A Developer’s Guide to the

Repository Open Service Interface Definition

Jeffrey Kahn
# Table of Contents

**Front Matter** .......................................................................................................................................................... 1  
  - Who Should Read This Book ................................................................................................................................. 1  
  - Scope of this Book ................................................................................................................................................... 1  
  - Use of Java™ Language Binding and Terminology .................................................................................................. 1  
  - Typographical and Other Conventions .................................................................................................................. 2  
  - The O.K.I. Licenses ................................................................................................................................................ 2  
  - Acknowledgements .................................................................................................................................................. 2  
  - About the Author .................................................................................................................................................... 2  
  - Getting Started ...................................................................................................................................................... 3  
  - Just a Few Architectural Themes and OSID Details .............................................................................................. 3  
  - Put On a Hat ............................................................................................................................................................ 3  
  - What A Repository Project Looks Like .................................................................................................................... 5  
  - A Bare-Bones or Proof-of-Concept Effort ................................................................................................................ 6  
    - Simplifying Assumptions ......................................................................................................................................... 6  
  - How to Build a Bare Bones Consumer ................................................................................................................... 6  
  - How to Build a Bare Bones Provider ....................................................................................................................... 10  
    - Provider Minimal Method Checklist ...................................................................................................................... 12  

**OSID Concepts** ...................................................................................................................................................... 13  
  - Types – How OSIDs Get Specific While Staying Abstract ....................................................................................... 13  
  - Unique Identifiers .................................................................................................................................................... 15  
  - Iterators .................................................................................................................................................................... 16  
  - Exceptions ............................................................................................................................................................... 16  
  - OsidContext ............................................................................................................................................................ 17  
    - Sharing data across OSIDs ........................................................................................................................................ 18  
  - OSID Approach to the Object Life Cycle ................................................................................................................ 21  
    - Why “get” methods are called “update” .................................................................................................................. 22  
    - Why There Is No “update” Method For Each “get” Method .............................................................................. 22  
  - OsidLoader ............................................................................................................................................................. 22  
    - Obtaining an Instance of an Object that Implements an OSID’s Manager Interface ........................................ 23  
    - Using a Different Loader ........................................................................................................................................ 23  
  - Assumptions About Machine Boundaries and the Network ................................................................................... 23  
  - Integration ............................................................................................................................................................... 25  

**Consumer’s Guide** .................................................................................................................................................. 29  
  - Sample Scenario ..................................................................................................................................................... 29  
  - Iterators ................................................................................................................................................................... 30  
  - How to Work With an Iterator .................................................................................................................................. 31
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching</td>
</tr>
<tr>
<td>Non-String Criteria</td>
</tr>
<tr>
<td>Criteria Adjustments</td>
</tr>
<tr>
<td>User Interface Considerations</td>
</tr>
<tr>
<td>Displaying Repositories</td>
</tr>
<tr>
<td>Instantiating and Comparing Types</td>
</tr>
<tr>
<td>Publishing You Own Types</td>
</tr>
<tr>
<td>Deciding What Exception To Throw and With Which Message</td>
</tr>
<tr>
<td>Describing Repositories</td>
</tr>
<tr>
<td>What Kinds of Content Stores Can Be Hidden Under The Repository OSID</td>
</tr>
<tr>
<td>Repository OSID And Databases That Support SQL</td>
</tr>
<tr>
<td>Repository OSID and Content Management Systems With APIs</td>
</tr>
<tr>
<td>Repository OSID And Web Services</td>
</tr>
<tr>
<td>Searching</td>
</tr>
<tr>
<td>Section</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Search Tools</td>
</tr>
<tr>
<td>Asynchronous Searches</td>
</tr>
<tr>
<td>Versioning</td>
</tr>
<tr>
<td>Repository, Authorization, and Hierarchy</td>
</tr>
<tr>
<td>Adapters</td>
</tr>
<tr>
<td>Adapter as Multiplexer</td>
</tr>
<tr>
<td>Updating Content / Formed-Based Data Entry</td>
</tr>
<tr>
<td>Legacy Migration</td>
</tr>
<tr>
<td>Some Implementation Comments</td>
</tr>
<tr>
<td>Manager method implementations delegating to Repository implementations</td>
</tr>
<tr>
<td>The Intent of the copyAsset() Method</td>
</tr>
<tr>
<td>Getting Assets</td>
</tr>
<tr>
<td>Aggregating AssetIterators</td>
</tr>
<tr>
<td>Just-in Time vs Prepared Objects</td>
</tr>
<tr>
<td>Part.isPopulatedByRepository()</td>
</tr>
<tr>
<td>Asset.getContentRecordStructure()</td>
</tr>
<tr>
<td>Effective and Expiration Dates</td>
</tr>
<tr>
<td>RecordStructure Formats and Schemas</td>
</tr>
<tr>
<td>Repeatable Records and Parts</td>
</tr>
<tr>
<td>Data Migration</td>
</tr>
<tr>
<td>Downloading Asset Content</td>
</tr>
<tr>
<td>Uploading Content</td>
</tr>
<tr>
<td>Other Topics</td>
</tr>
<tr>
<td>Compliance</td>
</tr>
<tr>
<td>Installation and Deployment</td>
</tr>
<tr>
<td>Consumer Deployment</td>
</tr>
<tr>
<td>Provider Deployment</td>
</tr>
<tr>
<td>OSID Versioning</td>
</tr>
<tr>
<td>Candidate Methods for a Future Release of the OSID</td>
</tr>
<tr>
<td>A Few Final Comments</td>
</tr>
</tbody>
</table>

Copyright © 2005, Massachusetts Institute of Technology
Front Matter

Who Should Read This Book

This book is for software developers

There is a saying: often experience is what you get right after you need it. The intent of this book is to give you the benefit of the experience before you need it.

This book has a narrow focus on the practical material developers need to consume or provide access to digital content. While the programming examples in this book are in Java, this book is intended for all programmers, not just those writing in Java.

If you answer yes to any of these, this book is for you:

☐ I need to access repository content programmatically and have been told to use something called “O.K.I.” or an “OSID”.

☐ I need to provide access to repository content programmatically and have been told to use something called “O.K.I.” or an “OSID”.

☐ I need to access various content sources. The communication protocols or metadata schemas I am supposed to use vary. I am looking for a single architectural approach to make this task more manageable. I don’t want my application to have code that embeds any particular way to get to this content.

☐ I need to provide access to repository content, but my customers want me to use different standards or I don’t want to be bound to any particular implementation technology.

☐ I need to operate both on the web and with desktop clients. Content may be local or remote. I want to hide all this variability so I can use a single approach.

☐ I am looking for a standards-based approach to content. I don’t want to be at the mercy of vendors that are offering proprietary solutions that give them the upper-hand.

If you manage software development or are interested in repositories or interoperability in general, you may find the ideas in this book helpful.

Scope of this Book

This book concentrates on:

- implementing the O.K.I. Repository Open Source Interface Definition (OSID), and
- consuming someone else’s implementation.

A Repository OSID implementation touches directly on the Shared and Id OSIDs and may touch indirectly on the Authentication, Authorization, and Logging OSIDs. Only the Repository OSID is addressed in depth.

There are other works that cover the other OSIDs and broader architectural and philosophical concerns. Refer to the O.K.I. project website (http://www.okiproject.org) for information about other sources.

Use of Java™ Language Binding and Terminology

The early adopters of OSIDs have been Java developers. While there has been some effort in PHP, Objective-C, C# and other languages, Java dominates. This book provides programming examples in Java and uses the Java language binding of the OSIDs.
This book uses the Java term for certain elements that have a different name in another language. For example, an *interface* in Java maps to a *protocol* in Objective-C. While an interface is *implemented* in Java, the protocol is *adopted* in Objective-C.

Method names may vary across language bindings. In some languages, “method” isn’t even the right term. “Function” may be. In this book, I use the Java language binding method names.

**Typographical and Other Conventions**

- Programming objects, data types, source listings, and other similar content appears in *monospace*.
- The names of OSID objects are in proper case.
- For brevity, an object that implements an interface is often shortened to just the interface name. For example, instead of writing: “an object that implements the Repository interface”, you will find: “a Repository”. Similarly, I say throw UNIMPLEMENTED as a short form of saying, “throw an exception with the message defined as UNIMPLEMENTED in the such-and-such OSID”.
- All method names appear in *monospace* and end with (). Note that this is the case even for methods that take arguments.
- The phrase “get” methods means all methods with the prefix ”get”. The same convention is used for other prefixes.

**The O.K.I. Licenses**

The OSIDs come with a license. This license covers MIT’s rights to the OSIDs and states that they should not be changed. Any OSID consumer or provider should comply with the terms of the SID Definition. The license is the same for all language bindings.

All code fragments use fully qualified names. No *imports* are used.

**Acknowledgements**

Many people contributed to the interplay of ideas from which this book emerged. Mostly directly, this work acknowledges the support of Scott Thorne, Jeff Merriman, Peter Wilkins, and Vijay Kumar.

**About the Author**

Jeffrey Kahn has been working with O.K.I. since 2002. As a consultant to MIT, Jeff made contributions to the architecture, documentation, reference implementations, and evangelism. Jeff has helped developers in many organizations to understand OSIDs. Jeff lives outside Atlanta with his wife, two children, and a golden retriever named “Peaches”.

Copyright © 2005, Massachusetts Institute of Technology
Getting Started

Just a Few Architectural Themes and OSID Details

Here are a few themes that you should just accept as information for now. They will help clarify why the OSIDs contain what they do. We will go into a lot more detail on these later.

- Improved interoperability is a goal. In order to achieve this, we do some things in a more indirect or abstract way than is necessary for a binary integration of two known parties at one point in time.
- Applications should be easy to write, even if that makes implementations harder to write.
- Service implementation approaches will change, so applications should be able to easily swap one for another. The details of the implementation should be hidden.
- All methods throw specific exceptions, one per OSID. All exception reasons are defined in the OSID and no others should be used. Return codes are not used.
- Almost every OSID object has a unique identifier.
- OSID providers are responsible for managing the lifecycle of OSID objects and for persisting their data. There are dedicated methods for these operations and “new” is not used.
- OSID methods that return multiple objects use typed iterators rather than, say, arrays.
- OSIDs use Types, basically labels that state the kind of thing an object is. The community defines the meaning of these Types. The OSID just gives us a place to hold these Types.
- To allow for OSID bindings to non-object-oriented languages, the OSIDs make minimal use of sub-classing.

Put On a Hat

I expect you have heard the expression, “wearing my developer hat, I say ...”. People use this idiom to let someone know that some comment stems from a particular vantage point or set of responsibilities that you are adopting rather than the position you will always hold. As a developer, you may be asked to wear more than one hat simultaneously. In practice, this is impossible, at least without looking silly. What you wind up doing is switching the hat you are wearing. This is always a challenge and can lead to confusion. Sometimes other people don’t know which role you are adopting when you say something, hence the use of the idiom is an effort to make things clear. So, wearing my O.K.I. architecture hat, I am leading up to something. Wearing my fellow-developer hat, I am looking for the point.

The architecture that underlies the Open Service Interface Definitions (OSIDs) places a boundary between service consumers and service providers. The consumer knows only the interfaces and nothing about the provider’s implementation detail. Similarly, the provider does not know its consumer. The consumer and provider
have a contract – “I’ll consume you this way and you will implement this interface to grant me access to a service”. The principal benefit of this approach is that consumer and provider are free to evolve independently.

As an OSID developer you may be asked to develop both consuming applications and OSID implementations. In any case, you need to be certain you wear one and only one hat at a time. Be wary of wearing both hats simultaneously and thereby building in knowledge of consuming application detail into your provider or provider detail into your consuming application. This idea is so important I am going to say it again. Wear one and only one hat at a time. Be wary of wearing both hats simultaneously and thereby building in knowledge of consuming application detail into your provider or provider detail into your consuming application.

As a consumer you might say, I know the OSID implementation I am using does XYZ and so I am going to make this assumption or optimization. Or I know the OSID implementation is really getting its content from this database or web service and so, just here, I am going to go directly to the source. As a provider, you might say, I know my caller is going to have a specific UI widget for this data and so I am going to return information in this way to make things work out. Try not to do this.

The consuming application may be closer to the presentation portion of an overall solution. The OSID implementation is concerned with things like getting to the content and working with metadata. The implementation should also worry about authorization. This is an important part. Note that I am not putting authorization (and authentication) in the domain of the OSID consumer. The OSID implementation is the part of the system getting the content and it is the implementation that should enforce authorization. I will explore this topic in more detail later.

Figure 1: OSIDs are a Boundary Between Consumer and Provider
So why is this mingling of consumer and provider so bad? After all, if you need to switch consumer or provider, you can always go in and rework that piece of dependent code. As is covered in great detail in other documents, this is a slippery slope that undercuts a range of value propositions for:

- return on investment
- implementation reuse
- managing complexity over the lifecycle of an application
- substitutability and the emergence of best-of-breed solutions
- test reuse
- customer choice
- time to market

What A Repository Project Looks Like

People ask this question a lot. They want to know what the steps might look like. Here is one outline; your experience will vary:

1. Review the O.K.I. Case Studies and other materials to see if the value propositions address your needs. OSIDs make sense when integration and interoperability are goals. OSIDs are not a magical solution to all design challenges.
2. Talk to some O.K.I. folks or others to feel comfortable.
3. Decide if you are a consumer, provider, or both.
4. If you are a provider, develop a read-only, simple search and retrieve provider. Make one repository with a display name. Make one type of asset and simple metadata. You may be best off modifying sample code. For testing, there is a repository dump utility that exercises an implementation. Don’t worry about only implementing some of the methods. The utility is very tolerant. Of course you can also try your provider with a real consumer application. Use already published Types.
5. If you are a consumer, develop a simple search and retrieve consumer. Display assets in some basic way. Use already published Types.

6. Evaluate this proof-of-concept so you can scope a complete implementation.

A Bare-Bones or Proof-of-Concept Effort

Often developers are asked to prepare a proof-of-concept or engage in a “discovery” activity as a prelude to creating a complete Repository OSID implementation. This work may be part of a scoping exercise or may be part of a campaign to “sell” the merits of a full implementation effort. This section is designed to help you accomplish this task directly and quickly.

Simplifying Assumptions

Since we are discussing a proof-of-concept, we are going to make some simplifying assumptions. A complete implementation deals with the areas we are going to skip and I’ll address those later. We are going to avoid major areas such as authorization. We will assume the consumer can search for content and get results. We also will not add, remove, or update content. It is easier to say what we will include than to say what we will leave out. We are going to focus on basic search and retrieve.

Base Bones Consumer

- Obtain a repository instance
- Call a repository search method, receiving assets in return
- Examine each asset

Bare Bones Provider

- Support repository instantiation
- Support searching the repository for assets that match some criteria
- Convert the result set to assets
- Return assets to caller

How to Build a Bare Bones Consumer

As a first pass we are going to do the following:

1. Load an instance of a class that implements the RepositoryManager interface
2. Get the Repositories that the manager supports
3. Call a search method on each Repository we find
4. Print the title of each asset returned
Here is a look at this sequence with a little more detail about the objects that get created and what methods we are going to call:

Figure 3: Sequence of Steps in a Bare-Bones Consumer

Some people like to look at diagrams; other people like to look at source code. Here is a code fragment that does our bare bones consuming:
try {
    java.util.Properties properties = new java.util.Properties();
    org.osid.repository.RepositoryManager repositoryManager = org.osid.OsidLoader.getManager("org.osid.repository.RepositoryManager", "edu.xyz.osidimpl.repository.foo", context, properties);
    java.io.Serializable searchCriteria = "abc";
    org.osid.shared.Properties searchProperties = null;
    org.osid.repository.RepositoryIterator repositoryIterator = repositoryManager.getRepositories();
    while (repositoryIterator.hasNextRepository()) {
        org.osid.repository.Repository nextRepository = repositoryIterator.nextRepository();
        System.out.println("Found a Repository called "+nextRepository.getDisplayName());
        org.osid.repository.AssetIterator assetIterator = nextRepository.getAssetsBySearch(searchCriteria, new org.osid.types.mit.KeywordSearchType(), searchProperties);
        while (assetIterator.hasNextAsset()) {
            org.osid.repository.Asset nextAsset = assetIterator.nextAsset();
            System.out.println("Found an Asset called "+nextAsset.getDisplayName());
        }
    }
} catch (Throwable t) {
    System.out.println(t.getMessage());
}

Figure 4: Source Code for Bare-Bones Consumer

Line-by-Line Commentary

All OSID methods can throw exceptions. This means that any OSID method calls should be enclosed in a try…catch block. For now, it is sufficient to just catch throwable. Line 2 instantiates an OsidContext object. For now, we just need one of these. The same holds for a properties object that we instantiate in Line 3. Lines 4…9 contain a single method call. This call uses an OsidLoader class to obtain an instance of an object that implements the Repository OSID’s RepositoryManager interface. Line 10 initializes a search criterion to “abc”. Typically, this data originates with user input. Line 11 initializes an object that implements the Properties interface in the Shared OSID. We can ignore the details for now. Line 12 obtains the set of Repositories that the RepositoryManager knows about. Note that this set is available through an iterator. In OSIDs, iterators are Typed and you use hasNextType and nextType methods to test for more objects and get objects in succession, respectively. In Lines 13…15 we get the next Repository and print out its display name. Lines 16…19 contain a single method call. This is the crux of searching in the Repository OSID. Searches take three arguments: criteria, a Type, and properties. Types are an important concept in OSIDs. For now, it is enough to know we are passing in a request for a keyword search. We can ignore the properties argument for now as well. Lines 20…22 get each Asset returned by the search and print its display name.

We have skipped over an enormous amount of detail, but the outline of consuming the Repository OSID for searching should have emerged.

Let’s dig one level deeper by looking at metadata. I am going to cover this a lot more later, but we can take a peek. Each Asset has zero or more Records. Each of the Records has zero or more Parts. Each Record also has a Record Structure. Parts have Part Structures. When you create a Record you pass in a specific RecordStructure; when you create a Part, you pass in a specific PartStructure. These structure assignments are immutable. If it helps, you can think of Structures as the schema and Record and Parts as the rows and fields in a database.
Figure 5: Modeling a Database Table Using Structures, Records, and Parts

The OSID model is more flexible than just a database, and with that flexibility comes a little heavier weight. For example, Records and Parts have identifiers. Enough for now. Here is how you get the metadata for an Asset:

1. given an Asset, get its Records
2. given a Record, get its Parts
3. given a Part, get its Part Structure
4. given a Part Structure, get its display name or its Type's description (metadata field's name)
5. given a Part, get its value (metadata field's value)

In code, this looks like:

```java
1. org.osid.repository.RecordIterator recordIterator = nextAsset.getRecords();
2. while (recordIterator.hasNextRecord()) {
3.   org.osid.repository.PartIterator partIterator = recordIterator.nextRecord().getParts();
4.   while (partIterator.hasNextPart()) {
5.     org.osid.repository.Part nextPart = partIterator.nextPart();
6.     String field = nextPart.getPartStructure().getDisplayName();
7.     String value = nextPart.getValue();
8.   }
9. }
```

Line-by-Line Commentary

As discussed earlier, all OSID methods can throw exceptions, so a `try..catch` block should enclose this fragment. Line 1 gets the Records for an Asset. Note that some Asset have Records, some don't. An Asset might just have a content object. In that case, call `getContent()` on the Asset. You can tell what kind of object
you are going to get in various ways we will discuss later. Assets that contain only metadata probably don't implement `getContent()`. Each Record has a Type and you can ask for all the Records for a particular Type. For now, we get them all. Lines 2..5 use the iterator methods to get us the next Part. Lines 5..7 extract the display name of the Part's Part Structure and the Part's value. Here we have seen how to get the metadata. How you display this metadata is another topic.

Where to go from here

Assuming you want to prove that you can get some content from your source through the OSID abstraction, the material above should do the trick. If you are being asked to implement the OSID as well, refer to the next section on the bare-bones provider.

How to Build a Bare Bones Provider

As a first pass, we will do this:

1. implement a minimal `RepositoryManager`, which includes trivially `assignConfiguration()`, `assignContext()`, `osid_version_2_0()`, and what we really need, `getRepositories()`.
2. implement a `Repository` with `getDisplayName()` and `getAssetsBySearch()` methods.
3. Implement an `Asset` with `getDisplayName()`. Also implement a protected constructor.

In order to make the Java compiler happy, you will need implementations for all the interface methods.

```java
package foo;

class RepositoryManager extends org.osid.repository.RepositoryManager {
    public org.osid.repository.RepositoryIterator getRepositories() throws org.osid.repository.RepositoryException {
        return new RepositoryIterator(new java.util.Vector().addElement(new Repository("Sample Repository")));}
}
```

The `RepositoryManager` interface implementation principally exists to return a list of `Repositories`. Note I really mean classes that implement the `Repository` implementation, but I will make things shorter with, I trust, no loss of clarity. We will explore other things the manager does later on. For now, this sample constructs a sample `Repository` and returns it in a `RepositoryIterator`. Iterators are typed in the OSIDs. My sample implementation has an iterator constructor that accepts a vector.

```java
class Repository extends org.osid.repository.Repository {
    private String displayName = null;
    protected Repository (String displayName) {
        this.displayName = displayName;
    }
    public String getDisplayName() throws org.osid.repository.RepositoryException {
        return this.displayName;
    }
}
```

Copyright © 2005, Massachusetts Institute of Technology
The sample Repository implementation, above, has a protected constructor that accepts a string. This string will be returned as the display name for the Repository. We use protected constructors because it is a principle in OSIDs that all object creation is done via a dedicated create method. The reason we do this is simple. OSIDs intend that implementations persist created objects. Creating an object by calling “new” does not imply this. To make applications easier to write, persistence is assumed and is the responsibility of the implementation. There are fewer worries for the application at the expense of adding one more concern to the implementation.

The heart of the Repository implementation may be the getAssetsBySearch() method. This method interprets the criteria based on the search Type and does the search. The search Type may also tell the implementation what to do with the search properties or the implementation has its own uses for these properties. Note that an implementation’s use of these properties outside the published Type agreement undercuts interoperability. Implementation configuration may be a better place for this. The search method returns an AssetIterator. Here that contains just one new Asset.

Asset.java

... public class Asset implements org.osid.repository.Asset {
...
  public String getDisplayName() throws org.osid.repository.RepositoryException {
    return this.displayName;
  }
}

The Asset implementation would have a protected constructor under the same model as the Repository. That code is omitted in this text. Above we just return a display name.

We have covered the basics of the implementation. Now I would like to comment on metadata:

If you are going to provide metadata, one option is to just return all the metadata via the Asset’s getContent() method. I don’t advocate this, but it is legitimate. A more flexible and interoperable model is to leave getContent() for non-metadata, that is the real Asset. For example, if the Asset is an image, the content is the image and the metadata describes the image. If we are going to return metadata, we need at least one Record. If we have a Record, we need a RecordStructure. A Record needs at least one Part. If we have a Part, we need a PartStructure. So the simplest thing is to always implement all four of these objects.

One RecordStructure may be enough. There is a lot of room for developer choice here. For example, if you are going to support more than one metadata schema, say Dublin Core and IEEE LOM, each could have in its own RecordStructure. Alternatively, this could just be two Parts in a common RecordStructure. Another example is if you have image metadata and also a few image sizes already made (i.e. you are exposing a service where the caller asks for the size image they want). Here the images could be in a separate RecordStructure. You could also use the Asset content for multiple images. Since the method returns an object, it can be anything including something that returns multiple image sizes.

Let’s start with one Part containing the complete metadata using Dublin Core. Here is the minimum you need, based on likely consumer behavior. There is no guarantee the consumer will call these methods, but we are just exploring an example:

---

1 Image metadata is sometime called a Tombstone.
1. The consumer is going to call `getRecords()` on the Asset. There is a form that allows for a filter by RecordStructure Type, but you could skip that. So you need a RecordIterator and at least one Record.

2. For the Record, the consumer is going to call the Record’s `getParts()` method. This means you need a PartIterator and at least one Part.

3. The consumer is going to call `getValue()` for the Part. To know what kind of value, the consumer is going to call `getType()` or `getDisplayName()` on the Part’s PartStructure.

**Provider Minimal Method Checklist**

**Asset**
- `getRecords()`

**RecordIterator**
- `constructor`
- `hasNextRecord()`
- `nextRecord()`

**Record**
- `constructor`
- `getParts()`

**PartIterator**
- `constructor`
- `hasNextPart()`
- `nextPart()`

**Part**
- `constructor`
- `getValue()`
- `getPartStructure()`

**PartStructure**
- `constructor`
- `getType()`
- `getDisplayName()`

Rather than return the entire metadata as, say, a string of XML, you may want to offer the metadata as individual elements. There is a more interoperable approach because it does not require the consumer to understand how to parse and interpret any particular metadata schema. If one provider uses Dublin Core, another IEEE LOM, and a third something proprietary, that means the consumer needs to know how to deal with all three. Life is much simpler for the consumer if the providers use common abstractions (Types). For example, all three schemas may have “keywords” although they have different ways of expressing them. Better to use a Part with a value and a PartStructure whose Type is a community defined “keywords”². Note that you may want to offer the complete metadata as well for those consumers that want to handle information that way.

**Where to go from here**

We have looked at a bare-bones provider implementation. At this point, you may want to pop up a level and look at some OSID concepts. Alternatively, you can look more deeply into providers.

---

² The O.K.I. project has already published many Types relating to repository and common metadata fields.
OSID Concepts

I am going to cover some architectural material here. The discussion is going to focus on what is useful for writing a consumer or implementing a provider. In general, please try and accept these concepts and concentrate on the pragmatic issues. I know many readers will want to understand this material in a lot more depth and understand the justification for why the architecture contains what it does. That kind of depth is outside the scope of this work. Consult the material on the OKI project website, http://www.okiproject.org for additional documentation.

Types – How OSIDs Get Specific While Staying Abstract

The title of this section probably has you scratching your head. How can something be specific and abstract at the same time? Let’s recall some of the OSID architectural philosophy and assumptions. OSIDs are designed to help make it easier to write applications and maintain them for a long time – particularly across multiple service implementation technology changes. One way to accomplish this goal is to keep specific protocol and data format detail out of the OSID. This is all fine and good, but how is the application going to be able to perform useful work if nothing is specific? The answer is the OSID Type.

The Type is a label. The Type gives a label to information a community shares about what an object is or what a process is. For example, a community defines an “image” asset. All image assets are supposed to have some well-defined characteristics. The details are published for the community. The OSID provides methods to hold these Types.

In our image asset example, there is a getAssetType() method for an Asset. A consumer calls this method to find out what kind of Asset it received from the provider. The provider implements the getAssetType() and returns a Type known to the community, which includes the provider and the consumer.

Everything the community needs to know about a Type is formalized in an Out-of-Band Agreement (OBA). You can find the community OBAs on the O.K.I. project website.

Types generally label the kind of thing an object is or the kind of process a provider should undertake. An example of a kind is “image”. An example of process is “title search”.

Types play a role in the dialog between consumer and provider. Here are some examples:

- What Types of Repositories do you have?
- Please give me your Repositories of Type xyz?
- I have a Repository, what Type is it?
- What Type Assets to you have?
- Please give me your Assets of Type xyz?
- I have an Asset, what Type is it?
- Please search using these criteria. This is the Type of search I want and you can interpret the criteria accordingly.

A type contains four strings:

- Authority
- Domain
- Keyword
- Description

These strings can have any value.

By convention, type strings use camelCase with a leading lower case character. For example “partStructure”. Try and avoid underscores, periods, commas, slashes, and other common delimiters in the strings. Try and keep the
names short but not cryptic. The descriptions should be short so that they can work in a UI list. Lengthy Type descriptions are better off in the OBA.

Two Types are the same if their authority, domain, and keyword strings match exactly. The description string is for information only.

You may want to store Types. You may want to store them as a single string, ignoring the description. There is no defined way to represent the authority, domain, and keyword as a single string. One best practice is the format domain/keyword@authority. For example: search/title@edu.mit.

Types are the way that OSIDs leave a place for detail, but leave the specification of the detail to a process outside the OSID. You might wonder how applications can last any longer or be any more flexible if they have Type-specific behavior. The answer is mixed. Applications may not need to do anything different based on a Type. It may be enough to pass it around or show it to the user. For example, if an application is going to ask a repository for assets of a particular Type, the user might pick the Type from a list the application generates from asking the provider what Types are supported. Here the application doesn’t have to know what the Type is, just that the provider supports it and the user picked it. In some cases, the application may not even have to know how to display the asset based on its Type. The application might get a URL from the repository and pass it to a browser for display or may allow for plug-in asset viewers — supplied by the provider’s vendor.

Another example of user-selected Types is a search Type. An application might have a plug-in criteria editor, or just a generic data entry field. The repository tells the consumer what search Types it supports, the application presents these to the user, the user picks one and enters criteria with a syntax specified in the published search Type agreement, and the application passes criteria and Type to the repository.

![Figure 6: Sample Use of Types in Search for Assets](image)

There is a proposal to add a `displayName` to Types in a future OSID version. This would offer short text suitable for display in a list and the description as supplemental information. Currently, since the description is long, it might or might not be the right thing for a user to pick in a list.
Unique Identifiers

OSIDs give ids to almost every object. The id should be unique within a scope that is appropriate to avoid ambiguity. For example, no two assets in a repository should have the same id. Globally unique ids are fine and may even be the most desirable, but there is no requirement that the ids be globally unique.

Ids should not contain any semantic information intended for consumers. The caller is supposed to use ids but not make any behavior change based on the value of the id. This approach means that applications can substitute providers without worrying about id semantics.

Ids have a string representation that can be any length. Consumers should be alert to this when they plan a user interface. Since ids have no semantic value, showing them in a UI may be of limited value.

Many objects have both an id and a display name. The id is intended for unambiguous identification. The display name is for show. A simple example is an Agent. The name of the agent is something a user is supposed to see and interpret. It may or may not be unique. The agent’s id might be shown, but it is just data. The purpose is to have a way to identify this specific agent and no other. The agent id should be unique at least across the system of record.

People have asked how to generate unique ids. There is an Id OSID and it has a `createId()` method. This returns an id with a unique string representation. There are many algorithms for generating unique strings.

It may be helpful to have a single Id service for an enterprise. Every system can use the same OSID provider. Ids are not Typed. For example, there is no difference between an id for an asset and an id for a course. If you need to have different algorithms for ids for different kinds of objects, then use multiple providers.

Consumers often pass ids to providers. As a consumer, you might have an Id object in hand. Alternatively, you might need to make one, perhaps from something you store. What you stored is likely the id's string representation rather than the id object. Strings are easier to save than objects and surely no harder. The Id OSID has a method, `getId()`, that returns an Id instance with a particular string representation. As an example of the process, there is a Repository method, `getAsset()` that takes an Id as input and expects to get the Asset back that has this Id.

![Diagram showing the flow of passing an id from a consumer to a provider](image)

**Figure 7: Consumer Passing Id to Consumer**

When a consumer passes an Id to a provider, the provider often needs to compare it to known ids. For the provider, the Id has a method `isEqualTo()`. This tests Ids for equality.
Iterators

OSIDs always use typed iterators to return multiple objects.

Iterators are beneficial when working with large objects or large numbers of objects. You can ask for each object, one at a time. You can discard the object when you are done with it. This should free up resources for the next object.

Iterators also add flexibility in providers’ choice of implementation. If, for example, all the objects for a result set need to be ready before the method returns any, this may impose some limits on the way you implement the provider. Only having to have the results available one at a time gives the provider more choice. A simple example is a service that gets objects remotely. Another situation could be an asynchronous process. Perhaps the results, say of a search, come in at different times. By the time the consumer has gotten to what was the last object, a new one might be available.

Iterators are useful when the underlying data structure does not address elements by index. Simple examples are trees or hash tables.

I’d like to separate the justification for iterators from the justification for iterators of objects of a specific Java object type. Iterators are widely used and I think you will see why we used them. Preferring iterators of typed objects to a generic iterator is more style or taste. For example, the RepositoryManager’s getRepositories() method returns a RepositoryIterator. It could have returned an Iterator instead. The advantage of the typed iterator (that is, an iterator that returns a specific Java object type) is that you don’t need to cast and the code is a little more self-documenting. We may also be reducing the chance of a casting error. One other point. A theme in OSIDs is that we want the consumer to be as easy to write as possible — even if that makes the provider’s job harder. Typed iterators are a little easier for the consumer.

Exceptions

Exceptions are used for method implementations to tell their caller that some problem has come up. The circumstance may signal something really wrong (e.g. a communication or system failure) or just a localized problem (e.g. couldn’t find some object given the id provided). OSIDs use exceptions rather than error codes. The exact mechanism is going to vary by language binding, but the concept should be similar.

The OSID philosophy is that providers offer limited detail about their inner workings. This helps foster provider substitutability and thereby increases the chances applications can survive technology change. Since providers may use different techniques, implementation-specific exceptions could be raised, but the application handling them would have to be adjusted to deal with each provider’s set. Since this would cause us to change the consumer, we might just have code that handles all exceptions in the same way. While this is a viable approach, treating every exception the same way does not offer any path for graceful recovery — all exceptions would be presumed fatal. The OSIDs strike a balance between treating all exceptions the same way and implementation-specific exceptions. The OSIDs declare one exception each and enumerate the causes.

- Providers should not throw any exception other than the one included in the OSID.
- Providers should not use a message other the ones defined by the OSID’s exception.
- Consumers should handle exceptions.

Here are some common exception messages and when to use them in a provider.

<table>
<thead>
<tr>
<th>OPERATION_FAILED</th>
<th>A general failure you don’t expect the consumer to be able to do anything with.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIGURATION_ERROR</td>
<td>Something is not configured correctly or a configuration setting cannot be found. This exception is commonly thrown in the assignConfiguration() method in the manager.</td>
</tr>
</tbody>
</table>

\(^4\) Called a reason, cause, etc in other languages
PERMISSION_DENIED | An authorization was not found for some operation. In response, an application might want to show an appropriate alert, stipple a menu, disable an option, etc.

UNIMPLEMENTED | This means there is no implementation for this method. Since methods can be implemented à la carte, this should not be unexpected. As with PERMISSION_DENIED, an application may need to adjust the UI. Note that some objects include methods like supportsXYZ. These methods return a boolean and help a consuming applications know more about what an implementation can do. This can affect the choice of provider to use or combination of providers to use. It might also change UI. The Repository object has methods supportsUpdate() and supportsVersioning().

UNKNOWN_ID | The id the caller has passed in is not known by the provider.

UNKNOWN_TYPE | The Type the caller has passed in is not known by the provider.

NULL_ARGUMENT | Some argument is NULL that should not be.

All OSID methods declare one exception they can throw. The exception is OSID-specific. For example, the Repository OSID's methods all throw the org.osid.repository.RepositoryException.

The Java language binding, uses subclasses as follows:

```java
java.lang.Object
  |__ java.lang.Throwable
     |__ org.osid.OsidException
          |__ org.osid.shared.SharedException
               |__ all other OSID-specific exceptions
```

All exceptions other than org.osid.OsidException subclass org.osid.OsidException.

Sometimes ignoring an exception is the right thing to do. For example, in a federated search with several providers, the consumer may want to ignore an exception thrown when calling getAssetsBySearch(). Rather than abandon the entire search, the caller keeps going.

OsidContext

The OSID methods cover the service activity, but not anything implementation specific. We needed a way for the consumer and the provider to exchange data, if necessary. The same applies to sharing data among OSID implementations. The OsidContext is the way.

The OsidContext is an object that holds key-value pairs. The keys are strings and the values are objects. The OsidContext is not intended for configuration information. We already have the assignConfiguration() method for this. Note that the assignConfiguration() method is unidirectional, from the consumer to the provider, but that is probably fine. The OsidContext is intended for two-way exchange, or at least it is possible. OSID implementations can also assign and get configuration in a chain. This can come in handy since one OSID implementation often needs another.

So what might you put in the context? One common practice is authentication-related information. This might be credentials, a session token, or something similar. One model has the Authentication OSID authenticating and placing some data in the context. This data might be needed by the Authorization OSID or the Repository OSID to help to know who the user is or what they can do.
Another use has come up in passing username and password information to an Authentication OSID implementation. The OSID implementation may be able to put up its own prompts for this data, or get it out of the context. The consumer can call for authentication with or without a UI.

There are really no rules on what can or cannot go into the context. Just a couple notes:

- The context is implemented in `org.osid.OsidContext`. This implementation should suffice.
- Context keys need to be unique. You might want to use a package name as a prefix to avoid a collision.
- If you are publishing a Type agreement with references to context keys, you might want to use the Type as a prefix. One convention for representing a Type is `domain / keyword @ authority`. For example, `search/keyword@edu.mit`.
- Any assignment for a key overwrites whatever value may already have been assigned.
- If you don’t need a key-value pair any longer and you know no one else does, you should remove the pair.
- While there is no method to get the keys in the context, the intent is to exchange information simply. No security support is included.
- The context can be assigned at manager implementation load time or later. The OsidLoader always calls `assignContext()`.

Sharing data across OSIDs

Let’s start with a common but boring case. The OsidLoader requires an OsidContext in the `getManager()` call. The loader could have been written with a method that omitted this argument, or the properties argument, or both. Rather than offer these four variations, the burden in placed on the consumer to provide these arguments. One of the few cases of making things a trifle harder for the consumer. Since all that is needed is to call `new`, the burden is not a heavy one. Here is a sample `getManager()` call:

```java
org.osid.repository.RepositoryManager repositoryManager =
    (org.osid.repository.RepositoryManager)org.osid.OsidLoader.getManager(
        "org.osid.repository.RepositoryManager",
        "edu.xyz.osidimpl.repository.foo"
    new org.osid.OsidContext(),
    new java.util.Properties());
```

The following picture tells the tale. The consumer creates a new OsidContext. The context is empty. The consumer creates a new properties, also empty. The consumer calls `getManager()`, a static method in the OsidLoader, and passes in the interface they want, a package, the context, and the properties. The loader is going to instantiate the provider and call the assign methods to pass in the context and properties. A lot of mechanics and nothing really passed.
Now let’s look at something more interesting. The consumer instantiates a new, empty OsidContext and then puts something in it. The context is passed, via the OsidLoader call, to the provider.

Figure 9: Passing a Non-Empty OsidContext to a Provider

As you probably have guessed, there is no rule that a consumer can’t also be a provider. In fact, most OSID implementations use other OSIDs. Some common examples are:

- Repository OSID uses Id OSID to generate new Ids.
- Repository OSID uses Logging OSID to report activity history or errors.
- Repository OSID uses Authorization OSID to determine where to enforce limited-access control.
- Authorization OSID uses Authentication OSID to authenticate user prior to answering whether the user is authorized to perform some operation on some object.

Here is a picture of a provider passing along the OsidContext it has to another provider. There is no requirement that there be only one OsidContext. The OsidContext is not implemented as a singleton. In practice, you may want to pass all providers the same OsidContext. This offers maximum visibility. On the other hand, you may not want to do this. The agreements that accompany Types will state whether anything in particular should be in the context.

![Figure 10: Passing the OsidContext Along to a Dependent Provider](image)

You may want to add your own information to the context before passing it along.
As a consumer, you start working with a provider implementation by obtaining a RepositoryManager. The intent is that there is no other approach. For example, if someone has given you a valid asset id, there is no way to instantiate that asset directly. You do not call `new`⁵. You obtain an instance of the manager and through that you can get an instance of the asset. This instantiation approach helps the provider carry out its responsibility for managing the lifecycle of OSID objects. OSID objects are created, added, updated, removed, and deleted using OSID methods. These methods need to insure that appropriate information is persisted in underlying system.

| create | instantiate and object and persist it |

⁵ Provider implementations should not use public constructors for OSID objects. Use protected or private visibility.
The provider is responsible for data consistency. Managers can implement the OsidTransactionManager interface. This interface includes mark(), commit(), and rollback() methods. Consumers and providers can use these methods to coordinate transaction support.

Why “get” methods are called “update”

In Java and other languages, methods that start with “get” return an object attribute or property. For example, getFoo() returns the value of the Foo attribute\(^6\). Methods that start with “set” assign a value to the attribute. Usually an implementation uses an instance variable to hold this data. With a “set” method, you normally don’t assume the assignment lasts beyond the object’s lifetime. With OSIDs, the provider is obligated to persist all assignments. We felt that “update” rather than “set” would underscore this commitment.

In almost all cases, the method is update. Exceptions are:

- assignConfiguration() and assignOsidContext() in the managers
- assignPriorityType() and assignFormatType() in the Logging OSID

Here the OSID uses “assign” in lieu of “set” simply to continue to stay away from that term. The intent with these methods is that data is assigned for the instance and is not persisted.

Why There Is No “update” Method For Each “get” Method

There are more “get” methods than “update” methods. For things like display names and descriptions, you can usually both get and update.

For id, you never update. The rationale is that the consumer should not be setting these, except in some cases in a create method. The provider needs to control these values and any semantic value they might have is exclusively the provider’s concern.\(^7\)

The consumer does not update the Type of an object. In some cases, the Type is set through a create method. The type should be immutable where they convey a kind. They may change where they convey a status or state.

Properties are also read-only, except during creation. The thinking is that you don’t want consumers changing these since the semantics of each key are not defined.

As of version 2.0 of the OSIDs, a model for updatable properties was not quite right. Rather than put something out that had the model wrong, we decided to defer this until version 3.0. For now, you could just add your own updateProperty() method, knowing in advance that interoperability may suffer.

OsidLoader

One of the value propositions of O.K.I. is implementation substitutability. The OsidLoader is a best-practice for loading a class from the Java classpath\(^8\). The class name is based on an OSID package name and an

---

\(^6\) In Objective-C, the “get” is omitted and assumed.

\(^7\) One exception is the Hierarchy OSID where providers just manage nodes and those nodes have ids that are meaningful only to the consumer.

---
implementation package name. The loader also assigns the OsidContext and configuration that is passed in. The approach offers a dynamic and generic solution that has worked well in certain applications.

**Obtaining an Instance of an Object that Implements an OSID’s Manager Interface**

The OSIDs distribution jar includes a “best practices” OsidLoader class. This returns an instance of an implementation of the manager interface for a particular OSID.

**Using a Different Loader**

As mentioned above, the OSID jar ships with the class `org.osid.OsidLoader`. Other loaders can be used following these requirements:

- OSID implementation consumers should only obtain objects that implement OSID interfaces through the manager or other OSID objects. Consumers should not be able to call `new` on any of these objects.
- Any loader must call `assignOsidContext()` on any manager immediately after instantiation.
- Next any loader must call `getConfiguration()`, add to the properties returned in the loader call, overwriting if needed, and then call `assignConfiguration()`.
- The loader should check the version of the OSID the implementation supports. This is done by verifying that the method `osidVersion_2_0` is implemented. This can happen at any time during this initialization.
- All the above gives implementations a chance to set up. No other OSID method should be called until this initial set happens.

**Assumptions About Machine Boundaries and the Network**

OSIDs don’t make these assumptions. In keeping with the philosophy of making applications simple, OSIDs always involve a local call. Whether there is a network and where it is does not play a role for the consumer. This approach allows the network and all its complexity to come and go and evolve on the provider side of the OSID boundary. The complexity doesn’t go away, but it is compartmentalized.

Let’s start with a desktop client application. Suppose we have three Repository OSID plug-ins, one per content source. Each OSID implementation uses a different mechanism to get to content, but the consuming application does not need to be concerned about this. The consuming application calls the OsidLoader to obtain RepositoryManager implementations from local jar files on the client machine. The consuming code can treat each OSID implementation in the same manner, without regard to where the content is managed.

8 The best-practice loader varies by binding. For example, in Objective-C, the loader uses bundles in a location set by convention.

9 Other bindings use different approaches for validation.
Let's consider another model. Suppose we have a web browser on a client machine. The web browser communicates with a web server application. On the web server, let's say in a servlet, is code to obtain an instance of a RepositoryManager from a jar located locally on the server.

In either arrangement, or in variations on these, the principle remains the same. The OSID consumer calls the OSID provider implementation locally. All connectivity issues lie with the provider.
Integration

One of the O.K.I. architecture’s central value propositions is ease of integration. Let’s start with a simple model: binary integration between a consumer of some service and a provider of that service. This happens all the time. I have an application “A” and I want to connect it to a content service “B”. Using a common approach, I design application “A” to have a module that will handle the connection, I put all the “B”-specific logic in that module, and I am done. If I want to get fancy (and I have the luxury of time and resources), I might make a general interface (or “B” exposes one) and have the module implement it. I do this because I foresee that I may change something in the module and don’t want that change to disturb the application.

![Figure 16: Simple Binary Integration](image)

There is nothing wrong with this approach, but there can be challenges later on. The issue of managing complexity is covered well in other documents, but here I am simply going to state that good modular design keeps the application unaware of the details of how it communicates with the content management system. If that mechanism can be neutral with regard to implementation technology, all the better. This is what OSIDs are. They are nice, abstract interfaces, designed for services most applications need. OSIDs are intended to be stable and last a long time. Here is the same diagram using OSIDs:
Using “OSID” language, we say an application consumes an OSID. The semicircle represents the OSID (interfaces). The circle represents the OSID implementation. This provider connects to content in some way; the details are not important.

Now let’s introduce some change.

- We want to go after a second content source; a new requirement. Maybe we have already finished the application or maybe it is just late in the cycle.
- Let’s say that the second content source exposes a completely different mechanism for communication. This is not so strange. The first source might have offered a web service or used a database driver or a custom API.
- When we wrote the application to start we might have decided to consume the public interface of the first content source since it was a popular standard. We didn’t worry about connecting to something with a different approach since we assumed other sources would offer the same mechanism. Maybe we foresaw the need for something more general, so we wrote that, but it still had some technology assumptions or we just didn’t get the factoring quite right to cover the new case. Recall the OSIDs were designed over time and with these problems in mind – without the pressure of a product to get out the door.
Here is what we might wind up with:

![Diagram showing Application with two data sources: Content 1 and Content 2.](image)

**Figure 18: Adding a Second Data Source Without OSIDs**

Well this isn’t so terrible. The application can handle this level of complexity. If the number of data sources grows, we might take the time to rework the interface to something more general so we can consolidate the connection code and simplify. After a couple of cycles of this, we would probably wind up with something that looks a lot like an OSID, but of course we would then like others to use our interfaces.

Jumping back to the OSID approach, here is where we would be after the second data source. Note that the interface is the same for both sources. Another benefit is that we can just substitute the second data source for the first, we don’t have to have both if we don’t want to.

On another topic, we should consider the case where the application and content provider are being written by different teams on different schedules. In our ad-hoc mode, the integration needs to be clarified at the start and remain stable throughout the project. This is easier said then done. With OSIDs, there is neutral ground on which to agree at the start.
As a final picture, let’s introduce another application. It is a whole lot easier if they also use the OSID. Now you can swap in new providers and consumers. Let me state this directly:

- Each new provider is available to all consumers
- Each new consumer is available to all providers

This is not just theory. There are case studies on the O.K.I. project site with specific examples.
Consumer’s Guide

The documentation is going to diverge here. I have tried to put information for the consumer in this section and the provider another section. As I mentioned in Put On a Hat, you should adopt one or the other role, not both simultaneously. This approach results in a little duplication of this section in the provider, but not much. I have tried to put all the material in common in the section on OSID Concepts.

Sample Scenario

To better understand what consumers might do, here is a scenario: A professor is developing material for a survey course of modern Western art. The professor wants to begin by searching repositories of several kinds and at several institutions. Initially, interest is not limited to images of well-known works. Perhaps reviews or articles about trends in modern art, video of technique, books about the subject, Internet sites devoted to the topic, etc. are all relevant. An application provides a way for the professor to identify the parameters of the search, execute it, and view results. Each result that meets the professor’s needs might be added as an asset in a special repository for the professor’s and the students’ use. Since there are many issues relating to the right to access and present material, there might be a range of approaches to handle assets. An asset might be a link, a copy, encrypted content that can be played only in a specific viewer, etc. The professor assembles the collection and then students interact with the repository in a variety of ways. There are a variety of metaphors for aggregation and delivery such as the shopping cart or basket, the slide show, the light box, filing cabinet and so on.

Different assets will have different intellectual property rights and access restrictions. Some information can be included by reference such as a link, but the material cannot be incorporated directly. Other information may be copied and used directly. Even so, such material may be large and therefore better incorporated by reference only. Some material may be encrypted in such a way that it can only be played through a specific application or only for a specific duration. Each player may have a unique key that is used when encrypting material so that use is tailored to the individual. The same application might be used for assets from a variety of sources.

The OSID approach facilitates integration. For example, the vendors or developers of repository systems such as DSPACE, FEDORA, ARTstor, and so on might develop or have developed implementations that present these products through the Repository OSID. An application, written to the OSID would work with one or more back-end services. Tools and approaches for searching, aggregating, or delivering content such as SRW can also use the OSID as an integrating standard. By employing the OSID, applications and back end systems can be reused more easily and more often.

Suppose we have a simple digital repository of images. We might have a single AssetType, namely “Image”. Associated with each AssetType might be two mandatory RecordStructures, one related to the image and one to the rights and source of the image.

The image RecordStructure might contain a PartStructure for whether the image is black and white or color. Another PartStructure might contain the image size in pixels. A third PartStructure might contain the image format, such as GIF, JPEG, or TIFF. Here is how these the Image RecordStructures might be defined:

RecordStructure

<table>
<thead>
<tr>
<th>DisplayName</th>
<th>- Image RecordStructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>- Organizes data on image color, size, and format</td>
</tr>
<tr>
<td>Id</td>
<td>- Assigned by implementation</td>
</tr>
<tr>
<td>Format</td>
<td>- This could be plain text, XML, or something else</td>
</tr>
<tr>
<td>Schema</td>
<td>- This could be a reference to a standard</td>
</tr>
<tr>
<td>Validation</td>
<td>- Might check for non-null entries</td>
</tr>
<tr>
<td>Type</td>
<td>- Defined by the community</td>
</tr>
<tr>
<td>PartStructures:</td>
<td></td>
</tr>
<tr>
<td>Color</td>
<td></td>
</tr>
</tbody>
</table>
PartStructure

Display Name: Image Color InfoPart
Description: Organizes data on whether an image is color or black and white
Id: Assigned by implementation
is Mandatory: Maybe yes
is Populated By DR: Maybe no
is Repeatable: No
Validation: Might check color or black and white as only valid entries
Type: Defined by the community

…Similar PartStructures for Size and Format

The RightsAndSource RecordStructure might contain a PartStructure for whether the image is freely available or restricted, who owns the image, and the original source of the image. Associated with each of these RecordStructures can be one or more Records for each Asset. In this particular case, probably one Record would make sense for the Image RecordStructure. One or more for the RightsAndSource RecordStructure – for example, there might be different records for different kinds of rights (ownership, reproduction, international, etc).

When you create a Record you pass in a specific RecordStructure; when you create a Part, you pass in a specific PartStructure. Any Record for an Image RecordStructure needs to have Parts for each Mandatory RecordStructure and will likely have Parts for all PartStructures. An implementation can make these Parts or let the Application make them. A sample image Record’s Part values might be:

- ColorPart: Color
- SizePart: 640 x 480
- Format: JPEG

A sample rights InfoRecord might be:

- AccessPart: Unrestricted
- OwnerPart: National Image Library
- SourcePart: 2003 Photographic Survey

Now we are going to pop back up a level to the general.

Iterators

Any OSID method that returns more than one object is going to do this with an iterator. Here are some things to know about iterators:

- You don’t know how many objects you are going to get, at least until you are told there are no more.
- You should test if there is an object available before you ask for the object itself. If you ask for an object and there are no more, you will get an exception. It is best to avoid this by checking first.
- You can’t skip over objects, you have to get them one at a time and in the order they are provided.
- There is usually no guarantee about the order of objects. There are a couple of exceptions to this rule: e.g. in the CourseManagement OSID.
- You can’t go backwards – you can’t ask for the previous object again. This also means you can’t restart. You have to ask for the whole iterator again.
• There is no guarantee that the provider has all the iterator objects ready. There may be a delay if you ask for objects faster than the provider has them ready. Imagine you are really fetching objects one at a time over a network.

• You are under no obligation to ask for all the objects. You can stop asking for the next one at any point.

• OSID iterators are Typed. There is a dedicated iterator for each object Type. In Java, an iterator returns a Java Object and you usually need to cast. The cast is not required with an OSID iterator. Typed iterators make source code more readable and remove the chance for a runtime cast exception. This is really an issue of style.

How to Work With an Iterator

Iterators have two methods:

hasNext: returns true if there is another object, false if there are no more to be had

next: returns the next object

For example, the iterator, org.osid.repository.RepositoryIterator has the two methods:

• public boolean hasNextRepository()

• public org.osid.repository.Repository nextRepository()

To print out all the display names of all the repositories, you do this (assume repositoryManager is non-null):

```java
org.osid.repository.RepositoryIterator repositoryIterator = repositoryManager.getRepositories();
while (repositoryIterator.hasNextRepository()) {
    org.osid.repository.Repository repository = repositoryIterator.nextRepository();
    System.out.println("Next Repository’s display name is " + repository().getDisplayName());
}
```

Of course there are ways to code this loop more succinctly, but you get the idea.

How to Work Around Iterator Inconveniences

One annoying issue with iterators is that you don’t have a length as you do with an array. You need to go through the iterator fully first. You probably want to store the objects somewhere. You need a data structure with a mutable size. A vector works in Java10. There are other structures. Once the objects are in a vector you can access them by index.

Iterators are handy when the objects are large. Maybe you want only one at a time and you are going to discard each one before you get the next one. Getting all the objects and storing them in a vector is probably a bad idea. Maybe it would be better to serialize them locally for subsequent random access.

Types

Instantiating Types

As a consumer you probably will need to work with Types in these ways:

• Instantiate a Type

• Pass Types to a provider

• Compare the Type a provider returns to ones you know about

---

10 In a Cocoa / Objective-C application you might use a NSMutableArray
• Store a Type and reconstitute it later
• Display a Type in a UI

Let’s look at these in detail. To instantiate a Type you can call `new` on a class you write or one the provider supplies. The class can:

accept arguments – a general purpose class that returns instances of many Types

not accept arguments – a Type-specific class with a default, empty constructor.

To implement you own Type class, simply subclass `org.osid.shared.Type` and call `super()`, either with no arguments or with a defined set:

```java
public class anyType extends org.osid.shared.Type {
    public anyType (String authority, String domain, String keyword, String description) {
        super(authority,domain,keyword);
    }
}
```
or

```java
public class fooType extends org.osid.shared.Type {
    public fooType () {
        super(edu.xyz","someDomain","foo");
    }
}
```

If you need a Type that is in a published out-of-band agreement, there may be a jar with the types already implemented. Instantiating one of these classes avoids the risk of a typographical mismatch.\(^\text{11}\)

To pass a Type to a provider method, you just need to instantiate it and pass the object in the method call. Note that providers generally have no need for the description of a Type. It is not used for comparison.

To compare one Type to another, call the `isEqualTo()` method on one of the Types and pass in the other. The equality test ignores the description attribute. You might want to load a provider implementation and find out what types it supports. If you need a type it doesn’t support, you may want to load a different implementation or adjust which type you use. Providers should publish the Types they support, but there may be some need for the consumer to adjust to different Types.

By convention, no method that returns an iterator of supported types should require special authorization. These methods are of the form `getXXXTypes()`. There is no such convention for information about the type of an object – methods of the form `getType()`.

If you need to store a type, you have several choices:

• You can store the Type object. In this case you simply deserialize the object to get an instance.

• You can store the string attributes. You must include the authority, domain, and keyword. You may want to store the description, maybe not. In this case, you pass the strings to a generic Type subclass or examine the strings to decide which Type-specific subclass to instantiate.

• You can store a string representation of the Type. One convention is `domain/keyword@authority`. In this case you need to parse the string representation to get the individual strings back out.

To show a type in a UI, either:

• show a string representation of the type in a list
• the description in a list
• attributes in columns of a table

\(^{11}\) Recall that the types strings: authority, domain, and keyword, are compared exactly for equality.
## Metadata

### Accessing a Single, Specific Metadata Field

Assets provide access to metadata through Records and Parts. Assets have zero or more Records and Records have zero or more Parts. Parts are analogous to the values of individual elements in a metadata schema. Two popular metadata schemas are Dublin Core and IEEE LOM. Each Record has a RecordStructure; each Part has a Part Structure. While a part corresponds to a metadata element value, the structure corresponds to the element. For a database, a record corresponds to a table row and a part to a column value. The structures correspond to the table schema.

![Entity Relationship for Metadata](image)

Each Record, Part, and structure has an id. Structures have types. There are methods for getting a specific Record or Part by id. Currently, there is no method to get a Part by the type of its PartStructure. Since any semantics for ids are known solely to the provider, as a consumer you are going to access all the metadata by iteratively walking the metadata or access a single, specific metadata field value by finding the Part whose PartStructure has the right type.

For example, suppose there is a thumbnail type for part structures. Assuming you don’t know anything about which Record holds the Part that has the value you want, you would do this:

```java
1. org.osid.repository.RecordIterator recordIterator = nextAsset.getRecords();
2. while (recordIterator.hasNextRecord()) {
3.   org.osid.repository.PartIterator partIterator = recordIterator.nextRecord().getParts();
4.   while (partIterator.hasNextPart()) {
5.     org.osid.repository.Part nextPart = partIterator.nextPart();
6.     if (nextPart.getPartStructure.getType().isEqualTo(thumbnailType)) {
7.       thumbnail = nextPart.getValue();
8.     }
9.   }
}
```

In this code, we assume the thumbnailType has already been instantiated. This could be done by calling `new` on a dedicated Type class in a jar that comes with the Thumbnail Type OBA or by using your own, generic type subclass and passing in the correct string 3-tuple. You might want to break out of the loop when you find the Thumbnail. There is probably just one. Also note that Parts can contain Parts. More exhaustive code would check Part’s Parts recursively.
**Metadata Inheritance**

Assets can contain zero or more Records. Each Record has a RecordStructure. These Records can be thought of as metadata. The metadata associated with one asset in a repository may have much in common with other assets. Whole sets of assets may have the same metadata except for a slight variation in one or two fields. This situation brings up two interesting issues:

- metadata ingestion
- propagating metadata changes

Let’s take a look at metadata ingestion first. It is possible that assets and their metadata will be entered into the content management system that underlies the repository using a process that runs outside the OSID. The content may already be in the system or it may be entered through a batch process or tools that feed the system of record. On the other hand, the OSID may be used to create some new assets and metadata. In this case, there are OSID methods to help.

One Asset method, `copyRecordStructure()`, adds records to the Asset\textsuperscript{12}. The method’s implementation makes point-in-time copies of another Asset’s records with a given record structure and adds them to the current Asset. Note the operation copies all records with this record structure. There could be one record; there could be more than one.

If you view this from the consumer’s perspective, you might have an Asset 1. You want to make a new Asset 2 that has the same metadata in almost all fields. Rather than enter the metadata manually, which is time consuming and prone to errors, you create the new Asset 2 and then call `copyRecordStructure()` passing in the id for Asset 1 and the id for the RecordStructure.

![Figure 22: Create a New Asset and Add Metadata from Another Asset](image)

\textsuperscript{12} This is the kind of method that may not be implemented by some providers. Of course, the provider should check the consumer has “update” authorization to perform this kind of operation.
Another scenario is where Asset 1 and Asset 2 already have their own metadata, but Asset 1’s records are for record structure A and Asset 2’s records are for record structure B. Now you want to add records for record structure B to Asset 1. This operation adds metadata with nothing being lost or replaced.

Figure 23: Adding Metadata from One Asset to Metadata for Another Asset

If the record structure is the same in both assets, the copy operation replaces content.

Figure 24: Replacing Metadata for One Asset with Metadata from Another
Let's now consider what happens when metadata changes. The copy operation takes the metadata from the source asset and copies it to the target asset, adding or replacing as appropriate. There is no link between these assets. If you change the metadata for the source asset, the target asset's metadata is unaffected. Let's suppose you do want more than this one-time behavior. You could call the `copyRecordStructure()` method again, but it can be hard to get the synchronization right. The OSID includes another method, `inheritRecordStructure()` that asks the provider implementation to keep the metadata for the target asset in sync with the source. Whether there is really a copy of the metadata or a reference is an implementation detail. Once the consumer calls `inheritRecordStructure()`, the link remains until it is broken by a call to `copyRecordStructure()`. After that target asset's metadata can change directly.

A  Consumer creates asset 1 with some metadata
B  Consumer creates asset 2
C  Consumer calls `inheritRecordStructure()`. Provider ensures that asset 2 inherits asset 1's metadata
D  Consumer changes asset 1's metadata, asset 2 reflects this
E  Consumer changes asset 1's metadata again, asset 2 reflects this
F  Consumer calls `copyRecordStructure()` to break inheritance.
G  Consumer changes asset 1's metadata – asset 2 is unaffected

Figure 25: Inheriting Metadata
Let’s explore this inheritance a little further. An asset might have three kinds of metadata:

- Presentation metadata
- Intellectual Property Metadata
- Content Metadata

It may make sense to put certain metadata in a record with a record structure you plan to inherit and other metadata in a record with a record structure you don’t plan to inherit. Perhaps only the presentation and intellectual property metadata are inherited. Note that not all assets need to inherit from the same asset. Sets of assets might inherit from different assets. Visualizing an inheritance hierarchy may help. Let’s assume color denotes some set of metadata. If a circle has the same color as its parent, it is inheriting the metadata unchanged. A change of color might indicate introduction of additional content for a new record structure.

Figure 26: Two Assets Inheriting Similar Metadata from Different Assets
There is nothing that says there can be only one inheritance hierarchy. Here we see separate hierarchies for presentation and digital rights management data.

Figure 27: An Asset Inheriting Different Kinds of Metadata from Different Assets
Searching

Federated Search

Federated search, searching more than one repository at a time, is more complex than searching a single repository. Here are some of the areas of additional complexity:

- Selecting which repositories to include in the search
- Deciding whether to return when you have responses from some but not all repositories
- Different metadata schemas for assets in different repositories
- Different query expression formats
- Different kinds of searches (for example, title search, author search, etc)
- Marking which repository a result came from
- Duplicate responses from assets that reside in more than one repository

This is a topic that has more depth than I will cover here. Just a few comments:

- You may want to support repository icons. The community should define an Asset Type for this.
- Using UI controls that key off the type of assets (content and metadata viewers is an example) may be a good way to remain adaptable.
- Similarly, different criteria editors based on search types will come in handy.
- You may want to have a way to fix up criteria and search types between the UI and the actual repository call.
- Giving each search a thread is a good practice.

Non-String Criteria

Criteria are commonly strings. At the simplest, a single word can do. In a more complex form, the string can be XQuery or CQL. Criteria can be something other than a string. One example is an image where the repository makes a pattern match.¹³

Criteria Adjustments

Suppose a provider repository does not support a particular search type. The repository might support another search type that the consumer can use, but it has different rules for criteria formatting. In this circumstance you have at least these options:

- Don’t search this repository.
- In the consumer, there may be logic to adjust the criteria and pass in an alternate search type.
- In the provider, there could be similar logic.

In either of the two options that involve logic, I suggest the consumer or provider provide a configurable extension mechanism for this. A simple interface, CriteriaAdjuster helps here. Without changing consumer or provider code, prior to any search a repository-specific criteria adjuster, if any, is called.

```java
public interface CriteriaAdjuster {
    java.io.Serializable adjust(java.io.Serializable searchCriteria, osid.shared.Type searchType);
}
```

¹³ Giunti Interactive Labs offers image matching of this kind.
Here is a sample implementation used in an implementation for the Celebrate federation. It takes a simple string, as specified in the “Title Search Type” OBA and convert its to XQuery.

```java
public class MyRepositoryCriteriaAdjuster
    implements CriteriaAdjuster {
    private org.osid.shared.Type titleSearchType = new edu.mit.types.mit.TitleSearchType();
    public java.io.Serializable adjust(java.io.Serializable searchCriteria , osid.shared.Type searchType) {
        try {
            if (searchType.isEqualTo(titleSearchType) && searchCriteria instanceof String) {
                String xquery = "<filter:filter schemaVersion="0.1"
                           xmlns:filter="http://celebrate.eun.org/xml/ns/filter-0_1"">
                return xquery;
            }
        }
        catch (Throwable t) {
        }
        return searchCriteria;
    }
}
```

Here is a code fragment that gets the id for a repository, checks for a criteria adjuster, and if present, applies it. Assume everything that needs to be defined has been.

```java
if ( (name = getCriteriaAdjusterForRepositoryId(repositoryId)) != null ) {
    Class adjusterClass = Class.forName(name);
    CriteriaAdjuster criteriaAdjuster = (CriteriaAdjuster)(adjusterClass.newInstance());
    searchCriteria = criteriaAdjuster.adjust(searchCriteria,searchType);
}
```

User Interface Considerations

### Displaying Repositories

You may want to show some information about the repositories supported through a manager. You may be building an application dedicated to a single repository or the application may want to hide this kind of detail. In that case, you may just want some branding such as the repository name or logo. This may be configured right into the consumer and not drawn from the repository returned by the manager.

Repositories can’t contain other repositories, so you might show providers and their repositories in a two-level-deep list (tree) or indent to show containment in a simple list, or use a logo for the provider and repository names. There are many choices here. To make things a little more complex, there can be a Type OBA for a “collection” asset. If you wanted to show these, you might have three-levels (or more), provider / repository / collection.

You can load one or more providers’ manager and they each have one or more repositories. Repositories have a type and you can ask for only repositories of a specific type. You can also get all the repositories and filter out the ones you don't want by type or other some other characteristic such as the types of assets they hold or whether they allow updates.

Each repository has an id, a display name, a description, and the type we just mentioned. There are several methods that tell you what kinds of assets a repository holds and what kinds of record structures an asset may and must contain. There is information on the search types (and other types) a repository supports.
So the choice is yours. You can do something simple such as display names in a list or something more comprehensive such as an “inspector” that shows all kinds of detail, or something in between.

A note on repository branding. I mentioned above that you might want to show a repository logo. There could be multiple logo sizes and other branding data such as a URL or a statement on permitted uses. None of this data is covered by specific methods in the OSID. One option is for the community to define a few asset Type OBAs for this integration. Since you can ask for an asset by type, a consumer could pluck out this data and show it in the user interface.

Displaying Search Types

Repository-related client applications typically offer a browser or search mode, or both. Given the enormous popularity of Google, Yahoo, Spotlight, and other search interfaces, many clients are offering a simple text field that does some form of keyword search and an Advanced Search with a more complex UI. The user might see the advanced search on the main screen or it might require an extra click to show it.

Depending on the UI, you may want to have the user select advanced search types from a list or other control. This brings up a few issues:

- How do you know what search types a repository supports?
- How do you present these types when searching across multiple repositories that may not support the same types?
- How do you convert the type the repository returns to something the user will find easy to understand?

We will look at these one at a time. Finding out what types a repository supports is easy. Call the `getRepository()` method on a repository. You will get a TypeIterator back. Get each Type from the iterator.

To get the repository in the first place, use the `RepositoryManager`. Note you can get all the repositories, all the repositories of a particular type, or a specific repository by id.

Federated searches are trickier. If every repository you want to include in the search supports the identical set of types, you have no problem. Assuming this is not the case, you need to apply some rule. Note that there may be one set of repositories that can participate in a federated search and a subset of repositories the user wants to include. The following suggested options assume you already have just that subset in hand.

- Show only distinct types – no duplicates
- Show all types and defer the issue of what to do if a repository does not support the type the user picks. Once the user picks a type, you could indicate which repositories will be included. Repositories that don’t support the type might be marked as excluded (grayed out, no check box, etc).
- Show all types and let providers handle unsupported types. Some providers will throw the UNKNOWN_TYPE exception when they are passed a search type they don’t know. Some providers might have configurable type mapping. This might let the provider, a system integrator, or the end user associate one type with alternates. If the provider gets a type it does not know, maybe there is a type it does know that is close enough for a search faithful to the consumer’s intent.
- Show only the types all repositories have in common.

Let’s discuss how to show the types. Types have authority, domain, keyword, and description strings. Here are some options:

- Show all strings in a multi-column table
- Show just the description
- Show just the keyword if it is unique, adding domain and authority only as needed
- Show the authority, domain, and keyword using some concatenation rule, e.g. domain/keyword@authority.
Displaying Search Criteria

Many user interfaces offer a single-line, plain-text field. You may want to offer alternate controls based on the search type. You might write these controls, but you might want to provide a way to install a custom control for some type you don’t know about right now. The search type OBA will include a specification for the search criteria format. Here are some things that might be specified and therefore affect the control, particularly if it offers validation.

- Delimiters
- Case-sensitivity
- Boolean operators
- Query expression standards (e.g. XQuery, CQL)

Displaying Search Results

Once the search is over, you will want to show the results. Here are some popular approaches:

- Table View – a table of results with columns for repository display name or icon, asset display name, asset description (possibly cropped), asset type or icon
- Lightbox or Preview / Thumbnail View – a small image for each asset and its display name and possibly a bit more.
- Citation or List View – a list with one entry for each asset. The list might contain the display name and description. You might also add the asset type.

There are some things to consider no matter the layout:

- It is important to distinguish between “the search is done and there were no results” and “the search is still running and therefore there are no results yet”. Of course you have many UI design options for this.
- You probably want to have a user gesture that will view the asset. Some assets will have a URL, so the viewer is a browser. For others, you would launch an appropriate viewer.
- You need to decide if you are going to clear the results of a previous search, let results accumulate, or let a gesture or preferences decide.
- If you are using a separate thread for each search or some searches take some time, you need to decide when to present results and how to show any late arrivals.
- The OSID is silent on the issue of the order of assets returned by a repository. Type OBAs can speak to this. This is important if you want to show prioritized or ranked results.
- If you want to save the results, you will want to keep the provider package, the repository id, and the asset id. All are strings. For example, you might want to e-mail a result to someone else. If the asset has a URL, you might just mail that URL. If you want to send a complete reference to the asset, include the three strings. Note that you could also send a form of the asset object. Be sensitive to digital rights issues. Since repositories are responsible for enforcing authorized access, sending the URL or asset id (and repository id and package name) may be the safest.

Displaying Assets

The kinds of assets that you can find in repositories vary widely. Here are just a few some examples:

- catalog records containing only metadata
- documents in various formats (e.g. plain text, RTF, PDF, HTML, Microsoft Word)

14 An Asset Type OBA may include icons the community can share.
• application files in various formats (e.g. spreadsheets, presentations)
• images in various formats
• audio in various forms
• video in various forms
• Flash™
• learning objects

Given all this variety, you may want to do something generic or specific. The generic route might pass the content to a browser or the operating system and let it make an effort to show the content or launch an application that can handle it. This solution should work pretty well and off-loads the responsibility for asset viewing to others—keeping the consumer application simple. If there is a content type the browser can’t handle, perhaps the user can download a plug-in for it.

A specific approach would have viewers for certain kinds of content and should include some extension mechanism so you can use a viewer someone else provides.  

**Displaying Metadata**

There are a variety of approaches you may want to consider here. Let’s start with the most general. Assets have Records, and Records have Parts. Parts can also have Parts. Both Records and Parts have structures. The following UI uses a horizontal “finder” control at the top. This control handles all possible containment relationships. The lower portion of the screen shows asset detail.

---

15 The application whose screen shots I include in the section on Displaying Metadata runs on Mac OS X and uses Cocoa. The application loads custom viewers, implemented as bundles and discovered at runtime.
Note the content control. This viewer might get the content of the asset and try and show it in the control. The control could be a mini-browser or some other player. If there is a Type OBA for the asset that the provider and consumer know about, showing the content faithfully is easier. Also the content can have its own record structure, which can have a type with an OBA.

![Figure 28: Displaying Metadata Generically -- Asset View](image)

Looking in more detail at this generic viewer, we see a display for each record. Note that there are not controls for every record or record structure attribute, but the idea is clear.

![Figure 29: Displaying Metadata Generically -- Record View](image)
Finally, a look at the part display.

![Figure 30: Displaying Metadata Generically -- Part View](image)

What we have just seen is not the only way to present the asset generically. A hierarchy is another valid approach. This is a place where the design of consumers varies and customers get choice. If you design and implement the viewers without building in any assumptions about provider content and implementations, the viewer should be of some value for any repository provider.

While a general solution is good, so is a tailored one. If an asset has metadata in a known schema, a viewer can offer a deeper integration. For example, if an asset has Dublin Core metadata, you could load a special viewer designed to handle Dublin Core. Not only can you tune the presentation to the schema, you can offer validated data entry and other features. Note that this viewer can still be generic enough to work across providers.

Getting even more tailored in presentation and treatment of metadata, a consumer can support custom asset viewers. These might be designed for a specific Type OBA and work with any provider or just work with one providers’ proprietary content. After all, there is no requirement that a provider use Types that have published OBAs. Here is an example of a “museum” view of an asset. The metadata schema is proprietary and certain fields (parts’ values) are arranged into a tombstone.
Displaying Thumbnails

Some repositories hold content for which there is a preview or thumbnail. There is no specific OSID method you call to get this\(^ {16}\). There is a part structure Type OBA for thumbnails. You can consult the O.K.I. project website. Assuming the provider supports the type, you would locate the Part in the Record for an Asset whose part structure has the Type for thumbnails. Depending on what record structure types an asset has, you might only look at certain records. Refer to *Accessing a Single, Specific Metadata Field* for an example of getting a Part with a given part structure type.

Displaying Exceptions

OSID providers can throw exceptions. You also need to handle exceptions that non-OSID method calls generate. There is limited value in showing a detailed error message to a user. You can log that information. This is the same philosophy the OSID providers use when they treat a consumer as if it were a user. They provide a general message that might allow the user to recover, if that is even possible, and log anything detailed.

Let’s take a concrete example. If you are logged in but don’t have authorization to perform some repository operation, the repository is going to throw the RepositoryException with the message string defined as `PERMISSION_DENIED` in the RepositoryException.\(^ {17}\) You may want to rephrase this in an alert for the user. Maybe you would suggest they have their authorization broadened or ask someone with broader permission to perform the operation. Another example is an unknown id. Maybe the user entered incorrect or obsolete information.

\(^ {16}\) There is a proposal in for version 3.0 of the OSIDs to include this.

\(^ {17}\) The current OSID jar (version 2.0) defines exception messages in US English.
Getting Assets

This material is covered under the Provider Guide, Getting Assets, so I am not going to repeat it here. I do want to comment on a special case of getting assets of several types at a time.

Getting Assets of Several Types

Let's say you want to return all the assets of Types A, B, and C. You have several choices:

- **Call `getAssets()`** to get all assets and check the type of each. This makes the most sense when the number of assets that are of Types A, B, or C represents almost all the assets.

- **Call `getAssetsByType()`** once for Type A, then for Type B, and then for Type C. Making three calls always seems less efficient than making one.

- **Call `getAssetsByType()`** with a new Type, D, that is defined to include Types A, B, and C. I don’t really like this solution. It seems to conflict with the idea of types.

- **Call `getAssetsBySearch()`** with a search type and criteria that express your intent. This might be a good solution. The community can define a search type for this.

- **Call `getAssetsByTypes()`**, a method defined by a provider that is not currently part of the OSID. Working outside the OSID is not encouraged, but can be done.

Representing a Reference to an Asset as XML

While an asset's id identifies an asset unambiguously, if you send someone the id's string representation, it may not be sufficient. The id will only be enough if the other person knows the package name for the jar that contains the class that implements the RepositoryManager. The RepositoryManager has a method, `getAsset()`, that returns an asset given an id. This assumes that the asset id, which must be unique for a repository, is also unique across the repositories under a manager. This should be the case. If a manager is going to support a new repository, the implementation should be alert to id conflicts. This could happen if a manager is supporting repositories developed by independent groups. They might have used the same id scheme. If this happened, you may not be able to change the ids after the fact. To be certain this circumstance is not going to come up, remote though it is, specify an asset using the repository id as well. For example, here is a piece of XML that you might adopt.

```xml
<manager package="edu.mit.osidmpl.repository.foo">
  <repository id = "010203">
    <asset id = "abcdef001"/>
  </repository>
</manager>
```

For more than one asset, you could use:

```xml
<manager package="edu.mit.osidmpl.repository.foo">
  <repository id = "010203">
    <asset id = "abcdef001"/>
    <asset id = "abcdef002"/>
  </repository>
</manager>
```
A Sample Repository Provider Inspector

Note the screen shots in this section are from a provider for the OSID version 1.0. At that time, the Repository OSID was called the DigitalRepository OSID, the convention for ordering Type strings was domain first instead of authority first, and the terms RecordStructure, PartStructure, Record, and Part were InfoStructure, InfoPart, InfoRecord, and InfoField, respectively.

Here we see a manager-level view.

![Figure 32: Results of Selecting All Repository Types and All Repositories](image)

Next, the Features pane contains 4 separate tables of data:

1. Information about the Asset Types one or more of the selected Repositories support.
2. Information about the RecordStructures they support. If one of the Asset Types is selected, mandatory RecordStructures include an asterisk prefix.
3. Status Types.
4. Search Types.

- Keyword matches search criteria with text in Assets’ name and description.
- Universal returns All assets in the repository. Any criteria are ignored.
- Media matches search criteria to Asset types in the Repository holding digital photos. Specifically, using “Image” as a criterion will find all the photos.
- DisplayName matches exactly criteria and Asset names.
The Search Pane reproduces the Search Types Table from the Features Pane. Selecting a specific Search Type, providing criteria, and pressing the Search button causes the application to call the Repository OSID implementation to ask for all Assets matching the search Type and criteria in the selected Repositories. When searching, you could any criteria as text. Different applications might use more elaborate criteria input mechanisms. For the Keyword search type, the text in the criteria control must match some part of either the name or description of a department or course Asset. For the Universal search type, this control’s content is ignored. For the Media search type, only “Image” is valid and will result in all photos being found. For the DisplayName search type, an exact match with an Asset’s display name is sought. If no search is selected, Universal is used.

\[18\] This is a case-sensitive match. Other and more elaborate ways of finding assets based on text could be supported.
Figure 34: Search For Images and Selection of “Rose” Asset

The Asset Pane provides information about a specific Asset. This includes the hierarchy of any Assets it contains, the Id, the display name, the Status, the Asset Type, the description, and the content of any Records. Each Record contains structured data about the Asset. Records are based on a RecordStructure. Each RecordStructure appears as a separate tab in a control at the bottom of the Asset Pane. Within the pane for that tab, all Parts and their content are presented. Images have Image and Intellectual Property (IP) RecordStructures.
Figure 35: Rose Asset and IP InfoStructure

The Viewer Pane shows the Asset’s content if any. In the case of Image Assets, this is the image itself.

Figure 36: Rose Asset’s Content

Searching again, this time using the Universal Search Type and selecting the Biology Department in the Courses Repository results in the following:
Updating Content / Formed-Based Data Entry

As a consumer, you may offer a data entry user interface. This assumes the provider you are using will allow updates. You can check this by calling the `supportsUpdates()` method.

Validation is another step you may want to address during data entry. This is discussed in more detail elsewhere. Here I want to mention that there is a `validateAsset()` method for a repository. Your data entry routine may want to call this. The status of an asset may change after this call.

1. Create a new asset (this includes setting the display name and description in the constructor)
2. Create Records
3. Create Parts (this includes setting the value in the constructor)
4. Update content
5. Validate

For existing assets
1. Update display name
2. Update description
3. Update parts
4. Update content
5. Validate

In either case, there is an optional step 0 and step 6, `mark()` and `commit()` respectively. This would allow repositories that support a “transaction” approach to know you are planning to make a series of modifications.
One option is to capture data for a new asset and when the user is ready to save, create an asset during the validation process and wind up with either a valid or invalid asset.

Figure 37: Validated Asset Creation

Figure 38: Data Entry With Asset Created During Validation
An alternate approach is to create the asset along with the UI for each asset's data entry. Here the asset is marked as invalid and is updated and then marked as valid if appropriate.

![Figure 39: Data Entry With Asset Created Along With Each New Form](image)

Figure 39: Data Entry With Asset Created Along With Each New Form
Provider’s Guide

This is the section for developers implementing OSIDs.

Suggested Package Naming Conventions

You are free to use whatever package name you want other than org.osid. I suggest you adopt increasing specialization, that you include osidimpl, that you include the OSID’s name, and that you allow for multiple implementations of any OSID by you. This means the form:

- edu.foostate.osidimpl.repository.xyz
- com.acmeplumbing.osidimpl.repository.abc

File Names and Interface Implementations

I usually name my files for the interface, such as RepositoryManager.java, RepositoryIterator.java, AssetIterator.java, and the other iterators. When I have multiple classes that implement the same interface, I put the interface name in the class name. For example XYZAsset.java ABCAsset.java. As a rule, I don’t implement more than one interface with the same class. An exception might be Repository and Hierarchy and Asset and Hierarchy Node.

Iterators

OSID methods that return multiple objects use Typed iterators. You will need to write these. They are quite simple. Iterators have two methods:

- hasNextObject() returns true if there is another object, false if there are no more to be had
- nextObject() returns the next object

For example, the iterator, org.osid.repository.RepositoryIterator has the two methods:

- public boolean hasNextRepository()
- public org.osid.repository.Repository nextRepository()

Here are two models for implementation. In the first, you have all the objects you are going to return and you pass these into a protected constructor. I am using a vector in this example.

```java
public class RepositoryIterator
    implements org.osid.repository.RepositoryIterator {
    private java.util.Vector vector = new java.util.Vector();
    private int i = 0;

    protected RepositoryIterator(java.util.Vector vector)
        throws org.osid.repository.RepositoryException {
        this.vector = vector;
    }

    public boolean hasNextRepository()
        throws org.osid.repository.RepositoryException {
        return (i < vector.size());
    }

    public org.osid.repository.Repository nextRepository()
        throws org.osid.repository.RepositoryException {
        if (i >= vector.size()) {
            throw new org.osid.repository.RepositoryException(
                org.osid.shared.SharedException.NO_MORE_ITERATOR_ELEMENTS);
        }
        ++i;
        return (new org.osid.repository.Repository(vector,i));
    }
}
```
Another approach is to use a private iterator variable, set that iterator from the input vector, and use the iterator’s
methods to implement the next and hasNext methods.

```java
public class RepositoryIterator implements org.osid.repository.RepositoryIterator {
    java.util.Iterator mIterator = null;
    protected RepositoryIterator(java.util.Vector vector)
    throws org.osid.repository.RepositoryException {
        mIterator = vector.iterator();
    }
    public boolean hasNextRepository()
    throws org.osid.repository.RepositoryException {
        return mIterator.hasNext();
    }
    public org.osid.repository.Repository nextRepository()
    throws org.osid.repository.RepositoryException {
        try {
            return (org.osid.repository.Repository) mIterator.next();
        } catch (java.util.NoSuchElementException e) {
            throw new org.osid.repository.RepositoryException(
                org.osid.shared.SharedException.NO_MORE_ITERATOR_ELEMENTS);
        }
    }
}
```

In a second approach, you are going to go through some process to decide if there is another object, to get the
next object to return to the caller, or both.

```java
public class RepositoryIterator
implements org.osid.repository.RepositoryIterator {
    public boolean hasNextRepository()
    throws org.osid.repository.RepositoryException {
        boolean more = someProcess();
        return more;
    }
    public org.osid.repository.Repository nextRepository()
    throws org.osid.repository.RepositoryException {
        return anotherProcess();
    }
}
```

The iterator implementations should reside in the same package as the other classes for the OSID. To
understand why, let’s discuss the implementations for the iterators defined in the Shared OSID. For example, you
return Types using a TypeIterator. This is defined in the Shared OSID, but you should implement it in the
Repository OSID. I am suggesting that you not implement the Shared OSID. Although it is a little more work, you
will be able to offer each of your OSID implementations as a stand-alone product if you implement the Shared
OSID iterators in each OSID that returns them. Let me say this in another way.

If you have an implementation of the Repository OSID, it needs to include a TypeIterator. If you have an
implementation of the Authorization OSID, it also needs a TypeIterator. Rather than implement this iterator once
in a utility package, so that it can be used by both the Repository and Authorization implementations, I suggest you implement the iterator twice, once for each OSID.

Figure 40: Recommendation That Each OSID Implement TypeIterator Separately

The justification for this extra effort is that a consumer can swap each of your OSIDs more easily if you reduce or eliminate inter-OSID-implementation dependencies.

Types

Instantiating and Comparing Types

You will want to return Types to the consumer. You can define a “generic” subclass of `org.osid.shared.Type` that accepts arguments or a “specific” subclass that only instantiates Types with a fixed set of string values. In either case, you will probably want to declare this classes constructor protected or private.

Use the `isEqualTo()` Type method to compare Types. This method tests for exactly equal authority, domain, and keyword strings. The test ignores the description.

Publishing You Own Types

The O.K.I. community is developing Type-related agreements (OBA). These spell out what a Type means. Try and use one for the Types that already exist rather than making your own. As a community, we want to keep the number of Types under control. Too many Types and we loose interoperability.

At the time I wrote this, the rules for Type strings were still fluid. What is emerging is:

| authority       | Use period-separated institution or organization names with increasing specificity. For example "edu.mit". |
- **domain**
  Use the name the OSID has for the method or object related to this Type. For example, if you want to define a new type of repository, the domain should be “repository”. Try and be as short as possible without causing ambiguity. Use camelCase for compound, for example, “partStructure”.

- **Keyword**
  Use something short. Use camelCase.

- **Description**
  Use whatever you want. For now, try and be short in case a consumer is going to display this in a list.

### Deciding What Exception To Throw and With Which Message

The JavaDoc for RepositoryException has the complete list of the exception messages you can use.

The JavaDoc for each method has the subset of exception messages you are likely to use. It is not wrong to use a message that is defined for the RepositoryException but not mentioned in the JavaDoc for the method. The JavaDoc writer might not have thought of a case involved in your implementation. It is not good form to introduce a message that is not in the OSID. Doing so undercuts interoperability since a consumer doesn’t know what to do in response to the message. The consumer might assume a worse or best case.

Only throw the one OSID exception in any of your methods. Handle any other exception that might be raised by any method you call. Don’t let those exceptions go unhandled. You can log any other information you think is relevant for a development or production version. For example, you might want to log a stack trace.

### Describing Repositories

When you implement the repository, you will need a display name, etc. Here are some comments:

- The display name should be distinctive and short.
- You don’t have to worry about guaranteeing the name is unique, the repository id is for unambiguous identification.
- Something descriptive and short enough to fit in a likely list control is a good idea.
- The description method returns something wordy.

Repositories have a type. Consult the communities Type OBAs to try and find a good match before suggesting your own. The Type should not focus on the underlying technology, e.g. Postgres is not a Type. Better types are multimedia, library catalog, etc. Note the type may include conventions about how to provide an icon, URL, etc.

There are no firm guidelines on how to map underlying content to repositories. For some systems, one repository might be right, for others, many. I think the level of aggregation should tie back to the breadth of searches. This might involve subject or media. For example, if you have a collection of old photos and a collection of press releases, these probably belong in separate repositories. This is really a judgment call on your end.

---

19 Other bindings use different models. For example, in Objective-C, there are reasons not messages. Also, rather than use #defines, we use externs and global variables.
In addition to the name, description, and type, you can also tell a lot about a repository by the type of assets it holds and the mandatory and optional record and part structures. As a provider, if you are going to use asset types – which I recommend strongly, you should return them through the `getAssetTypes()` method implementation.

Most repositories will include assets with metadata. If your assets have records, you should implement the `getRecordStructures()` and `getMandatoryRecordStructures(assetType)` methods. Your
implementation of `getRecordStructures()` should return the union of all record structures for all asset types. Only return distinct values – no duplicates.

Your implementation of `getMandatoryRecordStructures(assetType)` should return only the mandatory record structures for assets with the specified type.

![Venn Diagram of Record Structures, Mandatory and Otherwise](image)

Consumers can use this information to adjust the user interface or perform some data validation for completeness. Note that you should be consistent in the information you report in the mandatory method and checks you perform in the `validateAsset()` method.

**What Kinds of Content Stores Can Be Hidden Under The Repository OSID**

When you hear the word “repository” you probably have certain underlying content management systems in mind. Common ones are:

- database
- library
- catalog

We have explored a wider range in our thinking about the Repository OSID. Here are some examples:

- Mac OS X iTunes Library (also other iLife content)
- Mac OS X Spotlight
- IMS Content Packages
- XML Files
- Data Feeds
- Local File System

The point is that the Repository OSID can cover many kinds of information. What is in common is that there are discrete elements, elements have names and other descriptive data, and there is some form of enumeration or searching.
Repository OSID And Databases That Support SQL

You can implement the Repository OSID atop a database in a relatively straightforward way. The entire database might map to the RepositoryManager. The assignConfiguration() implementation would set up the JDBC driver, etc. Each table might be a repository, or several tables might be used. Each row might be an Asset with columns for name, description, id, type and so on. Other columns might hold metadata. “get” method implementations are SELECT statements and they return data gather from earlier statements. “update” method implementation might be INSERT or UPDATE statements. “create” and “add” methods are INSERTs, “delete” and “remove” methods are DELETEs.

Of course this is a simplification. In a production system there might be many tables, clever, complex, and efficient statements, and so on.

Databases offer user authorization by operation by table. You could elect to setup authorizations and let the database enforce them. Your Repository implementation would just submit the statement and handle the databases denial by throwing a PERMISSION_DENIED exception. Alternatively, you could skip setting up detailed authorizations for the database and call an Authorization OSID implementation instead. Note that another kind of exception a database might return relates to record locking. Probably, PERMISSION_DENIED is appropriate here too.

Repository OSID and Content Management Systems With APIs

You might have an even easier time implementing the Repository OSID atop a content management system’s API. Again the RepositoryManager assignConfiguration() is a good place to do initialization. Probably some of the OSID methods map directly to API calls while others map in a simple manner to one or more calls.
The content management system probably enforces authorization internally and that may be a good place to leave it. We would encourage the content management system vendor to consider implementing the Authorization OSID internally—as well as implementing the Repository OSID. For the vendor, a separate Authorization OSID service allows system integrators to deploy the vendor's system and a local enterprise authentication and authorization service. Some customers may prefer this flexibility.

Let me start by clarify what I am covering under this title. There are a range of offerings one might call Web Services. I am talking generally about functionality accessible using a standard Internet communication protocol like HTTP and data formats such as XML that are commonly used for web applications. One popular way to transfer information to a web application is SOAP, a protocol for transmitting messages formatted in XML and typically sent using HTTP. Pages formatted with HTML can be returned by a web server that handles HTTP requests. Specific websites and files on the Internet are identified by URL. The comments that follow apply to provider implementations that employ these standards and techniques.

Among a database, a content management system API, and a web service, the service probably is the loosest fit to the Repository OSID. Unlike the API, there are not going to be methods that map to the OSID methods. Unlike the database, the requests for data and operations are not as precise. On the plus side, a web service probably is going to return "asset" metadata as XML. XML, a hierarchical, tagged format, maps nicely to the asset / record / part model in the OSID.

Figure 45: Repository OSID Provider Covering Content Management System API

Repository OSID And Web Services
Your provider implementation can, as usual, setup in the assignConfiguration() method of the manager. You might represent the service as a single repository returned by the manager. The implementation of the asset search method can translate the criteria into an HTTP request or similar. Note that you could “expose” the web services query format in a published Type. To allow for the case where providers change, you probably want to support some of the existing search types and add a prefix, suffix, and any other fixing-up before calling the web service.

The result is probably XML. Things are harder for you if you get HTML back. In that case you would have to “screen scrape”, by which I mean parse the HTML to get the unformatted data. Ask the web service vendor if they can provide an XML response. Assuming you have XML, you can parse it and construct Assets, Records, and Parts from the content or just refer to the XML in the Asset, Record, and Part implementations.

Some developers have found a use for the OsidContext. Specifically, you might expect things like a session id to be there.

Searching

Let me step back just a little. The intent of the Repository OSID is to support a variety of applications. As background, in a traditional library, there are physical objects such as books. Separate from each book, is a catalog card that contains descriptive information about the book. This descriptive information is comprised of fields in a regular format, for example the book’s title, author, publisher, and so on. This information can act as a placeholder for the book and one can search for books of particular interest by searching the information fields’ content in various ways. Libraries hold resources other than books, for example periodicals, manuscripts, recordings, photo and video, etc. In general, each media type has its own kind of catalog card and there may also

---

20 You might store host, port, and login data in a properties file or elsewhere.
be subsets or collections of media covering a particular area or from a particular source, for example rare books or scientific periodicals.

One may want to search for material of a particular media type, from a particular collection, across collections, or on a subject. For example, if one wanted material on a famous person, one might want books, newspaper articles, photos, video, and more. Currently, some catalog and content are in electronic form. Not only will one want to organize and search material in more than one Repository, one will want to do the same across different institutions' Repositories. The Repository OSID is intended to catalog, search, and organize assets in this variety of applications.

Search Tools

Searching can take many forms and use many techniques. A few general patterns emerge for an examination of the way searches are performed:

**One-time, one-pass search** – In this mode, you are looking for information once. You may be looking for something very specific. You may not have a lot of time to invest in an exhaustive search. A simple example is looking for someone in a local directory. You know exactly what you want, say their location or phone number. You expect that they are either in the directory or not. If they are, you want their information, if not, you will use some other mechanism to find the information. The same directory may be searched often, but always for a single piece of data.

**One-time, refined search** – In this mode, you are looking for information, but there is some ambiguity or variety, either in the sources to be searched or in the information returned. For example, you are looking for an article on some subject. It may be a specific article or any article that meets your needs. You may know the author, title, subject, or approximate date of publication, but not all of these things. You expect that the article is in one of a set of journals, but again you don’t know which one. The first time you might cast a wide net by searching several repositories of articles and you search several times varying the values of various search fields until you find what you want. You might make additional searches, but only from among the set of results returned by some wide search. You might also want to narrow the list of articles to be searched before performing a more exhaustive kind of search, such as looking at the full text of articles rather than just their descriptive fields.

**Recurring search** – In this mode, you are looking for the same kind of information often. For example, you want to keep abreast of developments in a particular field. You may make a search, say once a week, for anything new on a given topic. You might be interested in searching a variety of media type such as books, journals, news articles, television broadcast archives, etc. You want to save your search so that it can be re-executed easily. Such a saved search can itself become an asset in a repository. Not only could you be interested in saving the search, but there might also be search-related preferences you want to preserve. Examples include the order in which you search repositories, whether or not you stop searching after some expenditure or time or number or results are found, how results are presented, and so on. An application can assist in defining searches, managing preferences, and performing searches periodically and notifying you of results.

Asynchronous Searches

You are free to use synchronous or asynchronous back-end content sources. For example, some federated search gateways, brokers, or peer-to-peer services send a search off to many targets and wait for responses to come back. Since the OSIDs use simple method calls and do not have callbacks or event or message listeners, this brings up some interesting issues:

- How long should a consumer wait for a response from a method?
- How does the consumer indicate how many responses should trigger a return of assets?
- What happens if additional responses come in after the search method returns assets?

The OSID is silent on the issue of response times for method calls. Naturally, you don’t want to keep the consumer waiting too long. The consumer should worry about issues such as calling the search method using a separate thread, and when to block application progress while waiting for a method to return. Here are some ideas:
• A search type OBA can comment on timeouts that providers should implement and consumers should specify. A timeout could be passed in through the criteria, less interoperably through search properties, or configured for the repository in some other way.

• The provider could have a configurable timeout, but not offer access to the setting through the OSID. A system integrator or administrator could set this.

• The consumer could set up its own timeout mechanism.

We have a similar issue and set of options with how many responses is enough to trigger the provider to return. Note that some search types will allow the consumer to specify the maximum number of responses they want.

The issue of late-arriving responses resulted in a community search type. The “more” search type OBA states that when a consumer uses it, the provider should return whatever is available since the last search. As a provider, you assume the criteria and properties are the same and you return whatever has accumulated since the previous search. This is a “poll” operation.

A final note. You may want to offer progress information to a consumer while the search is underway. You could put this information in the OsidContext and specify the details in an OBA.

Versioning

The OSIDs are designed with system evolution in mind. One of the principal benefits of the OSID is that it is easier to manage changes in systems built with this architecture than with other approaches. Once you change something, it can be said to have more than one version. If you discard permanently the earlier version, there is little version management to worry about.

![Changes Overwrite](image)

![Change Makes New Asset](image)

**Figure 47: No Versioning - Changes Overwrite or Changes Spawn New Asset**

If you need to track changes across versions and optionally return to an earlier version, you obviously need a version management strategy. Computer systems have evolved numerous methods to address this area.
including version control for software development, support for transaction management, archiving each version, and so on. The details of the implementation of any of these strategies, as with all OSID implementations, are intended to be contained wholly within the implementation and unknown to the user of the implementation.

Let’s look at how the Repository and Assessment OSIDs could be supported by a version management strategy. Repositories are designed to catalog, store, and retrieve things (assets). Each asset has a name and description, but these are not guaranteed to be unique. Since we want to identify assets unambiguously, each gets a unique identifier provided by the Repository that creates the asset. Your provider can go one step further and include a creation timestamp. When an asset changes, we might create a new asset version with a timestamp. Note that whether this process really makes a new asset with the same id and a different timestamp or uses some other approach, from the perspective of a consumer, assets can now be further specified by date.

![Figure 48: Each Change Spawns a New Asset Version](image)

We designed the Repository OSID so some version can be implemented. Specifically, a consumer can retrieve an Asset by id, in which case you return either the only Asset if there is only one, or the version of the Asset that you choose – the most recent seems reasonable. An Asset can also be retrieved by id and date (timestamp). This gives you some additional information to work with in its effort to return the right version. The date might be when an Asset was changed, or you might return the Asset whose changes happened most recently after the date. The particulars are, as always, up to you.
Some of the OSID implementations may be built atop a Repository implementation. Good candidates are ones concerned with hierarchical data and information about data. One candidate is Assessment. Assessments are divided into Sections and Items and each contains both data and information about that data. Another candidate is CourseManagement which contain CanonicalCourses, CourseOfferings, and CourseSections. This too follows the pattern of hierarchical data and information about data. Note that DigitalRepository itself might use an OSID Hierarchy implementation. Clearly, modularity, reuse, and interplay are intended.

Repository, Authorization, and Hierarchy

In this section I am going to discuss the interplay of three OSIDs: Repository, Authorization, and Hierarchy. In summary, repositories manage and provide access to content. Repositories typically enforce some access control over content and the Authorization OSID has methods for asking the appropriate questions. Now let’s look at more detail.

The Repository OSID uses containment widely. RepositoryManagers have Repositories, Repositories have Assets, Assets have Assets, and Assets have metadata. These containment relationships can be modeled with a hierarchy. Here is a look at what contains what:
Many repositories require authenticated and authorized access. Even if you do not need this, you probably want to insert authorization calls into your repository implementation. They can be NOPs, trivial calls that return true, or calls to an AuthorizationManager that tolerates everything\textsuperscript{21}. Adding the calls is not much work and saves time later if you need to introduce authorization.

Authorization involves an agent (who), a function (what operation), and a qualifier (what is acted upon). Agents are people, groups, or processes. There is an Agent OSID to manage these. Functions should focus on actions, decoupled from what will be affected. Some suggestions for simple functions are: \textbf{view}, which you would use in the authorization check at the start of most “get” methods and \textbf{maintain}, which you would use for all the create, delete, add, remove, and update methods. I say most for “get” methods because, by convention, there should be no authorization required to call methods of the form “getXXXTypes()”. Authorization also can involve a hierarchy of qualifiers.

Since the number of qualifiers may be quite large, you may not want an authorization that references each qualifier directly. It can be convenient to ask about both \textit{explicit authorization} and \textit{implicit authorization}.

- Explicit Question: “Is the user allowed to perform some function on some object?”
- Implicit Question: “Is the user allowed to perform some function on some object or on one of its parents in a hierarchy of objects?”

There is no requirement that you support implicit authorizations, but you can.

\textbf{Note that I have not mentioned authentication.} There is an Authentication OSID that addresses this area. One common approach is for the repository to only ask authorization questions and leave it to authorization to ask authentication questions.

Assets can contain assets. There is no requirement that your repository have nested or hierarchical assets but you can. For example, if you want to grant “Fred” read access to the 10,000 assets in a collection in a repository,

\textsuperscript{21} One of these should be available on the O.K.I. SourceForge site.
you may want to use a single authorization at the collection level rather than maintain authorizations for each asset.

Let’s look at one implementation scenario. We have a content management system that holds some data. Maybe this is a database that we access with a driver and that supports SQL queries. There is a Repository OSID implementation that uses SQL and the driver to expose the content. There is some asset containment. Let’s say the content maintains collections of images. There are collection assets and image assets. All image assets are under some collection asset. This asset containment logic is reflected in the OSID implementation by calls to the database where the containment is represented. The repository enforces access control based on responses from an Authorization OSID implementation. That implementation supports implicit authorizations, again by making queries to the database to determine containment. A user-facing application consumes the repository provider and also contains a UI and logic for managing collections. This might be done by calls to the repository or directly to the database.

![Figure 51: Interplay of Repository and Authorization with Hierarchy Logic](image)

This model is viable and uses OSIDs. Let’s consider some of the virtues and limitations of this approach:

- The application, repository, and authorization code has knowledge of the database and can be very efficient and precise in its operations.
- This is a bundled deployment. Even if the application manages the collections solely via repository calls, the authorization uses the containment information for the repository directly from the database. You can’t easily swap a new authorization implementation unless it also knows about the database.

It might be worth the development effort to introduce a Hierarchy OSID implementation in this model. Here is what that would look like:
Here are the benefits over the earlier approach:

- The repository provider can use a hierarchy to manage containment. This reduces the complexity of the repository implementation and allows us to introduce better hierarchy providers without disruption. The repository retains special knowledge of the database, which is proper.

- The authorization provider can use the same hierarchy provider to walk the containment during implicit authorization checks. Again, the authorization implementation is easier.

- The authorization and repository providers are decoupled further. We can have a new authorization implementation that works with any hierarchy. The link to the repository’s containment information in the database is broken.

- The application is smaller since hierarchy management can now be handled by tools dedicated to this purpose. You could have a tool that just allows the user to rearrange hierarchies. This tool might be nicer than anything you would put into an application whose central purpose was something else. A tool developer could have their code work with any Hierarchy OSID provider. Smaller and simpler applications are easier to manage.

Adapters

An OSID provider focuses on supporting the functions covered by the OSID. What if you want the implementation to include behavior relating to a different OSID, but don’t want to bind those behaviors tightly? In those cases, an adapter may be a useful architectural mechanism.
An adapter is an implementation of an OSID whose methods call methods with the same name in another implementation of the same OSID.

For example, say you want to create a group containing the roster in a course. This can be a convenient way to handle authorizations for a course. The CourseManagement OSID manages the roster while the Agent OSID manages the group. Creating a group is a side-effect from the principal function of a CourseManagement OSID implementation.

An adapter is an implementation where each method is either:

- a pass-through that simply calls another implementation of the OSID that does the real work; or
- performs some desired side-effect behavior.

Since the adapter and the principal implementation it calls have the same interface, that is, they implement the same OSID, an application calls the same method whether the implementation is the adapter or the principal. In this manner, desirable side-effect behavior (in our case creating a group from a roster) is factored into and out of solutions easily.

Let's back up a moment. One of the goals of O.K.I is to foster reuse among infrastructure and application services in educational software solutions. OSIDs are abstractions of functionality in a particular educational or infrastructural domain such as Course Management, Repository, Authorization, Authentication, etc. Many applications across different institutions will require these same services and reusing something that has already been developed may be more effective than building one’s own service.

Let’s explore several architectural models and discuss their merits and shortcomings with regard to reuse. Let’s further consider a specific scenario in which we want to develop a Course Management provider that automatically creates authorizations to match any course offering or course section roster. CourseManagement covers both a structured catalog of course, offerings, and sections and student enrollment rosters for offerings and sections. You will probably want to grant specific authorizations to all the students in a given roster. One option is to make an explicit authorization for each student in the roster. One option is to make an explicit authorization for each student in the roster. Of course, if the student adds or drops the course, you will want to reflect this in the authorizations. Another option is to make a group from each roster. A group allows you to perform authorizations in one step rather than for each student. Also, since the authorization is based on the current group, if a student is added or dropped, there is no need to add or drop the student individually from the authorization system. A group can also be handy for notifying students through user messaging and elsewhere.

Managing authorizations relating to rosters is outside the principal flow of CourseManagement and is a side-effect of implementing the enrollment methods. The authorization behavior can be added to a CourseManagement provider directly. The only liability is that the provider is less portable than a provider with no side-effects. If one environment wants the authorization and group management behaviors and another environment does not, the same implementation may not be appropriate. A trivial solution is to perform these side-effects based on an implementation configuration setting. A more involved, but also more robust solution, is an adapter.
An initial application architecture might expect to use an CourseManagement OSID ("A") and an Agent OSID ("B") provider separately to both manage rosters and create groups from rosters:

Figure 53: OSID Implementations Provide Their Own Services Only

This model has the virtue that each provider covers only the area of concern for that OSID and therefore can be reused or substituted easily. There are no side-effects to worry about. A deficiency with this approach is that the consumer has to manage two OSIDs and synchronize rosters and groups. This builds extra intelligence into the consumer rather than the provider. Recall that it is a design goal within O.K.I. to make application writing easy and support application reuse across implementation infrastructure. This may come at the expense of building extra intelligence into providers. An alternate approach is to produce providers that have additional value beyond the functions in a particular OSID. In other words, these implementations do what you ask and more.
The advantage of this approach is that the application calling the principal provider (in our case Course Management) does not have to worry about the separate issue of group management. The CourseManagement implementation does this as a benefit. The liability with this approach is that this additional behavior relating to groups is now hidden from the application; that can lead to integration issues, particularly when substituting this implementation for another that does not provide the value-added group management.

If you want the best of both models, you can use an adapter. The adapter is a special kind of OSID implementation. In general, most of the methods will simply pass along their arguments to the same-named method in another implementation of the same OSID. In this mode, the adapter is a passive pass-through. For methods that should have extra behavior, the adapter implementation does whatever is appropriate (in our example, this is managing a group through the Agent OSID) and then calls the principal implementation for the OSID. The benefits with this architecture are that you:

- preserve highly-reusable implementations that deal with one OSID only
- can add (or subtract) extra functionality simply by calling the adapter implementation rather than the principal implementation

Since the interface of the adapter and principal implementation is the same, the application does not have to change – only the call to load the implementation does.\(^{22}\)

---

\(^{22}\) A detailed discussion of implementation loading and substitution mechanics lies outside the scope of this paper.
It is worth noting that the adapter does not have to stand in front of the principal implementation. A valid approach is to call the adapter to do post-processing of a method, or nothing. This may not be as desirable a model since the principal implementation needs to always call an adapter and that is not standard behavior in O.K.I. Anything that is not standard makes reuse harder.

This same model of optional behavior could be handled by configuration in the principal implementation. It is a simple solution, but it requires knowledge of the second OSID to be embedded in the first and this increases the maintenance overhead and may be an integration issue as OSID versions change.

In summary, for certain applications, an adapter may be an effective way to compartmentalize value-added or side-effect behaviors. The approach keeps principal OSID implementations focused solely on their service and thereby maximizes reuse and streamlines maintenance.
Adapter as Multiplexer

We have taken a look at one pattern for adapters. Another is a multiplexer. A simple example is federated search. You could use an adapter to cover several repository providers under an umbrella.

![Figure 56: Using an Adapter to Federate Repository OSID Providers](image)

Updating Content / Formed-Based Data Entry

While many repository implementations will be read-only, some will support write functions. In OSIDs, you use the “update” methods for this. There are dedicated methods to update an asset’s display name (title) and description. There is also an update method for the content. For metadata, there are `create` and `update` methods. Let’s discuss a few issues:

- Authorization to change content
- Implementing the `validateAsset()`, `invalidateAsset()`, and `getStatus()` methods
- Update performance

Often users need a broader level of authorization to update content than to read it. Even if you are not using authorization yet, it might be prudent to insert a call to an authorization method when you implement update methods. The call can be to an authorization manager that always returns true for authorization methods. Alternatively, you could call a local method that returns true. The authorization to read content (the “get” methods) might have a `view` function in the triple: agent, function, qualifier. The qualifier is the asset (or Part) id. Authorization to write content might have a `maintain` function. This probably is the same for the “create”, “delete”, “add”, and “remove” methods.

It is good form to implement the `supportsUpdate()` method, particularly if you do. Your implementation that allows updates could always return true or you could base the return on the user's authorizations. A consumer may use this method to adjust the user interface.

The OSID offers two places to handle validation. The first is simply to throw an exception in the update method implementation. You may want to throw `NULL_ARGUMENT` when null is not valid content and `OPERATION_FAILED` for other cases. You can be more specific in a log file. For example, let’s say you have a

---

23 This will be a NOP or at least very fast, so the performance impact is negligible or zero. Also note that some authorization implementations load all the user's authorizations in memory at login and so authorization checks are very fast.
field that has to be a date. The consumer may provide something that cannot be converted to a date. The OSID does not offer an exception for this kind of format problem, in part because there can be so many variations. The Part value that the consumer is trying to update should have a PartStructure Type and the consumer should be able to use that to offer more helpful validation feedback. Alternatively, the consumer could build in data validation before the OSID is called.

The second place to handle validation is in the Repository's `validateAsset()` method. This validates an entire asset and returns a Boolean. You should set the asset's status in the implementation. You return the status in the `getStatus()` method. Status is a Type. If the consumer calls `invalidateAsset()` you are supposed to relax your record and part level validation, but also set the status to a type that means the consumer needs to validate later. This approach could let the consumer play around with content without having to handle exceptions for invalid data they were going to fix later.

Note there are methods to validate a record and part as well.

If you support invalid assets, you might be able to offer better performance during data entry. Another option is to implement the `OsidTransactionManager`. Tell your consumers to call `mark()` before and `commit()` after their changes to an asset.

### Legacy Migration

This topic is covered in more detail in "OSID Fundamentals". In brief, the OSIDs can help with legacy migration. This process can also help re-factor a monolithic application into a modular one. There is really just good modular design. Here are the steps:

1. Assume we start with a legacy, monolithic application with some functionality for repository access.
2. Introduce the Repository OSID so we have consumer and provider code inside the application.

3. Move the OSID implementation outside the system. This allows you to tease consumer and provider apart. Note we also reduce the complexity of the application.

4. We might now change the provider to work with a new data source.

5. We might now change the consumer. Note that now consumer and provider are free to evolve more independently than in the monolithic application.

Some Implementation Comments

Manager method implementations delegating to Repository implementations

The RepositoryManager supports zero or more repositories. Since zero is not interesting, let’s say one or more. There are a few methods in the Manager that also appear in the Repository. Here is the list:

- copyAsset
- getAsset
- getAssets
- getAssetDates
- getAssetsByDate
- getAssetsBySearch

The Intent of the copyAsset() Method

There is no guarantee that you should be able to copy assets across repositories. To copy assets within a repository seems reasonable, assuming the assets are the same or compatible types. Whether or not you can copy from one repository to another has a lot to do with asset types, your permissions, which may be on a per-repository basis, and the available of create and update support in the target repository.

Your provider implementation for Repository may allow asset copying. If so, you should copy the asset entirely. You may have a non-OSID way of doing this using underlying system services. Alternatively, if you have update and create method support, you can use OSID calls to make the copy:

```java
org.osid.repository.Asset sourceAsset = repository.getAsset(sourceAssetId);
org.osid.repository.Asset targetAsset =
    createAsset(sourceAsset.getDisplayName(), sourceAsset.getDescription(), sourceAsset.getType());
targetAsset.updateContent(sourceAsset.getContent());
org.osid.repository.RecordIterator recordIterator = source.getRecords();
while (recordIterator.hasNextRecord()) {
    org.osid.repository.Record sourceRecord = recordIterator.nextRecord();
    org.osid.repository.Record targetRecord =
        targetAsset.createRecord(sourceRecord.getRecordStructure().getId());
targetRecord.updateDisplayName(sourceRecord.getDisplayName());
    org.osid.repository.PartIterator partIterator = sourceRecord.getParts();
    while (partIterator.hasNextPart()) {
        org.osid.repository.Part sourcePart = partIterator.nextPart();
        org.osid.repository.Part targetPart = targetRecord.createPart(
            sourcePart.getPartStructure().getId(), sourcePart.getValue());
targetPart.updateDisplayName(sourcePart.getDisplayName());
    }
}
```
Note that a more complete implementation would handle the case of parts with parts using recursion. Of course, you would also add exception handling. Whether or not it makes any sense to copy things this way depends on the underlying system.

**Getting Assets**

Asset ids should be unique. If the ids are globally unique, there will be no case where the asset id is the same in two repositories. Hence globally unique ids are best. Having an id that is unique across all the repositories in a manager is good, but there could be issues when you add new repositories. Let's put this issue aside. If the manager can resolve the `getAsset()` call immediately, fine. This assumes the manager knows how all the repositories manage their assets. A more common implementation is to call the `getAsset()` method for each repository until some repository returns an asset. Note you will want to catch and swallow any exception (with the message `org.osid.shared.SharedException.UNKNOWN_ID`) when a repository does not find the asset — after all, all repositories except for one should throw the exception.

Calling `getAssets()` is like saying, as one developer told me, “pour out the ocean over here”. The point is that the number of assets can be large and some repositories won’t support this method. There are cases where this method makes perfect sense. Since the return is an iterator, there is no reason why the consumer may not want to browse all assets. The intent of this method is that you aggregate all the assets from all the repositories. Recall that there is no guarantee of any particular order of assets in a repository or of repositories in a manager.

The provider is going to return an asset iterator. There are several ways you can join these iterators together. Two approaches are, (1) to create a special asset iterator that takes a list of asset iterators in a constructor and responds to calls for `hasNextAsset()` and `nextAsset()` by exhausting each asset iterator in turn, or (2) gathering all the assets from each iterator and returning that aggregation. There are some flaws in the second approach. One is that you have to wait to get all the assets, even though you may never be asked for them. Another issue is size.

There is a form of version support available with asset dates. As with `getAsset()`, this method, called on the manager or repository, addresses a single asset. The same applies to `getAssetsByDate()`.

The `getAssetsBySearch()` method for the manager is federated searching. The issues that relate to federating across repositories in different managers may apply to federating across all repositories in a manager. The intent is that assets are aggregated.

**Aggregating AssetIterators**

Here is some sample code that might be helpful:

```java
public class AssetIterator implements org.osid.repository.AssetIterator {

    private java.util.Vector assetIteratorVector = new java.util.Vector();
    private org.osid.repository.AssetIterator currentAssetIterator = null;
    private int numIterators = 0;
    private int iteratorIndex = 0;

    protected AssetIterator() {
    }

    protected void addAssetIterator(org.osid.repository.AssetIterator assetIterator) {
        this.assetIteratorVector.addElement(assetIterator);
        numIterators++;
    }

    public boolean hasNextAsset() {
    }
}
```

24 For example, Search and Retrieve Web (SRW), which uses the Common Query Language (CQL) to express queries, requires a maximum number of records to return.
throws org.osid.repository.RepositoryException {
    if (this.currentAssetIterator == null) {
        this.currentAssetIterator =
            (org.osid.repository.AssetIterator)assetIteratorVector.firstElement();
    }

    if (this.currentAssetIterator.hasNextAsset()) {
        return true;
    }
    else {
        while (this.iteratorIndex < this.numIterators) {
            this.currentAssetIterator = (org.osid.repository.AssetIterator)
                assetIteratorVector.elementAt(++iteratorIndex);
            if (this.currentAssetIterator.hasNextAsset()) {
                return true;
            }
        }
        return false;
    }
}

public org.osid.repository.Asset nextAsset()
throws org.osid.repository.RepositoryException {
    return currentAssetIterator.nextAsset();
}

Just-in Time vs Prepared Objects

When a consumer calls your provider to ask for an asset, your provider probably calls some underlying system. You are going to construct an asset out of what the system returns to you and you return the asset to the consumer. One issue is when, if ever, you go back to the system, in response to calls to the asset. You might fetch everything you need the first time, you might fetch nothing, you might fetch some and later more. You may handle content and metadata differently in this regard. You may handle asset attributes such as display name and description differently from metadata (records and parts). I am trying to lay out the issue, not tell you what you should do.
Part.isPopulatedByRepository()

The intent of this method is to tell the consumer that the provider is going to populate this field. A simple example is a creation timestamp. The consumer might adjust the UI to mark this as read-only.
Asset.getContentRecordStructure()

The asset content is an object. While the asset type’s OBA should say something about what the content is, a provider can implement this method to return a RecordStructure. There can be a lot of information in the structure that can help the consumer adjust, programmatically, to the kind of content coming back.

Effective and Expiration Dates

Some repositories will have assets with both effective and expiration dates, some with one, and some repositories will not support these methods. As a convention, you might return an effective date that matches the timestamp when you fetched the asset or you could pick the date the asset was added to the repository. Returning something is probably more helpful than throwing an UNIMPLEMENTED exception. If there is no concept of expiration for your assets, you could throw the exception, return some date far in the future, or even set the expiration to when the user’s access might expire. As a community, we might want to have a way to indicate “no expiration date”. Throwing an exception with that message doesn’t feel quite right.

Note that if you are implementing repository underneath assessment, assessment dates can be these dates.

RecordStructure Formats and Schemas

This information might seem to be redundant given that record structures have a type. That type should mention format and schema. Having values for format and schema that a consumer can query programmatically, and in the case where it may not know the type, allows the consumer to create an appropriate UI. An example of a format is XML. The consumer might have an XML display widget it wants to use. An example of a schema is Dublin Core.

Repeatable Records and Parts

As a consumer, it is nice to know if a given record or part must appear no more than once in an asset or can appear more than once. There are implications for a user interface as well as data entry workflow.

Data Migration

As many of you know, migrating data from one content management system to another can be awkward and tedious. The OSID model can help here. Let’s assume you have two content management systems and you want to migrate data from one to the other. Let’s also assume there is no common exchange format, so you are going to have to stitch this together. Of course, just because one system exports a data format or instructions that another system imports, you may not have your problem solved.
In our migration example, data is coming out of the “apple” system and into the “orange” system. There is a piece of “apple” logic in the migration application and also a piece of “orange” logic. Moving data from system to system requires a few paths and perhaps an intermediate operation. I don’t believe this picture is more complex than reality; probably the picture is a simplification. If you need to write this application, you need to know about the particular detail of the two systems and the protocol for communication. You also have to change things if you later have to migrate to the “grape” system.

The OSID model offers the potential for a more simple application and one that can work, unchanged, with different systems\textsuperscript{25}. Note that the migration application no longer contains system-specific logic and that the two OSID consumers have a straightforward way to exchange information. The consumer of the “apple” provider would get\textsuperscript{26} repositories, assets, records, and parts. The consumer of the “orange” provider would create these objects. It would make things more simple if both providers share common record and part structures.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{Data Migration Without OSIDs}
\end{figure}

\textsuperscript{25} This is not theoretical. We actually had a case where we needed to do this and it worked well.

\textsuperscript{26} Actually a mix of get, iterate, and recurse
Figure 60: Data Migration With OSIDs

**Downloading Asset Content**

The Asset interface has separate methods for the asset, its metadata, and content. Some assets are metadata only and have no content. Some are metadata and content. Let’s take the case of an image. The asset holds the name, description, id, and type. There could be metadata that you would access by records and parts. The image itself would be the content. A preview or thumbnail might be part of the metadata. The content might be the full image or an object that can return images of different sizes. Note that over all this is the issue of access rights and usage rights. For example, can the user get the content, and if they get the content what does the metadata say about how that content can be used? A few approaches are emerging:

- The content is a URL to an image.
- The content is a byte stream of the image.
- The content is an object that has an interface for accessing the image in various ways.

Asset content comes with a content record structure, which has a type. Each approach corresponds to a distinct record structure type.

**Uploading Content**

There is the issue of upload mechanics. There is also the issue of asset creation or updating. Uploading content mechanics is outside the scope of what I want to discuss here. Asset creation and updating touch on other topics that are covered elsewhere, specifically:

- Object Lifecycle
- Versioning
- Metadata Copying and Inheritance
- Authorization

I know of one project that downloads the content to a configurable temporary directory and returns a local URL to this.
Other Topics

Compliance
At the moment, there is no formal compliance standard for OSID provider implementations. It is up to the community and the marketplace to declare what is sufficient. Let me go into a little more detail.

The OSIDs are designed to cover a wide range of circumstances and implementation approaches. You will notice that most OSIDs have categories of methods. These can be thought of as:

- **Query**
  - These are the “get” and related methods.
- **Lifecycle and Management**
  - These are the “create”, “delete”, “add”, “remove”, “update” methods.
- **Process**
  - These methods ask the provider to invoke some operation.

You might decide to implement the methods in one category but not another. In the case of the Repository OSID, many implementations will be “read-only” and just offer query methods. In the CourseManagement OSID, one might implement the course catalog but not the student enrollment methods.

If you are not going to implement a method, throw the OSID exception with the message `org.osid.OsidException.UNIMPLEMENTED`.

There are “supports” methods for some objects. For example, a Repository has `supportsUpdate()` and `supportsVersioning()` methods. You should implement these.

Objects should have display names and ids. If you are going to skip the description, it is better to use an empty string than returning null or throwing UNIMPLEMENTED.

Installation and Deployment
The O.K.I. architecture is intended for environments with a variety of consuming applications and providers. Substitution is also something we expect. One of the elements of O.K.I. that simplifies life greatly is that all calls from consumer to provider are local. The consumer does not have to be co-located with the end-user client application. An obvious example is a browser client. There the OSID consumer and provider might be on the web server. Note that the provider implementation may be a stub only. Let’s recap this in a list of things to think about:

- The end-user is going to use an application; the application or something it works with will be an OSID consumer.
- There may be several OSID providers being consumed: more than one for the same OSID and more than one OSID.
- OSID provider implementations may require configuration. This may be done by the provider, by the consumer, or by a system integrator. Configuration may be on a per-consumer basis, a per-user basis, or both.
- Consumers need to find the provider implementations locally; the implementation may be a stub or complete.
- There may be a single provider for different uses models or separate providers. One example is desktop and web client access. The provider implementation you call when you are consuming from a desktop application may be the same exact implementation you call in, say, a servlet on a web server. On the other hand, they may be different.
- A consumer may want to use a new version of a provider.
- The OSIDs are intended to be quite stable, but they may evolve. Each provider supports one or more OSID versions.
• There are binding-language dependent conventions about where best-practice loaders expect to find providers. These conventions make it possible for consumers to locate providers at runtime in a single way.

**Consumer Deployment**

You consumer needs to load whichever providers it needs. For providers written in Java, they are probably delivered as one or more jars that you need on your classpath or in a known location. Let’s take a look at what files we might have:

![Diagram showing OSID consumer and possible jars]

**Figure 61: What Might be in a Deployment**

Here we see an OSID provider. All the classes that implement the provider are in a jar. We also have the `okiOSID-2.0.jar` file. This contains the OSIDs from O.K.I. (interfaces, not implementation) as well as the OsidLoader. There are a number of jars that contain Types the community has defined with OBAs. You might want to use those classes instead of instantiating Types by passing arguments. Using dedicated classes reduces the chances of a typo or other mismatch between consumer and provider and in the provider identifying what it supports. Maybe the provider uses jars for support. Maybe there are also third-parties’ jars. In the case of multiple consumers and providers, a lot of these jars would be the same. It is a good idea to offer these sets of jars separately so a deployment can avoid duplication. Let’s look at a sample.
Here we have a sample for federated search. There are 5 jars for types, 5 OSID provider implementations, 2 jars that support the repository implementation, and some jars from third parties.

**Provider Deployment**

As a provider, you need to assemble whatever a consumer is going to need. As you can read in the section on Consumer Deployment, it is helpful if you separate the provider implementation and any particular support jars from jars that other providers may use. This allows consumers to avoid redundancy.

The community may agree on a location for jars (bundles in Objective-C) so that loaders can find them.

**OSID Versioning**

First let me put OSID change in the context of change in technology and in Types.

The OSIDs are designed to be stable and long-lived. After all, the basic dialog between consumer and provider for domains like hierarchy, authentication and authorization, repository, scheduling, etc are pretty well known. What changes often are data formats, schemas, communication protocols, security models, etc. OSIDs hide technical detail inside the provider and specific formats and processes via Types. So we expect the OSIDs to stay the same, new types to be added and then settle down, and implementation choices to always be fluid. To summarize:

- Consumer and Provider implementations change all the time and there are new techniques and technologies coming in and out of fashion.
- OSIDs changed a lot between version 0 and 1.0 and some between version 1.0 and 2.0. Change will be less and less dramatic and will take place over time. When a particular OSID first gets a lot of attention and developers try to use it, we have seen some OSID change and then things settle down. From now on, change in a specific OSID is likely to occur after an initial burst of adoption and then stabilize.
- Types and their out-of-band agreements (OBAs) get written just as adoption of a particular OSID picks up. The community quickly writes the obvious Types. Over time, a few new ones emerge while most developers can make do with what has already been developed.
It is the stated intention of O.K.I. that any new version be a superset of earlier versions. Some methods may be deprecated. This should make things stable.

As a provider, you need to pick a version to implement. The manager interface includes a method with the version's number. Implement this method to let consumers (and the OsidLoader) know what version you support. For example, in the Java language binding for version 2.0 OSIDs, the method is called `osidVersion_2_0()`.

Different languages provide different way to test this. A provider should also document what version is supported.

If you want to support more than one version in your provider, implement each version's number method.

The current OsidLoader allows only the current OSID version. Over time, alternate provider loaders may emerge that handle version-based loading with different rules.

We have had some experience with this issue when writing a federated search application. This consumer searches both Repository OSID v2.0 providers and v1.0 providers (actually called DigitalRepository OSID rc6.1). The consumer does not know about any particular provider – the end user picks them from a list. The application tries the v2.0 loader and if that fails tries the v1.0 loader. Along with the repository we keep a flag to indicate the

---

**Figure 63: A New OSID Version is a Superset of the Old**
version. We needed this since the order of type arguments, package names, and method signatures are different in the two versions. Essentially we have separate code paths. In another application, the user makes an integration gesture that specifies the OSID version. Clearly, supporting more than one OSID version is no fun. That is why it is good if these don't change often.

**Candidate Methods for a Future Release of the OSID**

I encourage you to add comments about the OSIDs in one of the forums for that. Two forums are the O.K.I. SourceForge site and the OSID spec section on the IMSGlobal site. A couple of my personal favorite requests are a getPreview() method so it is easy to get a thumbnail and getPartByPartStructureType() so you can pluck a value out of the metadata.

**A Few Final Comments**

- Don't necessarily use OSIDs to base your service implementation development. Have some good implementation and align it with the OSID, and offer the interface from that.

- OSIDs don't contain complete enough information to be useful. There needs to be data and a model that is at best implied by the OSID. This means the OSIDs and Types must be used together to form a complete solution.