr/Sands%20Fish%20-%20Bitstream%20Format%20Renovation.ppt]
- Paper presented at OR08: *BitstreamFormat+Renovation_DSpace Gets Real Technical Metadata*