Introduction to OSID V3
for V2 Developers

October 30, 2008

Tom Coppeto

DRAFT

Contents

1 Introduction

In 2006, The V3 process was kicked off and the group assembled a list of problems to solve and new functionality to add. Each item on the laundry list fell into one of three categories:

- Interoperability
- Simplicity
- Functionality

We can have any two. Describing a complete service with complete functionality can either be interoperable or simple. V2 achieves a certain balance but several interoperability issues and use cases are left as an exercise for the reader. Yet, the V3 working group defined interoperability as more important than the other two combined and this led to finding a new balance among these characteristics.

Looking at the V3 interfaces through the linear spreadsheet of definitions is formidable and one can easily get lost in what seems like a bloat of interface definitions. First, it would be helpful to define these terms with respect to the OSIDs.

- Interoperability: the degree to which an OSID Provider and OSID Consumer can successfully interface having never met
- Simplicity: the ease in which a new OSID Consumer or OSID Provider can get into the game, and, the specificity of the interface definition to avoid ambiguity without sacrificing interoperability
- Functionality: the degree of richness available through an interface definition such that it is desirable to use the OSID as a primary API for new development

While the feedback concerning OSID V2 collected in the meetings, online forums and by way of the IMS bug reporting generally was specific to a particular issue, use case or feature request, criticisms of the OSIDs received in various venues clustered around several themes.

- OSIDs too restrictive
- OSIDs too general
- OSIDs too complicated
- OSIDs too hard to use
- OSIDs use Type agreements
- OSIDs don't work in web environments
- OSIDs don't bind well to Java
- OSIDs are not cross-platform independent
- OSIDs should be replaced with Web Services
- OSID Authentication is broken and/or unnecessary
- OSIDs need more examples

The is not intended to paint the OSIDs in a poor light as there are many success stories, but drive the OSIDs in a positive direction to increase utility and adoption. V3 is an updated look at the OSIDs with emphasis on the engineering and clarity of the interfaces. The following directives in this effort have emerged.

- OSIDs should cater to the new developer or small project with a simple set of interfaces to achieve basic interoperability as well as be able to provide more complex services for those working with more complex systems.

- OSIDs should be clearer about what a Type agreement means and improve the methods of forging Type agreements.

- OSIDs need to be more sensitive to software execution, including thread-protection, session management, and error handling.

- OSIDs need to provide a better means for creating and updating data.

- OSIDs should only focus on issues of interoperability and remain neutral to any notion of best programming practice.

Given that this document represents an OSID V3 work-in-progress, it operates under the assumption that the goal of the OSIDs and its binding to a language platform remain valid. Some effort has been made to better understand what OSIDs are and the problem space addressed by them. The following summarizes the scope of the OSIDs:

- OSIDs are software interfaces used by software programmers (they also turn out to be a good design framework for architects). They are not network protocols, databases or data formats.

- OSIDs serve as a common rendezvous point in software for an OSID Consumer and an OSID Provider. This rendezvous point may have little to do with how either an OSID Consumer or an OSID Provider view their domain or construct their software.

- OSIDs do not specify implementation issues such as data persistence & transport.

- An OSID defines comprehensive interfaces to cover a broad domain of functionality for a service.

- An OSID is not a programmer's utility toolkit although tools may be developed and shared that complement the OSIDs.

- OSIDs are not a programmers guide to writing better code and they are not designed to prevent poor coding practices.

- OSIDs may be extended in such a way as to expose more data known to smaller domains.

- OSID Provider implementations can be composed of other OSIDs and OSID Consumers can orchestrate multiple OSIDs.

This document serves as an overview of proposed OSID changes and is being used as a platform for discussion in ongoing OSID workshops. This is an evolving document. Code examples are presented to offer concrete examples of the current state and to evaluate the engineering of the interfaces, thus *For Developers*.

What Is An OSID?

One challenge has been to define and communicate what an OSID is, and more importantly, what it isn't. The OSID binding is essentially an API and there will always be APIs that are easier to use or provide more granular control. The OSID is striving to achieve very broad interoperability and interoperability is what it values most.

Interoperability is a word applied to many similar but not identical concepts. If a computer sends a TCP packet that can be read by another computer that only understands TCP, we can say that both computers interoperate. However, if the sender transmits an ADSP packet to a TCP-only computer, we obviously do not have interoperability. At the level of the network and serialized protocols, all parties must agree to interoperate.

Interoperability can be examined at a higher level. If a program expects to retrieve data from an Oracle database, but that data is only available through HTTP, we do not have interoperability. An agreement can be made to store all data in an Oracle database to achieve interoperability. However, all parties would need to also agree on database

layout, names of columns, data types, stored procedures, and even the identity of the database instance.

A further climb of the stack into application software can strive to reach higher level agreements. If there was a single way an application stores and retrieves JPEG images, it would not matter what kind of database or file system stored the images. However, if the data stored describe people, it would require a different means to query and assemble the data unless there was an agreement on how to access any and all objects. This type of object-blind model often lacks the context necessary to build a useful application.

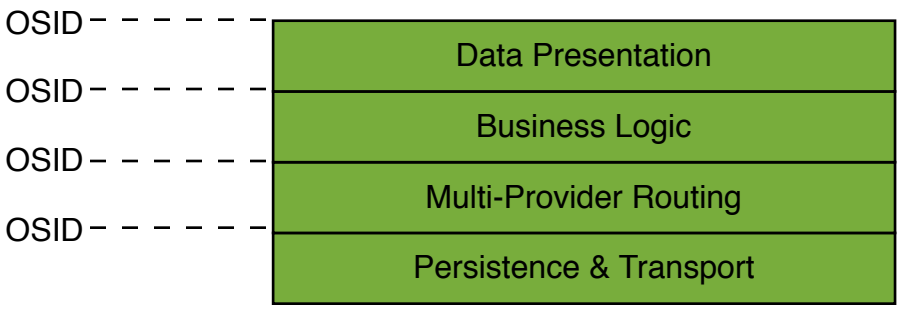
Climbing further and agreements can be made on application behavior and appearance. One might get interoperability among Swing widgets but this interoperability could not cross over into the web domain. The ultimate agreement is a single method and in many languages this is called `main()`.

OSIDs define interoperability at the service layer (thus the S). While this term is also used for a variety of things ranging from machines, to processes listening on ports, to useful application utilities, The OSID Service layer is where issues of protocol, persistence, and data transport are underneath. Their design point is simply to cover many of the mechanical variances commonly found in client/server computing while exposing enough context and extensibility on which to build a rich application.

The OSIDs expose to their consumers the means for searching, retrieving, creating, deleting and updating a variety of objects defined throughout its service definitions as well as the means for categorizing, making cross-object relationships and receiving out-of-band notification messages.

One might be tempted to draw a typical stack model but a one-dimensional view of the world does little to help classify what the OSIDs can do (we'll save that for the marketing department). If the OSIDs were pushed to a lower level in the system, they would expose a persistence model or the presence of a transport. If they were pulled too high up, they would either lose context or get locked into a specific application environment. The space the OSIDs occupy in a stack is not a line but a region.

At the lowest layer in this region is the system implementation and generally persistence and transport are the issues solved in most client/server systems. Different OSID Providers may be brought together in such a way as to give an application the point of view that it is using a single OSID Provider. There may also be encapsulated layers to localize or transform data, implement cross-service orchestration, or any such function that is not directly dependent on a graphics user interface environment (just one stack diagram and that's it).



A complete interoperability solution can be composed of a stack of OSIDs. The lowest OSID Provider in such a stack can be referred to as the OSID *implementation* while the upper layers are referred to as OSID Adapters. The highest layer responsible for instantiating the first OSID is generally referred to as the application which may or may not be a user interface. The layers can be skipped, they can be reordered, they can be duplicated. There are many possibilities to consider in design. The OSID is fundamentally a tool for encapsulation.

Encapsulation is the key. Encapsulation for whom, consumer or provider? The encapsulation OSIDs strive for is a two-way contract. The OSID Consumer and the OSID Provider are separated such that an unknown third party can interject. This third party can select only among multiple OSID Providers as well as select among multiple OSID-aware applications to mix-and-match. This third party may wish to provide some additional logic, through OSID Adapters, that transform data or implement business logic.

If a change in OSID Provider requires recompiling the application, then we have a problem. Now, does this mean all OSID Providers will work with all applications? No, that would require magic. In the realm of reality, an OSID Provider implementing a Repository service to provide JPEG images should be replaceable with another provider of the Repository service also providing JPEG images. But as this document will explore, there is much more than can be done other than simply delivering the bits of the image if we minimize the service assumptions made by the application and be creative in how software interfaces can be used.

So, we don't yet have a crisp or catchy definition but a series of characteristics and a vision of a problem to be solved.

OSIDs are software interfaces. The software interface is expected to be consumed and implemented within a single running application environment. The implementation may, in turn, make use of over-the-wire protocols and databases. The code immediately on each side of the interface, whether dynamically loaded or not, runs as part of a single running program.

Uh oh, language dependence! When in Java use JDBC and when in Ruby use DBI. Software constructs are required to implement the interface patterns that fuel the OSIDs. Without them, the OSIDs would be just a common transport. And that's just no fun.

Reusability

Basic approaches tend to model OSIDs as reusable libraries to achieve a *write-once* reusable set of service utilities that can be plugged into an application. The OSIDs are a good choice for such a framework and there are certainly other choices available that accomplish this goal for some service domains.

The OSIDs broad approach to the domain covered compared with other utility frameworks can be subtle. A *line-by-line* comparison and the OSIDs appear less detailed offering the application less control of the workings and configuration of the underlying implementation.

The OSID value proposition goes both ways. If one were to calculate the total cost of ownership from customer interaction through development to ongoing maintenance and deployment, the number is surprising. Systems, with their complex databases and protocols, racks of servers, power supplies, and miles of cable, would seem a likely candidate for cost management. However, a well-designed application carries with it teams of programmers, interface designers, usability testers, accessibility watchers, documentation writers, project managers, and quality assurance processes.

The return on investment also differs. Many system components, such as authentication servers, directories, and databases are not developed in-house and tend to last for many years, evolving slowly from one patch release to the next. In many environments, these activities are performed by a skeletal staff responsible for the maintenance of a multiple systems.

Application changes tend to be more significant from one release to the next requiring sizable ongoing investment. Limiting assumptions made by application designers of the problem scope and the underlying infrastructure, when it does change, often result in throwing away many work-years of investment. Application code that is too tangled up with the details of data markup, authentication systems, persistence models or an obsolete libraries may find itself the subject of a meeting to decide if its time to walk away.

OSIDs force themselves between the application and these assumptions. This can be cumbersome to application programmers accustomed to handling the underlying details directly (they can also write the OSID Providers to accompany their application).

OSIDs exist to protect investments in applications. This is where the bulk of the cost is.

2 Specification Framework

OSIDs

The OSIDs are specifications defined in XML documents called the X-OSIDs. The X-OSID schema strictly defines the structure of an OSID interface. The purpose of the X-OSID is produce OSID Bindings and other artifacts generated from the specifications, such as presentational documents. The end-products are the OSID Bindings. One does not code to the X-OSID since it is merely XML data that expresses a specification. It is also not useful to view the X-OSID as a means of data transport since an OSID does not specify data format and transport.

The current draft OSIDs are maintained in Excel spreadsheets. This was, in theory, for the convenience of writing the specification and to clearly read the definitions. The data models, the relationships among the various interfaces, and what it takes to get started using an OSID is difficult to ascertain without some familiarity with the OSID patterns and binding behavior. We don't expect a developer to pick up a spreadsheet and start coding. This is a job better suited for a developer's guide written for a targeted audience, scoped to a particular problem area, within a given programming language.

OSID Binders

The OSID Binders transform the OSIDs into OSID Bindings which are expressions of the specifications in native programming languages. The V2 Java OSID Binding created some awkward constructs as seen from a programmer's view. The original OSIDs were written in Java while at the same time attempting to predict how it would later fit into other programming language paradigms. Because V3 OSIDs use a neutral specification language, there is room for a language binder to figure out how to best apply the definitions to a given language while living within the bounds of the OSIDs. An example is Java exception handling where the OSIDs simply define error states and the Java OSID Binder decides how to implement them.

The rules used by a given language binder are another level of specifications as they most directly affect the interoperability among software components. For all practical purposes, the spreadsheets and the X-OSIDs are simply back-end tools for constructing the OSID Bindings people will use and upon which interoperability depends.

Writing to The OSIDs

A shortcut throughout the V3 OSIDs is the use of interface inheritance. This technique helps to define clusters of common methods, readability, and ensure consistency. Depending on how the OSID Binder treats this (a binder may expand the common definitions and eliminate any hint of inheritance, or it may preserve the constructs, or it may provide new hierarchy scheme), there is the opportunity for misinterpretation on how the OSIDs should work.

In V2, all managers implement the OsidManager super-interface, which contains a set of methods common to all managers. An implementation of RepositoryManager may elect to

a) Implement an OsidManager and parallel the OSID specification.
b) Implement an OsidManager and organize the methods differently.

In V3, several additional top-level interfaces are defined. One such example is OsidObject that which other objects such as Agent and Asset implement. The Java binder (currently) respects this interface hierarchy and defines an OsidObject Java interface. An OSID implementation of Asset has the following choices:

a) Implement an OsidObject and extend other Asset from it.
b) Do not implement OsidObject separately but satisfy the complete interface contract for both Asset and OsidObject in the Asset implementation.
c) Use a completely different class hierarchy for the implementation of Asset and satisfy the interface contract for OsidObject + Asset in some other object.

The moral of the story is that while an implementation must ultimately satisfy the interface contract specified in an OSID, it is free to do whatever it wants to get there. The OSIDs do not describe the arrangement of the implementation classes. A common arrangement in V3 is to define a class that implements all of RepositoryProfile, RepositoryManager and RepositoryProxyManager. However, it would not be wise to have the same object implement AssetLookupSession or AssetList since these interfaces require stateful objects while the managers are stateless. More on this later.

The super-interfaces do provide a handy vocabulary to describe a category of OSID interfaces. This document refers to these interfaces to mean *all interfaces of this kind*.

The class hierarchy is strictly in the realm of the OSID Provider and the OSID Consumer should not make any assumptions concerning interface inheritance. In Java, for example, one can use a combination of casting, generics and/or polymorphism to move up or down both the interface and object hierarchies. This poses a problem for the OSIDs because the OSID Provider's objects must not be visible to the OSID Consumer. There's

also a possibility that casting an OSID interface creates another *object* that does not exist from the OSID Provider's view (see Casting). Therefore, OSID V3 tries not to specify any objects or methods that require casting because any relationships are now explicitly described in the specification. The super-interfaces, if bound, do not imply a class relationship and shouldn't be accessed by an OSID Consumer.

Two exceptions to the casting rule are the instantiation of a manager (although it could be considered cleaner to specify a method for each OSID, but we didn't bother) and acquiring an interface to a record. More on this mechanism later.

Specification Principles

Along with the scope listed in the previous section, there are also a number of principles that have been followed in assembling the OSID Specifications.

- OSIDs define a contract between two parties. Their interoperability depends on a contract that is clear and precise at the expense of specification verbosity.
  - Methods define arguments and return the values using specifically typed primitives and interfaces.
  - Having two crisp methods is better than one vague method requiring more code to implement.
- OSIDs do not assume *likely* scenarios. OSID patterns are applied consistently, even when it seems like overkill, to facilitate adaptation or other unforeseen circumstances.
- Method signatures are constructed using a set of defined primitives and other OSID interfaces. The OSID Specification cannot reference any external primitive or class (although an OSID Binder may inject one).
- Method overloading is not permitted.
- Methods are not a scarce resource. Parameter-stuffing a method should be avoided with a set of well-scoped methods.
- OSIDs should avoid situations where an OSID Consumer implements interoperability logic based on an error state (programming by exception). Apart from operational, user-driven errors, there must be a way to use an OSID to avoid interoperability or parameter validity errors.
- Consistency is important to establish a single learning curve.
- Inline interface documentation faces the OSID Consumer. The OSID Provider is bound by the method contracts and may stray from the letter of the

documentation. Issues of application control flow that should be respected by an OSID Provider are documented in the Provider Notes.

There are also a number of usage principles in consuming or implementing an OSID.

- Nulls are not permitted as return values or method arguments.
- OSID Providers should help enforce the interface contract through state and parameter value checking even if it doesn't care.
- OSID Providers are expected to manage unspecified integration issues such as configuration, localization or presentation formats. Often, these are best encapsulated in an OSID Adapter.
- OSID Provider implementors should not sweat over a pile of methods with no return value. It isn't likely the OSID Consumer will notice.
- OSID Consumers should avoid breaking encapsulation through casting except where explicitly permitted following a Type negotiation.
- OSID Consumers should have a healthy skepticism of an OSID Provider, apart from adherence to the specification. Examples include eternal blocking and infinite iteration.

3 Structural Changes

Osid Managers

In V2, the manager's primary function is to access OsidObjects, which, in turn, may access other OsidObjects. OSID Providers had the option to not implement arbitrary methods resulting in a vaguer understanding of interoperability.

In V2, all data access through an OSID to an implementation had to be thread protected in the event multiple processing threads accessed the OsidManager. Managing session data across multiple users using a single manager was not possible.

In V3, the OsidManager remains the primary entry point but is reduced in scope. Its only purpose is to profile a service and provide access to OsidSessions. Access to OsidObjects is described in OsidSessions. The OsidManager profiles a service by providing methods to test for compliance for some aspect of a service, *is it supported?* If the OSID depends on any Types, the OsidManager also provides the means to test for Type compatibility.

The interoperability tests inside the OsidManager are specified in an OsidProfile. The OsidProfile interface is not visible to the OSID Consumer but is implemented by an OsidManager. It is used as a means of organizing the managers' interoperability issues separately from session instantiation that may occur among multiple managers within the same OSID.

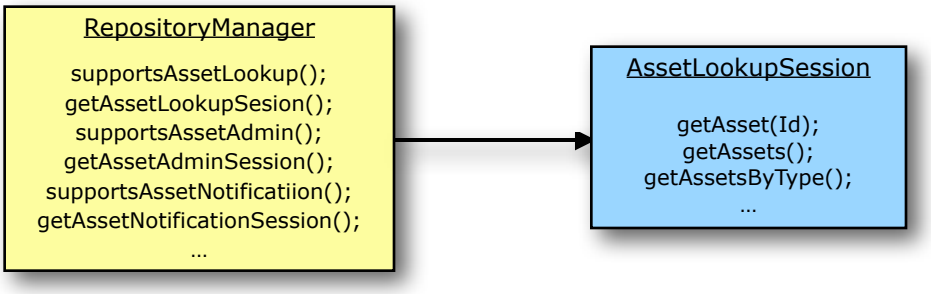
Osid Sessions

The primary purpose of the OsidSession is to provide a place where session related data can reside and to allow the OsidManager to operate in a stateless multi-user environment. The OsidSession, by specification, is to be used by a single user and within a single processing thread. An OsidManager is intended to be instantiated once per service while an OsidSession may be created for each user, operation or request. An implementation should therefore strive to perform most of the heavy-lifting while initializing an OsidManager, and attempt to keep the OsidSession initialization as lightweight as possible since the OsidSessions will be created and tossed frequently while a single OsidManager may be running for the entire life of the application.

The OsidSession defines a cluster of methods that represent an aspect of a service. The groupings are sorted such that it is more likely than not a OSID Consumer and/or OSID Provider will be interested in all the methods in a cluster. Many clusters are organized

along the lines of read and write although the sessions provide a framework for adding more interesting aspects to an OSID while not adding a burden to those OSID Consumers or OSID Providers who have no interest in additional functionality. Consequently, there may be many sessions defined in an OSID. A simple application may only use a single session while an administrative console may be interested in making use of many sessions.

Methods defined in an OsidSession are usually designated as mandatory because it is the support of a session itself that is optional. Therefore, interface level compliance can be described in terms of what OsidSessions are required by an OSID Consumer and available from an OSID Provider. This compliance is described in the OsidManager.



A session is created through a manager.

Osid Objects

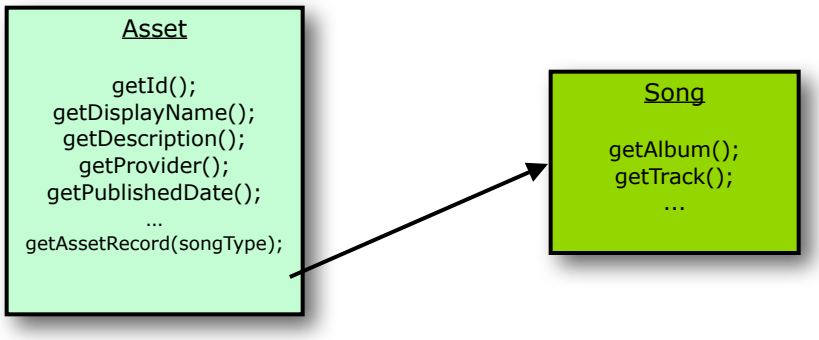
OsidObjects map to a resource used within a service provider and are identified by an Id. In V2, the OsidObject is generally used to carry the Id and a set of generally applicable data that can be retrieved or updated. In some cases the OsidObject would take on an object-oriented flavor and provide more action-oriented methods and in other cases actions were defined in the OsidManager.

In V3, the OsidObject is strictly the carrier of an Id, and a set of data. Actions upon objects, mappings with other objects, and updates to the data are generally defined in an OsidSession. The goal is not to involve the OsidObject into additional compliance issues already addressed at the session level.

An example OSID object.

The OsidObject is not an object in an object-oriented way. The OSIDs describe services, not objects, and as such they are expressed in a service-oriented fashion through a series of sessions and methods. The OsidObject identifies a resource managed by a service, along with any data describing it. It cannot be viewed as a complete object or data record although it may be implemented as a single object within an OSID Provider. More importantly, persistence is based on changing one or more elements of the object, not persisting a snapshot of the object through the service layer. This view is consistent with V2. V3 OSIDs separate read, write and search operations, and better facilitate bulk changes to an object (see Updating OSID Objects).

The data defined in an OsidObject is minimal to keep the objects general. Where it can be said, *an object generally has,* the data element is defined. However the data may be extended through an OsidRecord. An OsidRecord is an interface whose specification is identified by a Type and is intended to be used to access additional data elements. Relationships among OSID objects and any actions are defined in a session. An Asset, for example, may be extended to describe a Song using a Song Asset Type.



An example OSID object.

Where OSID objects are required as method parameters, their Id is used. This removes the requirement that an object retrieval must always be performed before an action is taken. For method returns, the OSID Consumer generally has the choice of retrieving the OsidObject or the Id.

Properties

Some V2 OsidObjects, such as Assessment, Asset, Message, and LogEntry define only the top level generalized data elements and leave any additional data to be defined in the form of an interface plug and an associated Type.

In V3, this model has been generalized across all OsidObjects through OsidRecord interfaces leaving the Properties mechanism to be somewhat redundant. However, accessing data access through a record requires a Type agreement. Some applications may find it useful to display information available through an OsidRecord without having any knowledge about the record type.

For this reason, a properties list remains available through the OsidObject. The properties interface has been simplified to allow for easier traversal and the properties Type has been consolidated into the OsidRecord Type.

Lists

In V2, the OSID Iterator provides a sequential one-at-a-time pass through of a set of objects. V3 allows for multiple objects to be requested in a single call. The interface is now referred to as an OsidList and remains sequential.

An example OSID list..

The OSID iterators attracted criticism when compared to built-in language constructs such as arrays, hash tables, and Java's Collection interfaces. There are two design principles worth noting here.

One principle is that the OSIDs are *typed*. Methods return what they say they return. getNextAsset() returns an Asset and getNextAgent() returns an Agent. In cases where this cannot be possible, a Type is used to define the return. The interface contract is defined to be as specific as possible to maximize interoperability.

Another principle is that an iterator maybe bottomless, or may appear bottomless if the applications runs out of memory. An application programmer might assume a fixed amount and the idea of adapting to a variable result set may seem daunting. However, as more OSID Providers are brought into a federation, the question of how will the application handle all the results becomes relevant. OSIDs do not assume a fixed set of

elements that can be all held in core memory at once and do provide a framework to increase the size of a federation behind the application programmer's back.

A stream is a good way to think about an OsidList.

OsidContext

The OsidContext had been used primarily as a means of passing authentication-related data through the interface for the purposes of seeding an Authentication OSID Provider. Understanding how authentication should be performed using the OSIDs has been difficult and in practice many methods have been implemented.

The V3 Authentication OSID has been reworked to define an in-band method of passing data for the process of acquiring or validating an authentication credential that requires an agreed Type specification (typed interface) between the OSID Consumer and OSID Provider.

V3 favors making agreements based on interface specifications and not based on data (see Types). OsidContext is not defined in V3.

DateTime

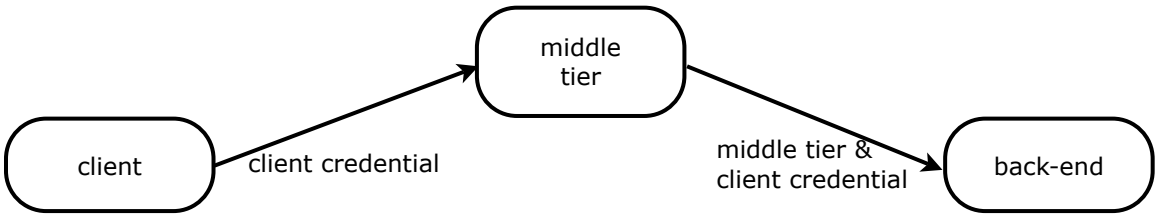
V2 used a long to represent an abstract date and time. The problem with many abstract date/time representations is the precision is fixed. An OSID Provider using a date of 1776 is represented by an abstract date of January 1, 1776 12:00.00 a.m. If the date is unknown, often 0 or -1 is used. The New Year's party of 1970 had a lot of events going on.

V3 defines a DateTime interface to convey variable precision. A date/time might only be known to the nearest century or the measurement recorded at a specific picosecond. The DateTime isn't subject to limitations caused by counting milliseconds from an arbitrary time.

DateTime may also convey uncertainty. A historical event might have occurred in January 1944 but it is known that it occurred some time during that winter. A DateTime can convey this by expressing year 1944, month 1, with a granularity of month, and an uncertainty of +2 months / -1 month.

4 Proxy Authentication

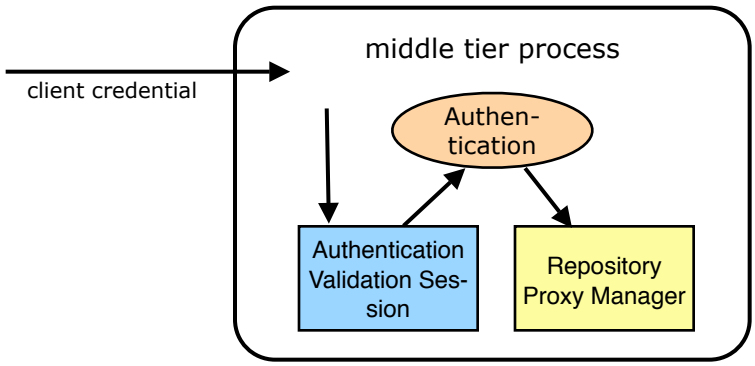
Proxy authentication is used when an authentication credential needs passing from service to service. For example, a middle tier service may authenticate its user, and in turn authenticate to a third-tier service. The third tier may be able to authenticate the client directly based on some cryptographic data or based on an assertion from the middle tier. This argues for a means to pass authentication or an identity through all the OSIDs because it *might* be used in a middle tier.



Authentication flow in a typical three-tier system.

In V2, this is somewhat addressed by using the OsidContext. The missing pieces are session creation and the means to accurately describe interoperability. In V3, a separate OsidProxyManager is defined for each OSID to support the passing of an external credential at the time of session acquisition. The separation at the manager level is intended not to confuse proxy-authentication with the acquisition of a primary authentication credential.

In the middle tier, the existence of an explicit back-end tier is somewhat of a red herring since this would be encapsulated by another OSID. The question really becomes, how do we pass a credential or an identity into an OSID from the outside? On the client side, the Authentication OSID is encapsulated within another OSID and not directly accessed by the application layer. On the server side, the process flips and the process of authentication occurs before another service, local or remote, is accessed.



Examining the middle-tier. The client credential received from the incoming protocol is passed to an Authentication OSID that returns an Authentication object. This Authentication object contains the Agent identity and can be passed to another OSID via an OsidProxyManager.

The inbound Authentication OSID Provider and the Repository OSID Provider (which in turn may use another Authentication OSID Provider) agree on the Authentication Type. While the Authentication object makes available the Agent Id for use with an Authorization OSID internal to Repository, its Type specifies the interface providing access to any credentials that may be passed to another tier.

Some application server environments handle user authentication outside the scope of any given application instance. In these cases, the Authentication OSID does nothing more than map the supplied authentication identifier to an Agent Id and wrap that inside an Authentication object. Using this design pattern, although not useful for passing a client credential to another tier, the mapping of an authentication identifier to an OSID identifier is modularized. This modularity is very helpful for solving problems of federation and changes to authentication specific identifiers.

One may wish to use the Authorization OSID directly for application-specific authorizations outside the scope of any other OSID. Having access to the Agent Id via Authentication makes this possible.

An OSID Provider may elect to support a proxy authentication manager where proxy authentication is not applicable to broaden interface interoperability.

```
Cookie[] cookies = request.getCookies();
for (Cookie c: cookies) {
    if (c.getName().equals(SAML_COOKIE)) {
        token.setInputData(cookie.getValue());
    }
}

AuthenticationValidationSession avs;
RepositorySearchSession rss;
Authentication auth;
SamlTokenInterface token = new SamlTokenInterface();

try {
    avs = authMgr.getAuthenticationValidationSession();

    try {
        auth = avs.authenticate(token, SAML_TYPE);
    } catch (UnsupportedException ue) {
        error("authentication provider does not support SAML_TYPE");
    } catch (PermissionDeniedException pde) {
        error("hmmm.. I have no authorization to authenticate this user. How silly.");
    }

    try {
        rss = repProxyMgr.getAssetSearchSession(auth);
    } catch (UnsupportedException ue) {
        error("repository provider does not support this authentication");
    }
} catch (OperationFailedException ofe) {
    error("an error occurred somewhere along the way");
}
```

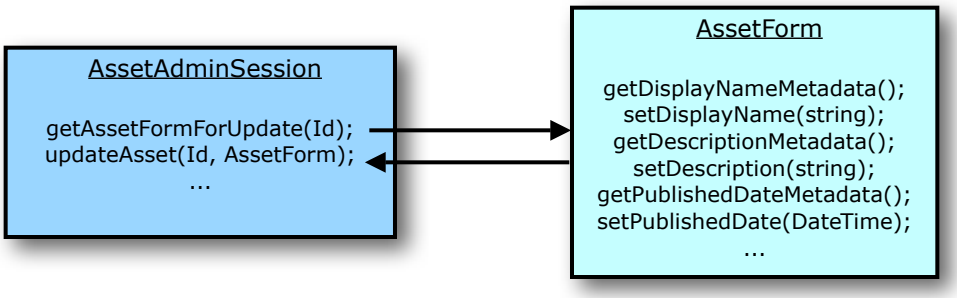
Code example of a servlet handling a SAML token via a cookie and passing the identity to a reposi? tory service. The SamlTokenInterface is an example type specification understood by the authenti? cation service identified by the Type SAML—TYPE. The managers have been instantiated as class variables in the servlet and used by all servlet instances.

Managing OSID Objects

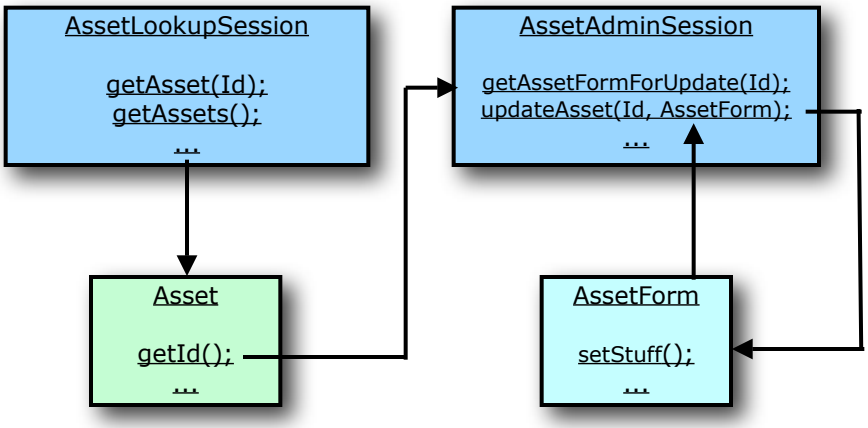
5

In V2, updating an OsidObject is performed through an update of an individual data element within the object itself. Multiple updates, even if required by the OSID Provider, had to be enabled through the OSID Transaction interface. V2 combines both an object and a transaction oriented update paradigm where object retrieval is required before an update can occur and each data element has a corresponding update method. The rules for updating data is assumed as part of a data agreement which is not consistently identified with a Type.

V3 uses an entirely different approach. First, a separate interface for updating exists for each OsidObject that defines a set of methods. These methods generally correspond to the data elements defined in that object. The OSID Consumer may set one or more elements, then pass the interface through an update method in an OsidSession. This interface is called an OsidForm.



Updating an OSID object involves getting its form, setting the desired changes and invoking the session's update method.



A more complete flow showing including an Asset retrieval. The retrieval is optional if the OSID Consumer already knows the asset's Id.

It appears the object has been split into two halves; the read half and the update half. This allows the OSID Provider maximum flexibility in implementing an update mechanism and allows for a separate compliance designation. Here are a couple of implementation scenarios.

- An OSID Provider may implement a transactional-based mechanism where the implementation the OsidForm contains only metadata and any element set by the OSID Consumer in the form is included in a single update transaction.

- An OSID Provider may implement an object-based mechanism where the OSID Provider's object implements both the Asset and AssetForm interfaces, and contains all the data known to the OSID Provider. Invoking the update method persists the entire object using an object-oriented persistence layer. In this scenario, getAssetFormForUpdate() and getAsset(Id) retrieve the same object.

A Type that indicates the typed interface for an OsidObject also specified the interface for the OsidForm.

Metadata

The form also makes available metadata on a per-element basis to aid in specifying certain restrictions the OSI Provider may impose on an update. Metadata for a description may indicate the text length is limited to 255 characters, or a valid number is between 1 and 10, or only the strings "red", "green" and "blue" are accepted. An OsidForm is requested for the object to be modified and the OSID Provider can vary the metadata on an object by object basis.

```
AssetForm form = session.getAssetFormForUpdate(assetId);
Metdata metdadata = form.getMetadataForDisplayName();
if (metadata.isReadOnly()) {
    print “display name cannot be modified”;
} else {
    print “Display name limited to “ + metdata.getMaxStringLength() + “ characters.”
}
…
form.setDisplayName(displayName);
if (!form.isValid()) {
    print “Display name is invalid: “ + form.getValidationMessage();
} else {
    session.updateAsset(assetId, form);
}
```

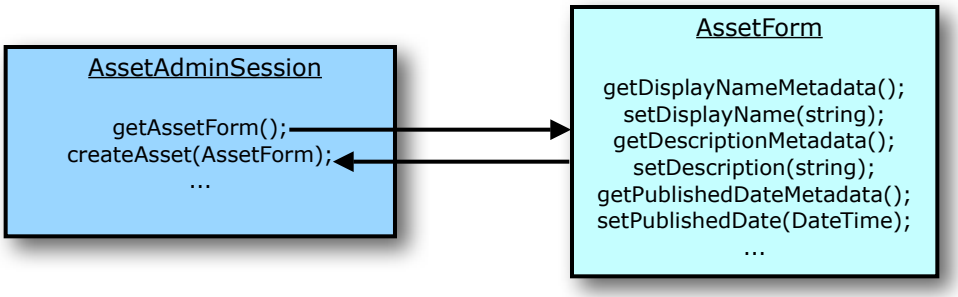
Code example of examining Metadata before updating an Asset.

Metadata is supplied by an OSID Provider to help guide the OSID Consumer through an update. Metadata can help an OSID Consumer avoid an error resulting from invalid data and provide hints to a user interface on how to display input fields.

OsidForms also offer a validation method. This method can help catch problems resulting from the combination of data submitted through an OsidForm.

Creating OSID Objects

In V2, creating an OsidObject involved specifying the set of object elements as parameters defined in a create method. In V3, the OsidForm is used to create OSID objects. The OsidForm also serves to simplify the parameter lists of the create methods.



The create process is analogous to updating an OSID object.

An OSID Provider may require certain data to be supplied in creating an Asset. An OSID Provider can specify in the Metadata if the field is required or optional. The OSID also needs to convey which OsidRecords are required for create. It does this through the canCreateAssetWithRecordTypes() method.

```
/* attempt to create the asset with no records */
Type[] types = new Type[1];
if (!session.canCreateAssetWithRecordType(types)) {
    print “A record type is required to create an asset.”;
}

/* attempt to create an asset with a painting record */
types[0] = paintingRecordType;
if (!session.canCreateAssetWithRecordType(types)) {
    error “Cannot create an asset with a painting record type.”;
    return;
}

AssetForm form = session.getAssetFormForCreate();
if (!form.hasRecordType(paintingRecordType)) {
    error “form has no record, why did it say I could create one?”;
    return;
}

PaintingFormRecord record = form.getAssetFormRecord(paintingRecordType);
Metadata metadata = record.getCanvasWidthMetadata();
if (metadata.isReadOnly()) {
    error “cannot set canvas width”;
    return;
}

if ((newCanvasWidth > metadata.getMaxCardinalSize()) ||
    (newCanvasWidth < metadata.getMinCardinalSize())) {
    error “canvas width is invalid”;
    return;
}

if (!form.isValid()) {
    error “invalid form: “ + form.getValidationMessage();
    return;
}

/* this example only checked a single field, there may be multiple required
   fields for a successful create operation */

Asset asset = session.createAsset(form);
```

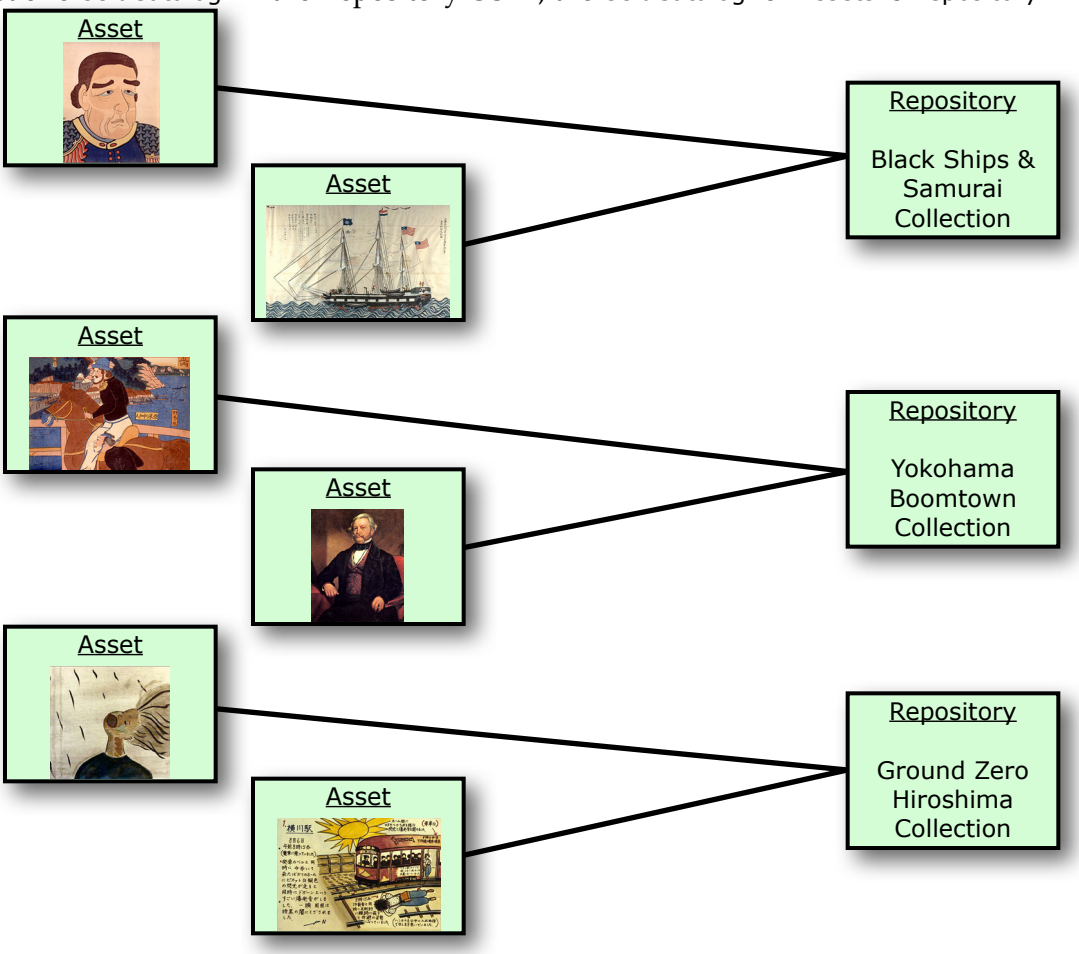
Code example of creating an Asset with a record.

6 Cataloging & Federation

OsidCatalogs

Some OSID Objects serve the purpose of cataloging other OSID Objects. In V2, examples of this categorization are Repository, Dictionary, and ReadableLog. V3 has expanded this set into a variety of other OSIDs.

An object *maps* to an OsidCatalog. At the interface level, this does not necessarily imply that an OsiodCatalog *contains* an OsidObject although it may be implemented that way. An OsidObject may be mapped to more than one OsidCatalog and is always mapped to at least one OsidCatalog. In the Repository OSID, the OsidCatalog for Assets is Repository.

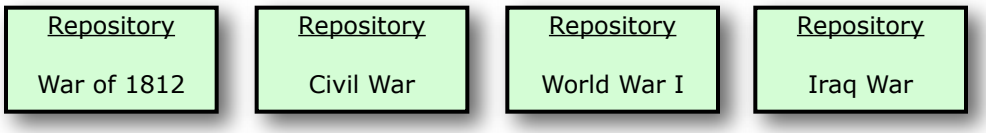


An example use of Repositories for the MIT Visualizing Cultures Project.

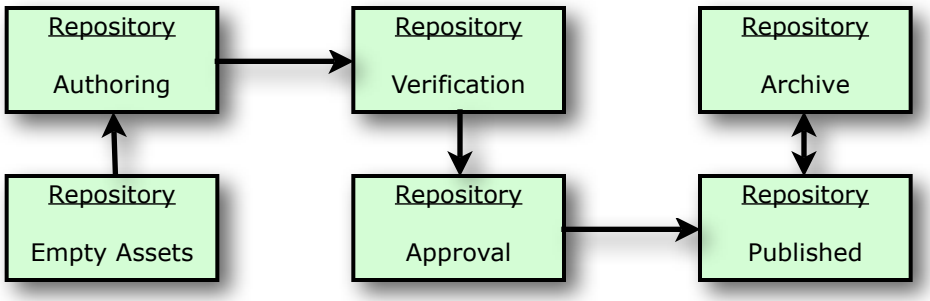
An OsidCatalog is an OsidObject and it defines additional methods to convey a provider of the catalog. This can offer a different interpretation for the use of a catalog.

Example use of Repositories representing various providers.

Cataloging objects can be used to serve a variety of organizing needs ranging from expressing an ontology, to customizing a view for an OSID Consumer (a Repository of PDF files) to facilitating a workflow (a Repository of incomplete Assets). A Repository can be used to present categories of Assets useful to an OSID Consumer that may or may not be independent of the organization of the underlying data storage.



An example organization using Repositories.



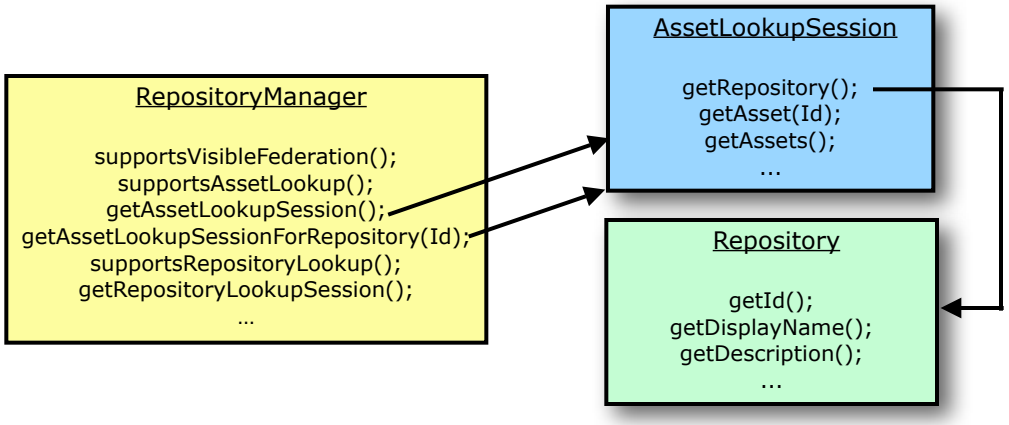
An example workflow using Repositories.

The Repository OSID in particular defines additional relationships for capturing subject matter described in the Repository chapter and the Workflow OSID can capture the state of the Asset as it travels through a Workflow process. Cataloging, however, provides a generic means for organizing OsidObjects throughout the OSIDs that can be used to provide desired views of collections to an OSID Consumer.

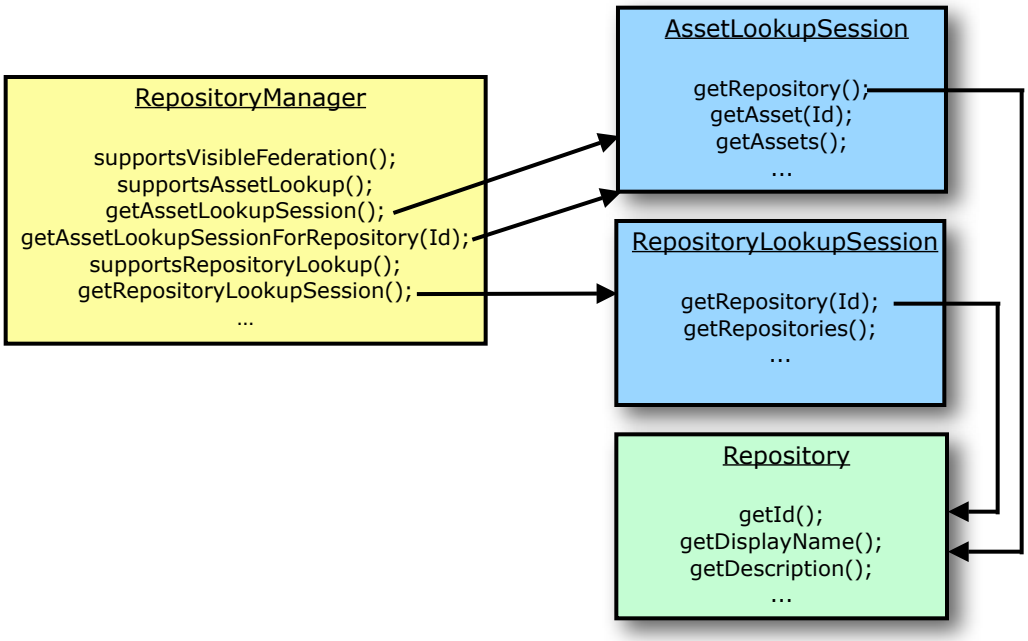
Catalogs and Sessions

When an OsidObject is created, it is created with a mapping to an OsidCatalog. For example, an Asset is created with a mapping to the Repository to which the AssetAdminSession is associated. An AssetLookupSession is associated with a particular Repository from which to retrieve Assets. An OSID provides a separate session for changing these mappings.

Simple OSID Providers and OSID Consumers may have no use for the notion of cataloging. As such, it is an optional area of compliance referred to as *visible federation*. A federation is visible if one can request an OSID Session within a category or container. There is always a default session that does not require specifying a federated object. An AssetLookupSession, for example, can be accessed by specifying a Repository or by specifying no Repository. This default mechanism exists for OSID Consumers that are unaware how to select a Repository and leaves it to the OSID Provider how an appropriate federated object is selected.



Getting a default session. An OSID Provider may elect to provide a root hierarchy node or some other Repository may be assigned for a particular OSID Consumer based on its configuration.



Getting an AssetLookupSession where the OSID Consumer specifies the Repository.

The previous diagrams show how the sessions come together to describe several types of interoperability.

- The OSID Consumer has no knowledge of a Repository nor does it know to find one. In V2, a Repository Id would be given to the application programmer to hand into the Repository OSID. This makes it more difficult to change the Repository used by the application and removes the ability for the OSID Provider to reorganize. OSID Adapters, although requiring additional code, provide the ultimate flexibility in deciding how a default Repository can be selected.

- The OSID Consumer requires access to Repositories and provides end-user functionality for searching and selecting Repositories. Such functionality may go hand in hand with an application to organize Assets or as a means to scope a federated search. This requires the OSID Provider support Repository lookup sessions.

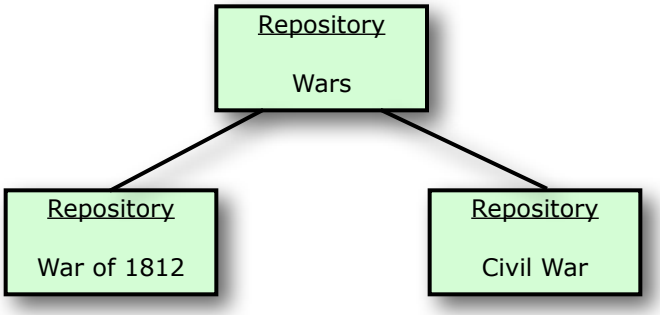
- The OSID Provider has no concept of Repositories. In this case, the OSID Provider must expose its own Asset collection through a single Repository interface for the getRepository() method of the AssetLookupSession and assign a plausible Id to it. This would not be compatible with an OSID Consumer requiring visible federation or Repository lookups. However, this can be layered over such an OSID Provider through the use of an OSID Adapter.

The Cataloging OSID can be used by an OSID Provider to factor out the cataloging implementation into a separate OSID.

Hierarchical Catalogs

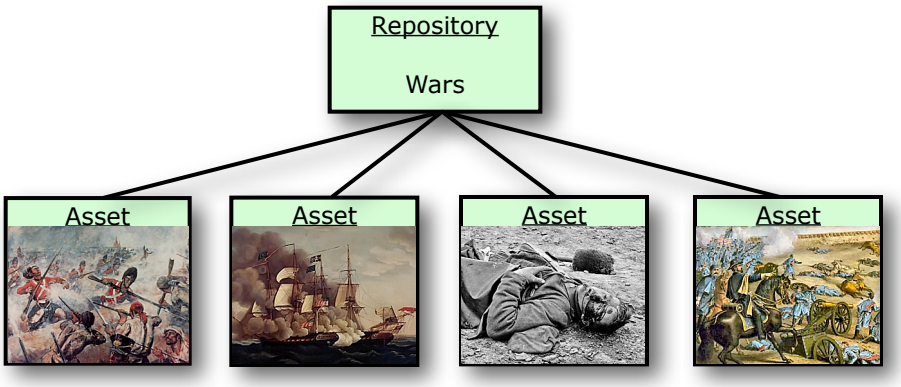
Many V3 OSIDs define interfaces for organizing objects into hierarchies. The hierarchy interface is exposed directly in the OSID to facilitate the alignment of an OsidObject with an Id in the hierarchy. Often, a position in the hierarchy will imply a certain behavior depending on the OSID, the OsidObject and the hierarchy defined. These behaviors are defined in the specification.

For example, a Repository OSID may not only support the Repository service, but also support a hierarchy of Repositories. A Repository that is a parent of another Repository *implicitly* includes the Assets of the child Repository in its Repository. This is a useful tool in a federated scheme where an OSID Provider may wish to offer more granularity in its federated views.



Repositories structured in a hierarchy.

Repositories and their Assets.



The Wars Repository implicitly includes the assets of each child Repository.

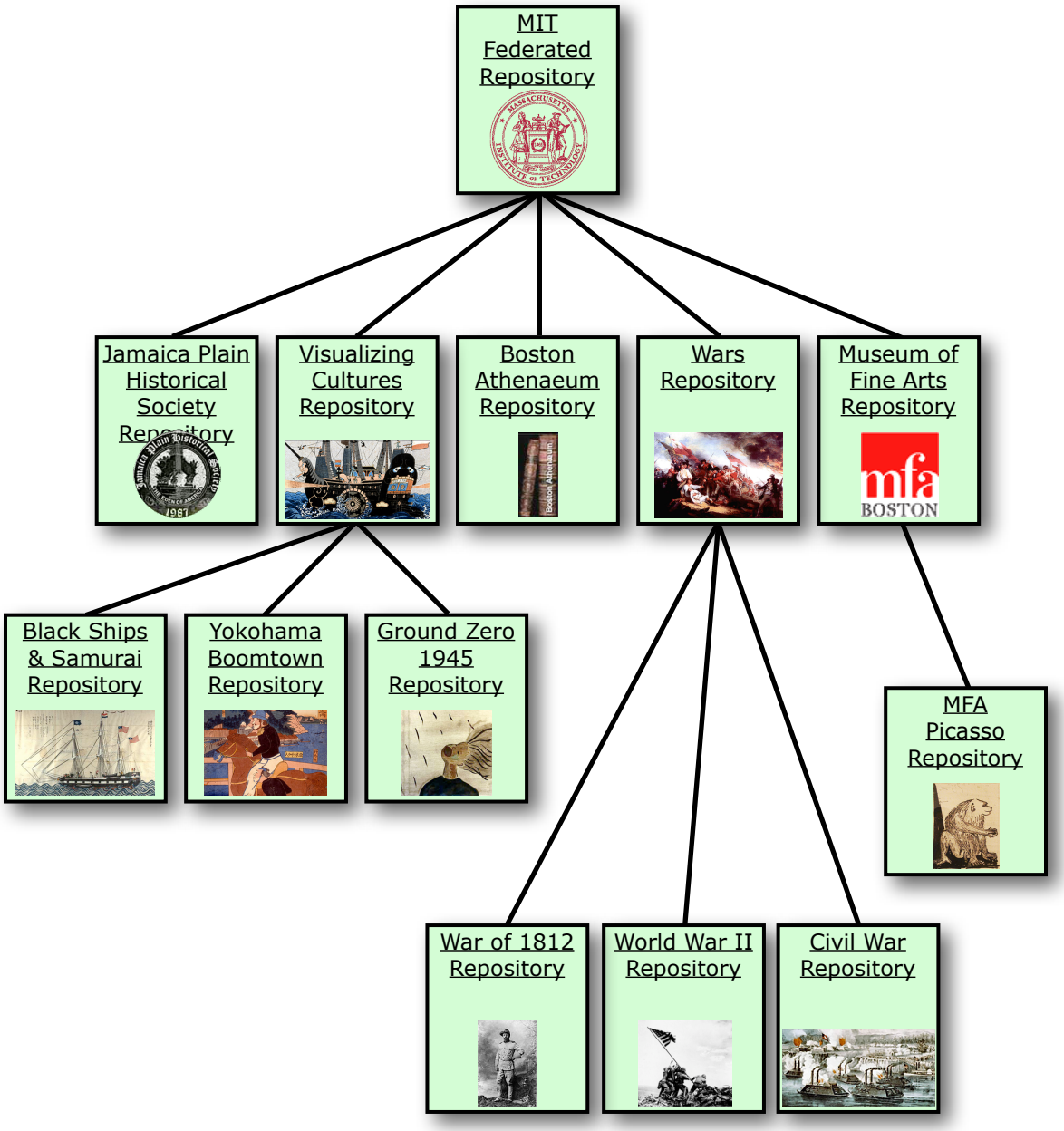
The OsidCatalog hierarchies behave in reverse of a hierarchy that describes inheritance. This is so that the roots of the hierarchy roots represent the root of the federation. Other hierarchies may define different behavior.

A federated scheme, such as the one illustrated above, creates an ambiguity when it comes to identifying objects for administrative operations. For example, an OSID Consumer may wish to look at an implicit repository for searching but see the explicit repository when managing Repository mappings. OsidSessions provide methods to enable and disable federation. These toggles are called *views* (see Session Views).

The Hierarchy OSID can be used by an OSID Provider to factor out a hierarchy implementation.

Catalog Adapters

Federation can be accomplished by using OSID Adapters. An OSID Adapter can combine multiple repositories from various OSID Providers. A simple federating OSID Adapter may loop through each sub-repository on a method-by-method basis. More complex adapters can be constructed that optimize for search patterns, routing or subject material.



A federation of repositories.

Catalog OSID Adapters may be developed for a variety of purposes. See the OSID Adapters chapter for more examples.

7 Searching

In V2, searching is performed by passing an arbitrary search object. In V3, searching is performed by using a set of interfaces to increase interoperability.

Basic Search

The basic interface is an OsidQuery that can be used to construct a simple search of objects. The Repository OSID defines an AssetQuery for searching Assets. The AssetQuery specifies basic query parameters that align with the Asset object. This include the asset's display name, description, or provider.

```
AssetQuery query = searchSession.getAssetQuery();
query.matchDescription("*blues*", wildcardStringMatchType, true);
AssetList assets = searchSession.getAssetsByQuery(query);
```

Code example of using a search query. Assets are returned whose descriptions contain the string "blues".

The boolean flag in the match method instructs the OSID Provider to perform a positive or a negative match. String match methods also define a Type parameter to describe the format of the string.

Multiple fields can be matched within a query. Multiple field terms behave like a boolean AND.

```
AssetQuery query = searchSession.getAssetQuery();
query.matchDescription("*blues*", wildcardStringMatchType, true);
query.matchDisplayName("Liberace", wordStringMatchType, false);
AssetList assets = searchSession.getAssetsByQuery(query);
```

Code example of using a search query. Assets are returned whose descriptions contain the string "blues" AND whose display names do not contain the word "Liberace".

OsidQuery interfaces also define a method for matching arbitrary *keywords*. What these keywords match is up to the OSID Provider but they can be used to implement a simple Google-esque search.

```
AssetQuery query = searchSession.getAssetQuery();
query.matchKeyword("music", wordStringMatchType, true);
AssetList assets = searchSession.getAssetsByQuery(query);
```

Code example of using a search query of keywords. Assets are returned that are relevant for "music" in some way only known to the OSID provider.

An OSID Consumer may invoke a match method multiple times. This results in a nested boolean OR.

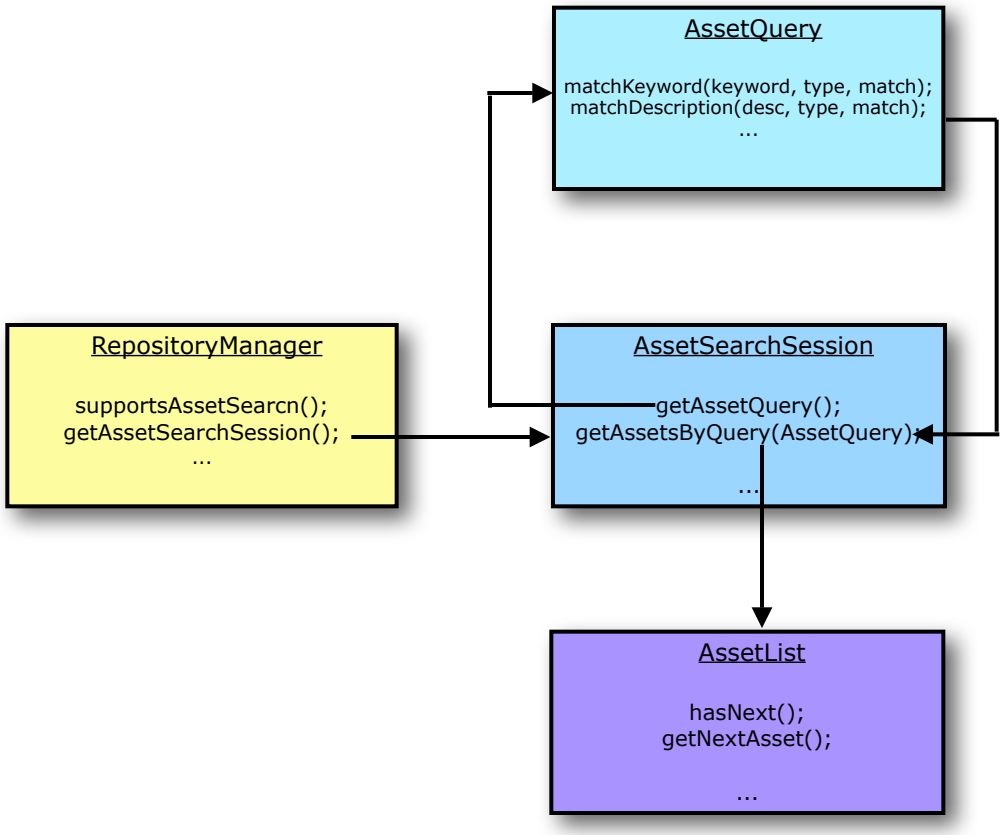
```
AssetQuery assetQuery = searchSession.getAssetQuery();
query.matchPublishedDate(DateTime1, true);
query.matchPublishedDate(DateTime2, true);

AssetList assets = searchSession.getAssetsByQuery(query);
```

Code example of using multiple matches of the same element. Assets are returned that whose published date matches the date specified in DateTime1 or DateTime2.

It might seem that if dealing with text searches, multiple invocations of matchDescription() could result in an AND term. To keep the methods simple and provide consistency with non-string match methods where an AND results in an empty set, particular arrangements of string searches has been delegated to the stringMatchType that could indicate a pattern of a word sequence. For example:

```
matchDescription("fox:brown:quick",matchWordsAnyOrder, true);
```

The diagram below illustrates the interface control flow of a basic search query. The AssetQuery interface is supplied by the OSID Provider and the OSID Consumer is required to submit an AssetQuery it retrieved from the OSID Provider. The implementation of the AssetQuery contains logic to assemble the query terms and map those terms to underlying data.



The control flow for a basic asset search.

Searching Records

An OsidObject can contain zero or more OsidRecords indicated by the record Types it supports. An OsidQuery parallels the OsidObect and supports an OsidQuery record for each record supported in the OsidObject.

Joining Queries

Some OsidObjects define references to other OsidObjects. An Asset, for example, defines a Provider that is represented as a Resource. Assets may also relate through optional sessions to various other objects, such as a Repository. It is useful to be able to assemble search queries across these related objects. The result is an OsidQuery that joins the OsidQuery interfaces defined for other objects, or even in other OSIDs.

The complexity this can introduce for an OSID Provider can be significant and it can also result in circular dependencies if taken too far. To mitigate this, an OsidQuery specifies all

joined terms as optional. However, an implementation of some of this functionality can provide rich query functionality using the interoperability of the core specifications.

Below are two examples of searching for Assets by Provider.

```
/* get assets whose provider has the specified Id */
AssetQuery assetQuery = searchSession.getAssetQuery();
assetQuery.matchProviderId(houghtonMifflinProviderId, true);
AssetList assets = searchSession.getAssetsByQuery(assetQuery);


/* get assets by the provider's display name */
assetQuery = searchSession.getAssetQuery();
if (assetQuery.supportsResourceQuery()) {
    ResourceQuery resourceQuery = assetQuery.getProviderQuery();
    resourceQuery.matchDisplayName("Houghton Mifflin", plainStringMatchType, true);
}
assets = searchSession.getAssetsByQuery(assetQuery);
```

Code examples of searching for Assets of a given Provider.

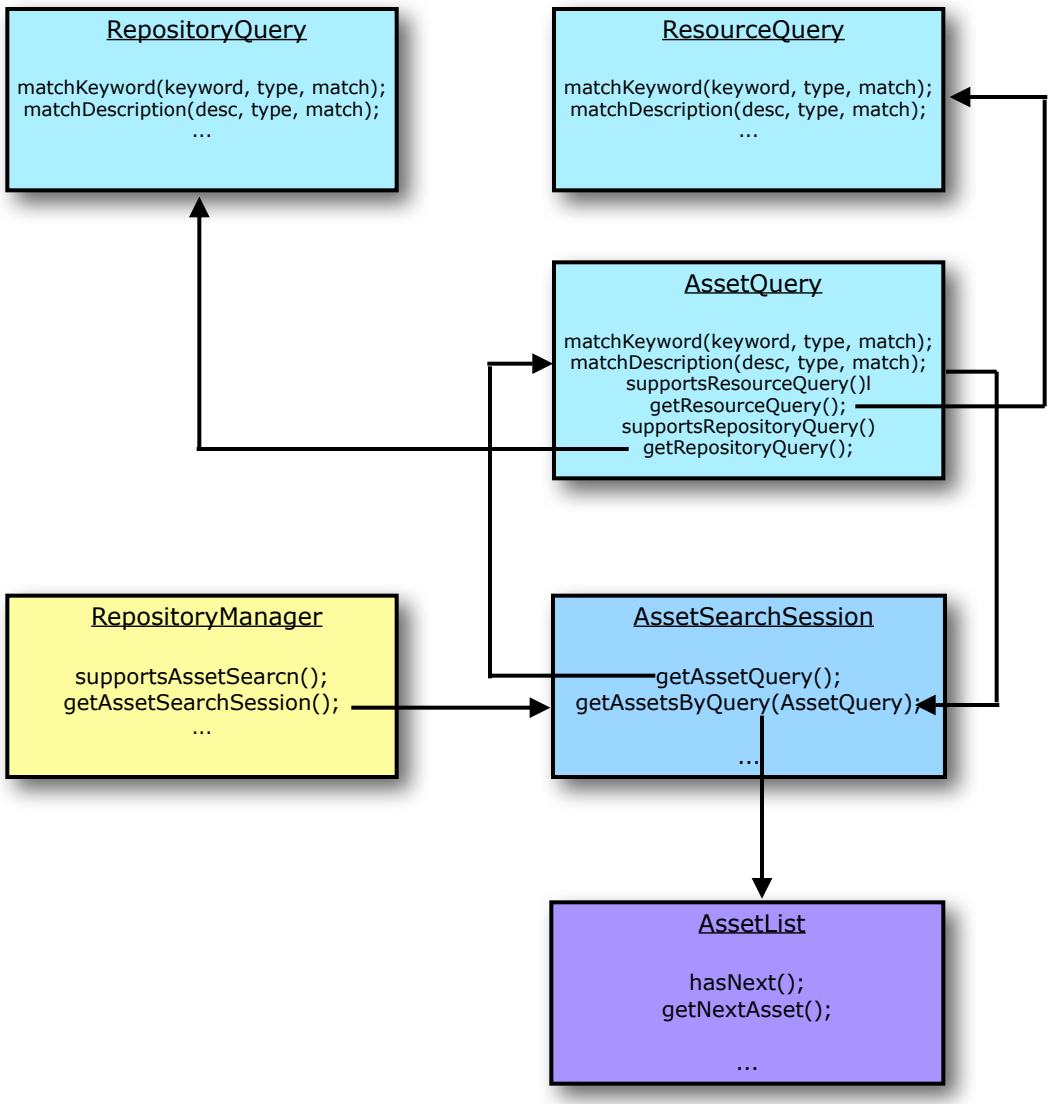
In the first query, the OSID Consumer has knowledge of a particular Resource Id. This type of query can be performed through a simple match. If the OSID Consumer wishes to perform a fuzzier query without knowledge of an Id, a ResourceQuery interface can be accessed directly via the AssetQuery interface if the OSID Provider supports one. While the AssetQuery is submitted into the search session to initiate the search transaction, the method to retrieve the ResourceQuery appends the query term to the AssetQuery and can be used without having to pass it back. The AssetQuery implementation is responsible for keeping its state.

The boolean behavior is the same as the other match methods in the query. The ResourceQuery term is AND'd with any other terms specified in the AssetQuery. Multiple retrievals of a ResourceQuery on the provider method are OR'd.

```
DisplayName AND (Description1 OR Description2) AND (ProviderQuery1 OR ProviderQuery2)
```

The same boolean operator rules apply inside the Provider ResourceQuery.

If more than one AssetQuery is submitted to the search session, these are also OR'd. This is the functional equivalent of performing separate queries and concatenating the results.



The control flow for an asset search joining terms for repository and provider.

Advanced Search Patterns

So far, the OSID searching patterns described the assembly of query terms. The other consideration is the ability to govern the entire search. An OsidSearch interface can be submitted alongside an OsidQuery array to manage the search.

A common search option is to instruct the OSID Provider to limit the number of search results.

```
AssetQuery query = searchSession.getAssetQuery();
query.matchDisplayName("food", stringMatchType, true);

AssetSearch search = searchSession.getAssetSearch();
search.limitResultSet(101, 200);
AssetSearchResults results = searchSession.getAssetBySearch(query, search);

AssetList assets = results.getAssets();
```

Code example of performing a search of food requesting results 101 through 200.

The getAssetsBySearch() method is the full monty. It also provides the means for ordering results, estimating number of hits, and searching within a result set. An OsidSearchResults interface that is returned represents the result of a search that can contain results other than a list of objects. Here are some examples:

```
AssetQuery query = searchSession.getAssetQuery();
query.matchDisplayName("food", stringMatchType, true);

AssetSearch search = searchSession.getAssetSearch();
AssetSearchResults results = searchSession.getAssetBySearch(query, search);

query = searchSession.getAssetQuery();
query.matchDescription("pizza", wordStringMatchType, true);

AssetSearch search2 = searchSession.getAssetSearch();
search2.searchWithinAssetResults(search);
results = searchSession.getAssetsBySearch(query, search2);
AssetList assets = results.getAssets();
```

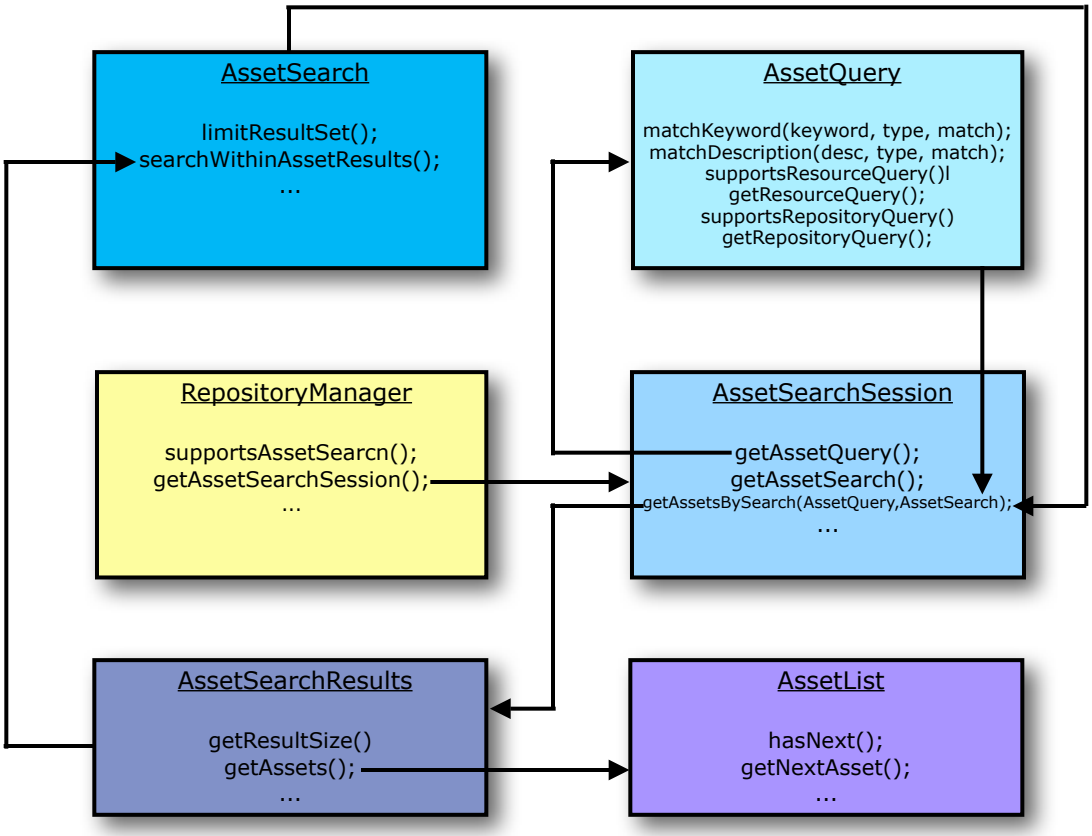
Code example of performing a search within a search.

The first result set is a handle to the results of the first search. A second AssetSearch is retrieved that accepts the results of the first search. Also note that a new AssetQuery also needs to be retrieved. These interfaces cover objects that implement state and can be used only once.

```
AssetQuery query = searchSession.getAssetQuery();
query.matchDisplayName("food", stringMatchType, true);

AssetSearch search = searchSession.getAssetSearch();
search.limitResultSet(1, 10);
AssetSearchResults results = searchSession.getAssetBySearch(query, search);

print ("Results 1 - 10 of about " + results.getResultSize() + " for food.");
```

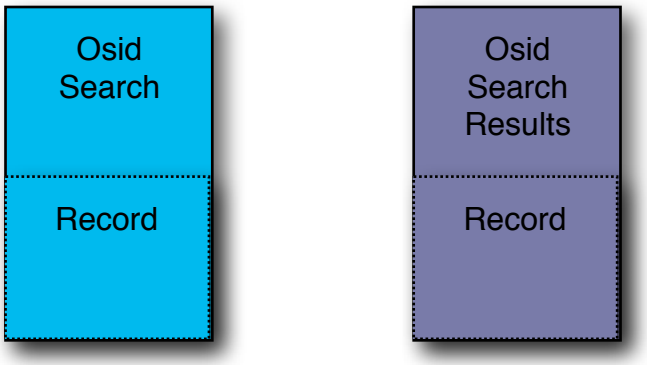
Code example of estimating the number of search hits.

An OsidSearchResults also provides a method for getting a size estimate back from the search (the output line may look familiar). The caveat here is that the result size may have little to do with the number of assets available in the returned list. In this case, the number of assets are limited to the first ten, but it is helpful for a user to see an estimate of the quality of the overall search. Even in popular search engines, this number is never accurate and should never be used as input to memory allocation.



The control flow for an asset search using the AssetSearch and AssetSearchResults interfaces.

Finally, ordering of results may be specified using an OsidOrder interface. The OsidOrder interfaces parallel the OsidObject and OsidQuery interfaces and their record interfaces are identified with the same record Type.



A search record Type specifies the interfaces available across this set of core OSID interfaces.

```
AssetQuery query = searchSession.getAssetQuery();
query.matchDisplayName("food", stringMatchType, true);

AssetSearch search = searchSession.getAssetSearch();
AssetOrder order = searchSession.getAssetOrder();
order.orderByTitle();
order.orderByProvider();
search.orderAssetResults(order);

AssetSearchResults results = searchSession.getAssetBySearch(query, search);
```

Code example of ordering the Asset search results first by title then by provider.

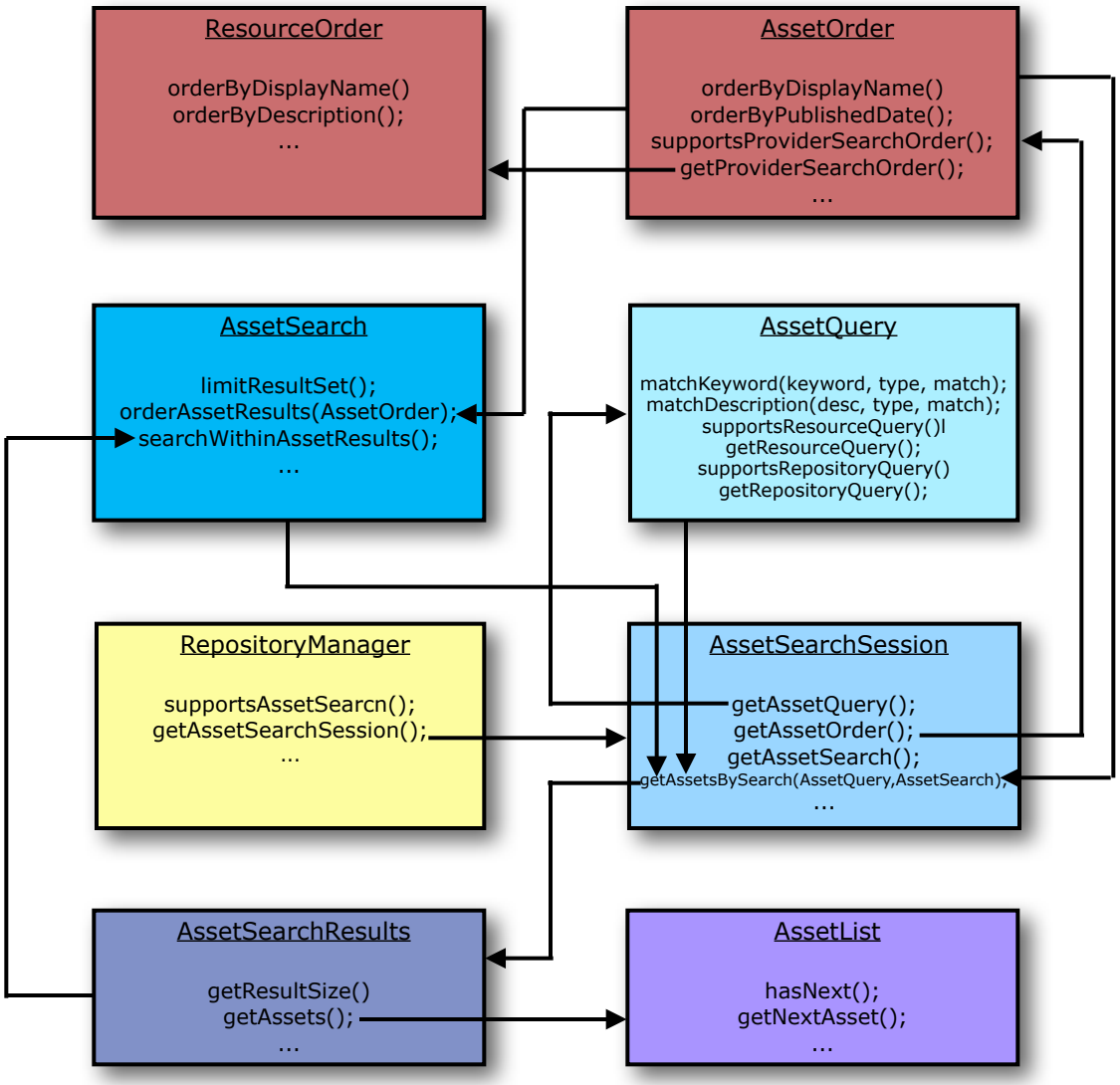
```
AssetQuery query = searchSession.getAssetQuery();
query.matchDisplayName("food", stringMatchType, true);

AssetSearch search = searchSession.getAssetSearch();
AssetOrder assetOrder = searchSession.getAssetOrder();
assetOrder.orderByTitle();

if (assetOrder.supportsProviderOrder()) {
    ResourceOrder providerOrder = assetOrder.getProviderOrder();
    providerOrder.orderByDisplayName();
}

search.orderAssetResults(assetOrder);

AssetSearchResults results = searchSession.getAssetBySearch(query, search);
```

Code example of ordering the Asset search results first by title then by the provider's display name..

Ordering can be daisy chained in a similar manner to the query. In the first ordering example, the OSID Consumer requested an ordering by provider but didn't specify what

in the provider to sort. In the second example, the OSID Consumer specified the display name of the provider.

Not all such relationships are available through the ordering interfaces. An ordering relationship is available if there is a one-to-one relationship between the two objects. Since an asset may be mapped to multiple repositories, there is no way to specify an ordering of asset results by repository.



The control flow for an asset search using the AssetSearch, AssetOrder and AssetSearchResults interfaces. If records or additional joins are added, this diagram might get a little complicated.

The OsidSearch and the OsidSearchResults work in tandem to govern a search independent of the query terms. They may also contain records that further convey details of the search, perhaps the time it took for the search to complete. These search record types are independent of the object record types and must be negotiated separately through an OsidProfile.

8 Notifications

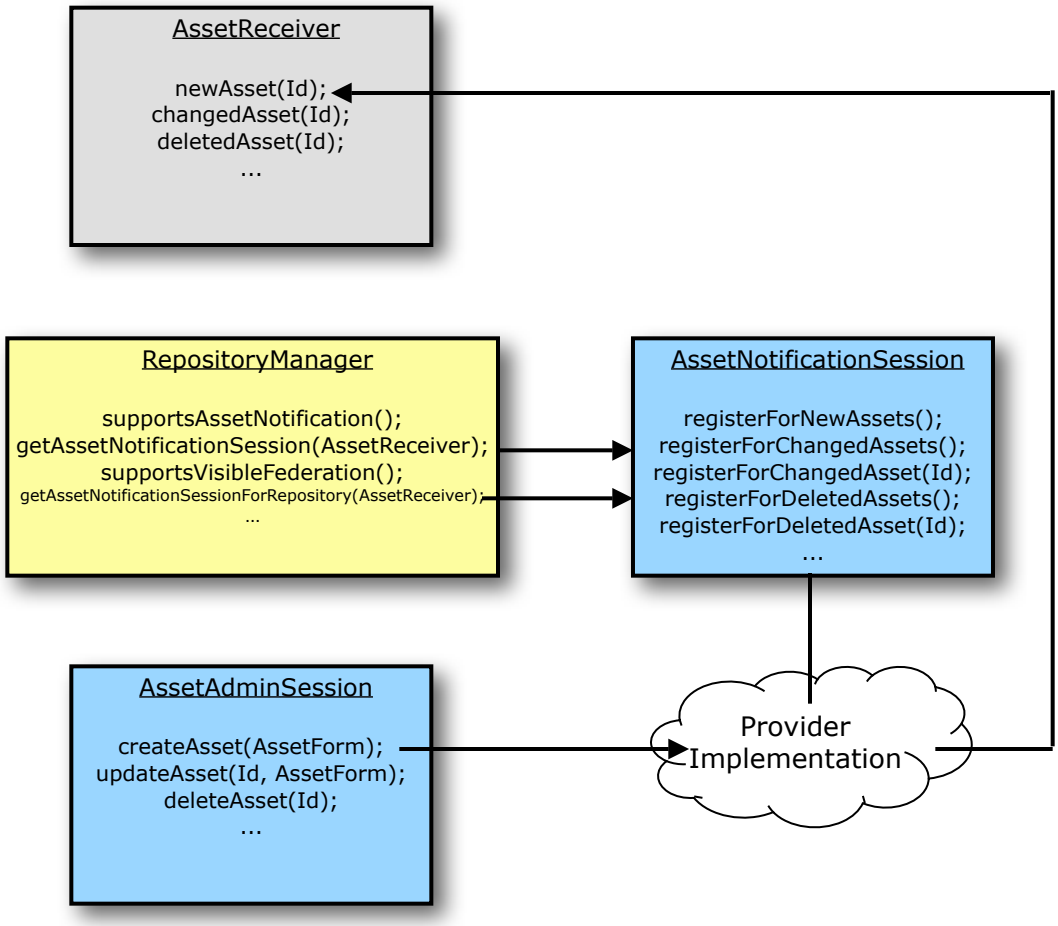
OSID Consumers may hold onto objects for display purposes however any changes to the objects are not reflected in the application unless the application refreshes the data at some predetermined interval. The notification sessions provide an asynchronous means to acquire real-time data.

Caching in various forms is sometimes an important tool however there's no way of knowing when the object cached is changed without the use of polling. It is desirable to modify a OSID Provider's behavior using layered OSID Adapters. The notification sessions provide the means in which an OSID Provider can more efficiently enable the creation of such OSID Adapters. A simple polling mechanism may be implemented under the notification service where the polling parameters are managed by the OSID Provider or a more scalable enterprise service bus can be utilized.

Another scenario is where an OSID Provider does not provide real-time data within its objects but a third party OSID Adapter can access a notification mechanism for the same data, such as a service bus. A real-time OSID Adapter can be layered upon an OSID Provider to provide both real-time data and a notification service.

The notification mechanism is implemented using a callback mechanism through the OSID. An `OsidReceiver` interface is specified for the particular `OsidObject` where notifications are defined. The OSID Consumer gives its receiver implementation to the notification session where it can then register for various service events. The events tend to be defined as new, updated or deleted objects. The OSID Provider will invoke methods in the `OsidReceiver` corresponding to these events.

The Messaging OSID can be used to factor out the notification aspect.



An AssetReceiver interface is implemented by the consumer and given to an AssetNotificationSession where the OSID Consumer can subscribe to notifications pertaining to various service events.

9 Session Controls

Pre-Authorizations

A series of pre-authorization methods are defined in V3 to assist applications in presenting a user interface that is more tailored to what the user is authorized to do. This pattern is to help avoid the problem of an application displaying every conceivable function of a service only to report on all the exceptions that may result. The pre-authorizations are expressed as methods such as canAccessAssets(), canCreateAssets() and canDeleteAsset(Id). A return of true does not guarantee that the corresponding methods will succeed but only that a return of false can be interpreted as *don't bother*. An OSID Provider that has no means of determining a pre-authorization will simply return true and defer the authorization check.

Pre-authorization is also useful in not wasting the user's time in creating an asset only to discover that it cannot be persisted.

Views

OsidSessions generally define views to instruct the OSID Provider on the desired result set. Views are managed as toggles within a single session. One set of views commonly found describes the behavior of a federated catalog. When an OSID Consumer is performing lookups, it generally makes sense to look at the federated view of the associated session catalog. Although an asset might not be defined within the current repository, for example, the OSID Provider should fetch the asset in any child of the repository. This is referred to as a *federated view*.

However, when managing the mappings between assets and repositories, the federated view produces false indicators of what has been explicitly defined in a repository. The OSID Consumer has the option of choosing an *isolated view* to compensate for this behavior.

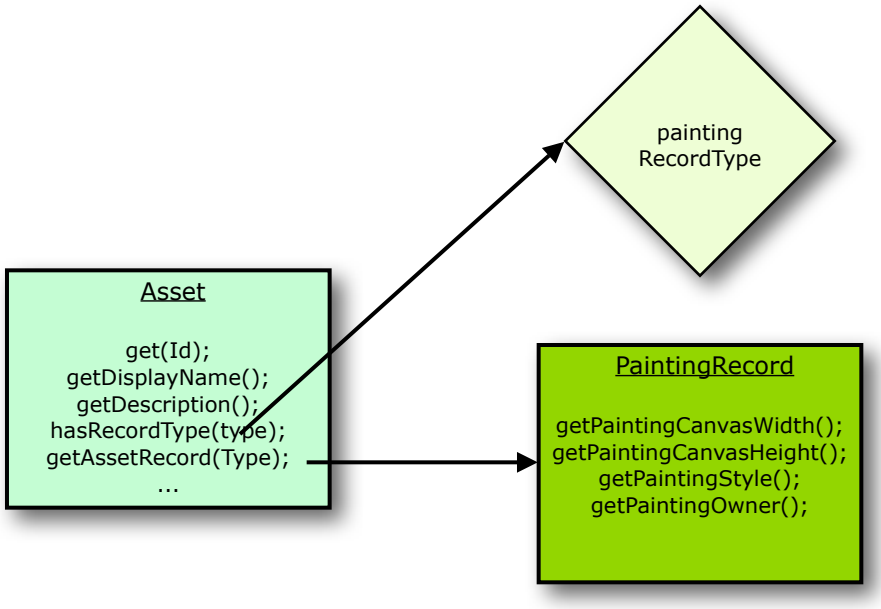
Another issue relates to what the OSID Provider should do when retrieving a set of OSID objects whether by Id, Type or some other lookup method. If the user is not authorized to see one or more objects in the resulting set, the OSID Provider may either return a PERMISSION_DENIED error or it may simply omit that object from the resulting set. The latter option increases interoperability but is not desirable in cases where accuracy is required, such as synchronizing data. This behavior can also be managed with another pair of views in the lookup sessions.

A *comparative view* instructs the OSID Provider that interoperability is more important than accuracy and the OSID Provider may omit objects from a returned list. A *plenary view* instructs the OSID Provider that accuracy is more important in which cases an error should be returned instead of an incomplete list.

The search interfaces assume a comparative view.

10 OSID Records

An OsidObject may include zero or more OsidRecords. OsidRecords are a means of adding information to an OsidObject. Each OsidRecord is in itself an interface whose specification is identified by a Type. In V2, types were used to describe either a set of properties or a kind of serializable object (where a Java String also implements Serializable). V3 scopes a record to an interface that implements an OsidRecord interface.



An example Asset defining a painting record.

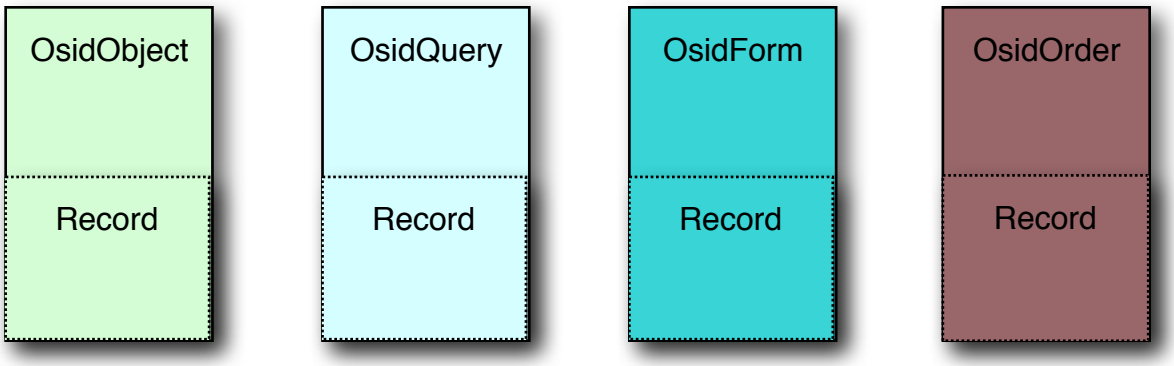
In the above example, an Asset supporting the paintingRecordType can deliver an object that implements the PaintingRecord interface. Although Types are used to describe other kinds of agreements in the OSIDs, the definition of an OsidRecord Type is the set of methods composing an OsidRecord interface.

From here on we'll favor the phrase *record type specification* in lieu of data or type agreement regarding data available in OsidObjects. Part of the function of O.K.I. will continue to be publishing record type interface specifications and now the language of these agreements is more clearly defined. They look like OSIDs.

In the painting example, getAssetRecord() is defined to return an AssetRecord interface. This is one of the only places in the OSIDs where a cast is used. The OSID Consumer has

asked the OSID Provider of Asset for a record that implements the paintingRecordType and the OSID Provider is required to return on if supported. It is safe for the OSID Consumer to cast the returned AssetRecord into a PaintingRecord interface.

An OsidObject's record Type is an extension of the core OSID Specification defining a set of methods that can be invoked by an OSID Consumer. This record Type also specifies the interface for the OsidQuery, OsidOrder, and OsidForm records. The OsidSearch and OsidSearchResults are not specified through the OsidObject's record Type. They are specified through their own search Type however the OSIDs allow these search Types to vary on an OsidObject by OsidObject basis.



An OsidObject's record Type specifies the interfaces available across this set of core OSID interfaces.

Genus Types

It might be tempting to use the record Type as a key for what the OsidObject represents. The OsidRecord is intended to be a specification of methods and the existence of an OsidRecord Type means the methods specified by the record Type must exist. An application might want to know the difference between a book and a magazine. Conjuring a record Type for each of book and magazine will impede interoperability when the methods can be defined within a PublishedMaterialRecord to handle any published material.

To avoid overloading record Types, all OsidObjects also define a *genus* Type. The genus Type is used for nothing except to convey a singular classification of an OsidObject without affecting the interoperability among record specifications. Genus Types can be queried and modified but there can only be one genus Type per OsidObject. A genus Type can be relevant where there is an *is a* relationship. More complex taxonomies should make use of catalogs.

11 Errors & Exceptions

Specification Errors and The Java Binding

Due to its Java heritage, V2 specifies exceptions. The V2 pattern is to define a single exception for each OSID with various types of errors defined as string constants housed within the exception. Because error handling varies widely across language platforms, V3 defines only the type of error that may result from a method invocation and delegates interpretation of the error syntax to the binding. V3 defines fewer errors although the context of an error may vary across methods.

For the current V3 Java binding in development, OSID errors are mapped directly to Java exceptions. An advantage of this binding is that an OSID Consumer can directly catch a `PERMISSION_DENIED` and handle authorizations independently from other exceptions without resorting to string matching. And because `OsidExceptions` are sub-classed from Exception, exception messages and chaining are available which can provide a better explanation for what went wrong. V2 didn't do this because passing messages from the OSID Provider (e.g.: *your printer is out of paper*) was seen as a violation of encapsulation.

Encapsulation

To understand the advantages and disadvantages of this binding scheme, a common understanding of encapsulation must be established. V3 defines encapsulation as protecting the OSID Provider implementation from the application code (although if the application casts then it's game over) where the application code is able to perform an action that ties it to a particular implementation through exposure of the internal workings of that implementation. It doesn't define encapsulation as hiding useful diagnostic information from an end-user and V2 has fallen short this front. One can argue that an application can parse *your printer is out of paper,* in whatever language it is localized in, and take some action. This author says if you can pull that off then you have earned your get out of jail free card.

Another encapsulation issue related to this exception scheme is seen in layering OSIDs. In V2 a Repository OSID Provider would have to explicitly catch and re-throw exceptions thrown from an underlying Authorization OSID. Some consider this to be a Java anti-pattern because, frankly, it is a pain in the neck. In this V3 binding, if the error types are shared in both the Repository and Authorization method definitions, which they usually are, then Repository has the option of letting the Authorization exception

sail through. The OsidException does not identify itself as coming from Authorization, so there's no *reasonable* programmatic way for the application to understand an Authorization OSID exists under Repository. It might be useful to instruct an end-user on what he needs to do to get authorized.

Java Runtime Exceptions

The other V3 binding twist is that some OSID errors are bound to Java runtime exceptions that do not need to be explicitly caught. These errors follow the Java paradigm of classifying errors between programming errors and other errors. OSID errors that should be avoided by an OSID Consumer through correct use of the methods such as NULL_ARGUMENT, UNIMPLEMENTED and ILLEGAL_STATE are bound to runtime exceptions and not require special handling at each layer other than to create a bug report.

Generally, an OSID Consumer should catch an exception if it is able to do something about it. Many OSID errors fall into the category of programmatic or integration related problems where an application may not handle it in a special way other than to say, *something doesn't work*. In Java, these are mapped to runtime exceptions that do not require an explicit declaration or catch. The remaining errors are user-oriented and communicate problems such as *not found* and *already exists*. The catch all remains OPERATION _FAILED.

Execution Flow

Errors are specified for methods where they are believed to make sense with sensitivity to the Java platform that any non-runtime exception method needs to be wrapped in a try-catch block. To ease the burden on the OSID Consumer, some methods do not specify any errors at all which may lead an OSID Provider to implement in such a way as to avoid the error. A Java OSID Provider may not throw an exception or runtime exception not defined in the interface (Java Errors resulting from the JVM are out of scope).

Error deferment is a technique that can be used when a method implementation fails but no error is defined. A case is the OSID list where in V3 the errors have been removed from the hasNext() method. If the implementation of the list loses contact with the incoming stream and cannot execute hasNext(), for example, the implementation should return true so the OSID Consumer will continue execution and return the error on the getNext() method.

The OSIDs define errors that are sensitive to the execution flow of the OSID Consumer by reducing the number of places where an interrupted may occur but remain general enough to cover a wide variety of cases. Not all errors defined may result from a given OSID Provider, but it is strongly recommended that errors at the programming level,

such as NULL_ARGUMENT and INVALID_STATE, always be honored by the OSID Provider in that it will aid in the interoperability for the OSID Consumer across a more diverse set of OSID Providers.

Errors and Method Contracts

Errors map to exceptions where available in the language binding and should be handled and caught where the OSID Consumer wishes to take some action based on an error condition. An error is used to indicate a failure of contract. A method returns the specified object or an error results. Nulls are not permitted as a third option.

An OSID Consumer should always have the option of using the interface in such a way to avoid dealing with errors directly in line with code execution. For example, getAsset(assetId) may result in a not found, authorization, or operation failure. In the not found case, the OSID Consumer has to deal with the error and may take an action that affects the logic of the code, such as selecting a new asset to retrieve. In the latter two there's nothing the OSID Consumer can do other than to report on the problem and move on.

New methods have been defined to assist in managing situations outside of the error mechanism. One example are the pre-authorizations that can be used to mitigate dependence on authorization errors. Another method is to use existence checks to remove reliance on not found errors.

```
try {
    try {
        asset = session.getAsset(assetId);
    } catch (NotFoundException nfe) {
        go back and try another Id;
    }
} catch (OsidException oe);
    log(oe);
}
```

can become:

```
try {
    if (!session.assetExists(assetId)) {
        assetId = another Id;
    }
    asset = session.getAsset(assetId);
} catch (OsidException oe) {
    log(oe);
}
```

OSID Objects

Generally, error definitions have been removed for simple data access from an `OsidObject`. `OsidObjects` have the `Id` available. `getId()` should not result in an error since the `Id` does not change. The picture gets less clear for the other data retrieval methods whose data may change and the OSID Provider may wish to provide real-time access.

The OSID Consumer is in the best position to decide whether it is interested in real-time data. Forcing the OSID Consumer to catch errors in updating data when it has no interest is unproductive. Yet, other consumers may rely on real-time data. Let's first look at the ways in which an OSID Provider might implement an OSID object.

1. The OSID Provider supplies all the data upon retrieval. The retrieval method results in an error if the data cannot be accessed. The data is stale.

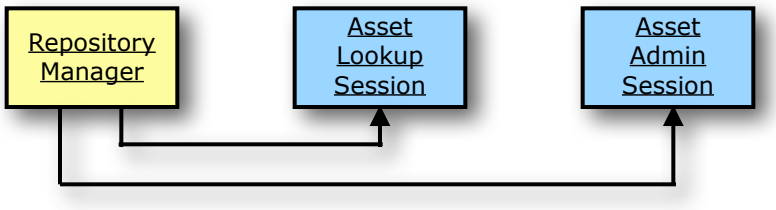
2. The OSID Provider supplies only the `Id` upon retrieval and each method fetches the corresponding data element. The data is fresh but is not available if something fails.

3. The OSID Provider supplies all the data upon retrieval and updates the data when it is changed through the notification or some other mechanism. The data is fresh and of an update fails, the data stales silently.

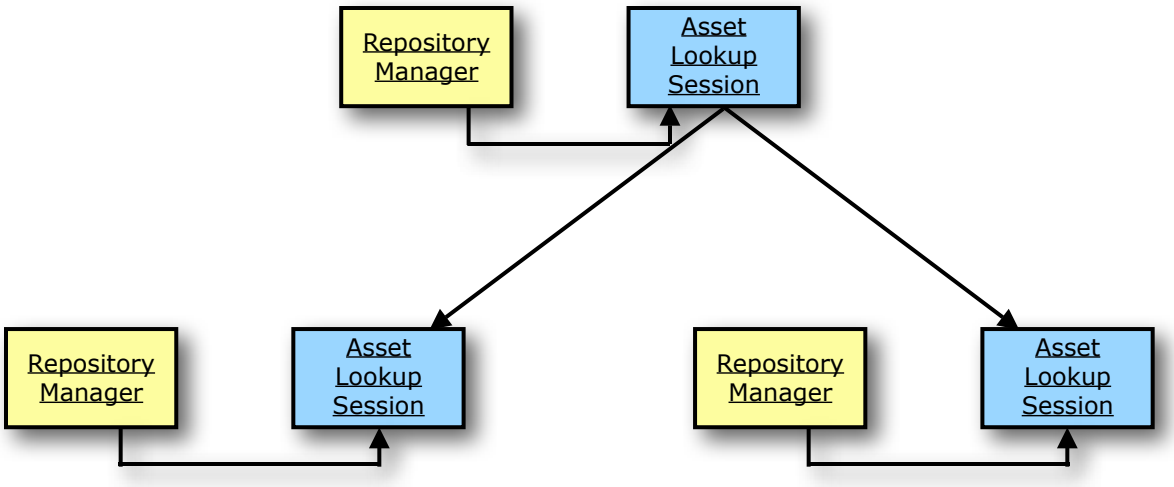
Many of today's applications implement (1) although (3) would appear to be the more efficient option. In either of case (2) or (3), some way should exist to let the discriminating OSID Consumer know that the data is stale. The method `isCurrent()` exists in the `OsidObject` to test for stale data that can be used or ignored according to the OSID Consumer's needs. This method is used to indicate any potential staleness in the OSID object which includes the case where the OSID Provider is operational but has no means of keeping the data current.

12 OSID Adapters

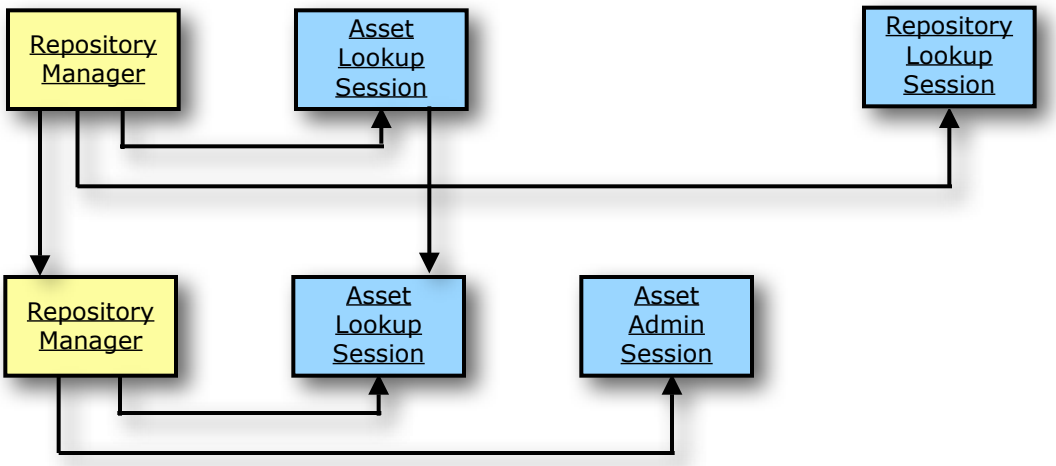
An OSID Adapter pattern involves the layering of an OSID Provider on top of another to tailor a service for a particular OSID Consumer or family of OSID Consumers. In V3, this technique may be designed in a more structured manner through the session model. Let's start with a Repository OSID Provider that implements an AssetLookupSession and an AssetAdminSession and work out a few possibilities from there.



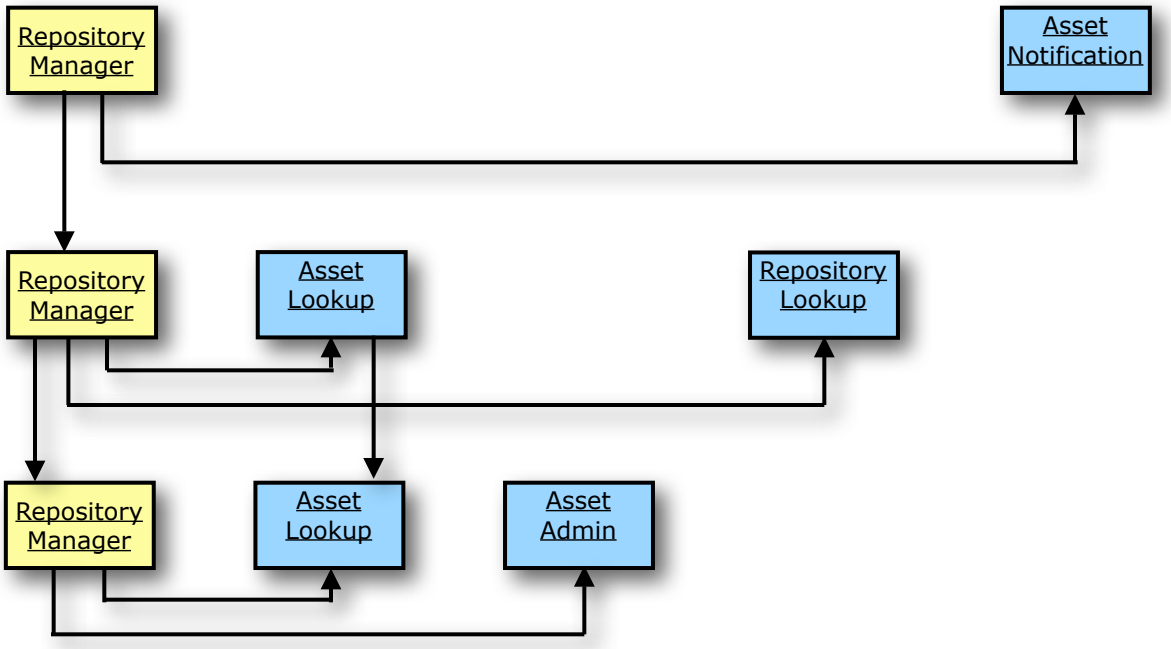
A basic Repository OSID.



A federating pattern for combining Repository OSID Providers.

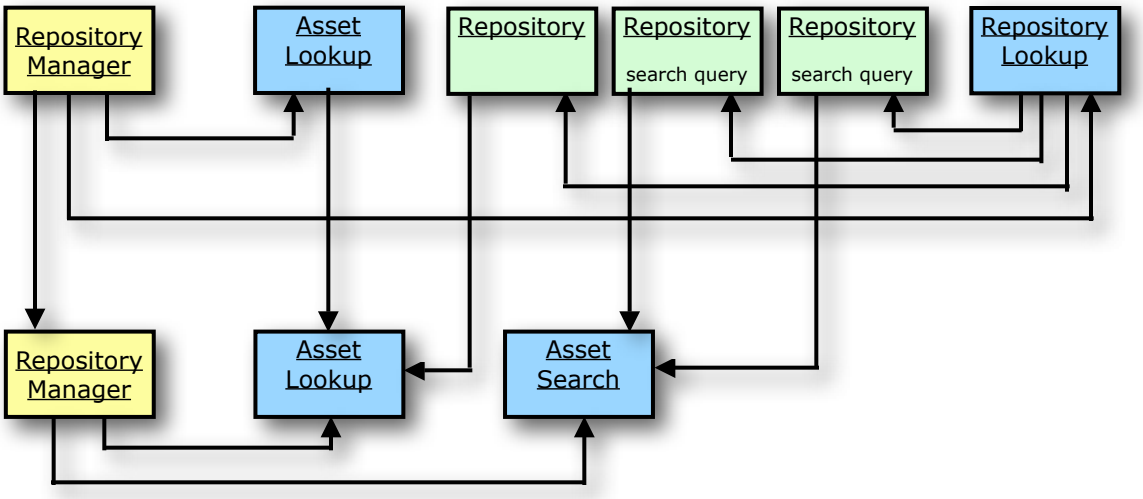


Adding a Repository service to an existing Repository OSID Provider. The OSID Adapter maintains its own Asset–Repository mappings and adapts the lookup the manager and lookup session



A notification service is added to the previous Repository OSID Provider. The notification OSID Adapter can implement a listener to an information bus, communicate with the backend Repository server directly, or implement a simply polling mechanism (yes, polling is bad, which is why one might want it contained in a managed layer).

Many more OSID Adapter patterns are such as translating or adding new types, caching object retrievals, fixing bad implementations, offering different organizations of objects, etc. Here is another example using an OsidCatalog:



A catalog OSID Adapter. A set of Repository objects are created in the adapter. One Repository maps to a lookup of Assets in the underlying OSID. The other two Repository objects represent queries to be performed in the underlying search session. For example, a Repository of Picasso's can be created by searching for Picasso-related assets to provide a simplified lookup service for an application by encapsulating the more complex query.

13 Agent & Authentication

Authentication Process

A strongly perceived problem in V2 was the Authentication OSID and its relationship to Agent. Some of the issues stem from the fact that Authentication is a multi-service process and incongruous to the other service-oriented OSIDs that also requires an implementation alignment with the Agent OSID.

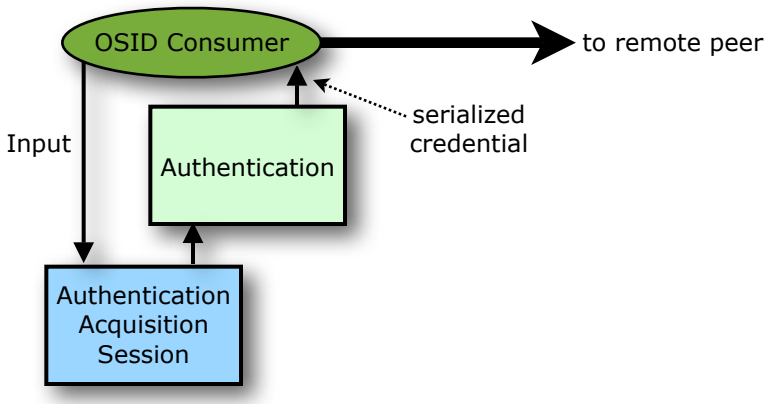
The steps of an authentication process are:

- Acquire a client authentication credential.
- Transport the credential to a remote peer.
- Validate the credential and determining the identity.

V3 divides the Authentication OSID into two aspects represented by sessions

- authentication acquisition
- authentication validation

and defines an Authentication object (not an OsidObject) to encapsulate the credential. Transporting a credential is the responsibility of the OSID Consumer since a credential will be serialized, in some format, and embedded in some application protocol. The OSID only defines the touch points of this process.



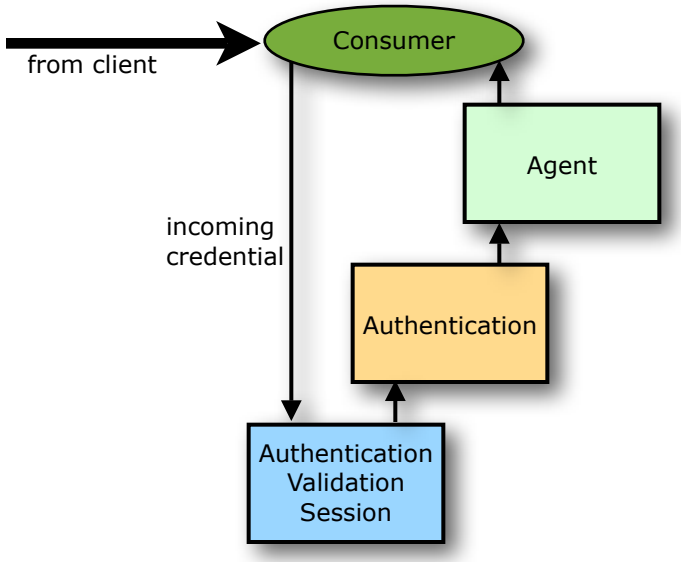
The authentication acquisition process.

The above diagram shows the general authentication acquisition process which is used by a client-side application to gather authentication credentials in some agreed upon format and send them to a remote peer for validation. Ideally, the specifics of the authentication technology should be encapsulated within the Authentication OSID. A username/password pair can be considered technology specific.

Yet the diagram shows an input. The input mechanism was designed for challenge-response systems where a credential is generated in response to a challenge from a remote peer. With a different type agreement, it *can* be used for a username and password such that the OSID Consumer is tied to username/password based Authentication OSID Providers.

In an enterprise environment, applications that prompt for passwords that are not the approved single sign-on mechanism are frowned upon. In an integrated desktop environment, the operating system handles login and provides means to reduce the number of times a user needs to be prompted for a password (Apple's Keychain, for example). However, in the interest of neutrality, the specification allows for a variety of possibilities applying the authentication input.

Another consideration is a Login component, where the component itself is coupled to the Authentication OSID Provider and the application remains uncoupled from the component. See Components.



The authentication validation process.

The validation process is similar to the acquisition process where a type agreement specifies the format of the incoming credential retrieved from within an application

protocol. The Authentication OSID returns a representation of that credential upon validation which can reveal an Agent identity.

Agent

The agent identity is available from the validated authentication credential. In V2, Agent was used as a resource identifier that not only was used for identifying an authenticated entity for the Authorization OSID, but was also used to identify an entity that could receive a message or be scheduled on a calendar. V3 factors these problems into distinct OSIDs.

- Agent: is used to identify an authenticate-able entity for the Authorization OSID. Agent is referenced in authorization and any other place throughout the OSIDs where a method is available to describe: *who* did what.

- Resource: is used to separate the information about a person, place or thing from its authentication counterpart. Resources are used throughout the OSIDs where a method is available to describe: did what to *whom*. Decoupling Resource from Agent also permits a Resource to map to multiple authentication identities.

In Messaging, for example, a commitment reads Agent *sends a message to* Resource. The Agent OSID maps only to an authentication credential. Consequently, Agent is merged into the Authentication OSID and now one can argue that it now has service status. The Group portion of Agent has been moved to Resource.

14 Fun With Repository

Core Asset

The core V3 Asset defines a richer interface to capture common data commonly found in digital content.

- getTitle(): the proper title may or may not be the same as getDisplayName()
- getProvider(): a Resource to identify a publisher or distributor
- getSource(): a Resource to identify the source or origin
- getCreatedDate(): gets the date when the asset (not necessarily the object in the digital repository) was created

Along with these methods, several relationships can be constructed to convey more information about the Asset. These are described in the following sections.

The Meaning of Asset

The structure of V3 Asset shifts around from what was defined in V2. What was delivered through Records and Parts (like an abstract database) is now be specified as OsidRecords, AssetContent and methods. This is hopefully a simplification. It may also be obvious that all the Asset examples in this paper appear to have a bias toward digital media.

What is a Repository Asset? An asset represents something in digital form. Without some digital representation it is simply a data store. Looking back at the Wartime asset series, the OSID assets displayed each represent an image as part of a collection of war related items. It isn't necessarily the *war*.

How an Asset is defined is sometimes in the eye of the definer. Consider these possibilities:

- Picasso's painting, *Le Singe*, is located in the Museum of Fine Arts. An Asset in the museum's Repository contains a photograph of the painting. The museum uses the Repository to track its painting inventory and stores a single reference

image per painting. The museum considers the Asset represent the painting and it contains the image plus the dimensions of the canvas.

- A photograph of *Le Singe* was taken by another photographer. In the photographer's repository, the photographer considers the photograph to be the Asset and it contains the image plus the shutter speed and frame number.

- The photographer produces each of a large, medium and thumbnail version of the photograph of *Le Singe*. The photographer considers these to be digital representations of the same asset because they were created from the same photographic image.

- The next day, the photographer takes another identical photograph of *Le Singe*. He considers this one to be a separate Asset from his previous photograph of the painting. He then gives it to the MFA to use in their painting inventory repository.

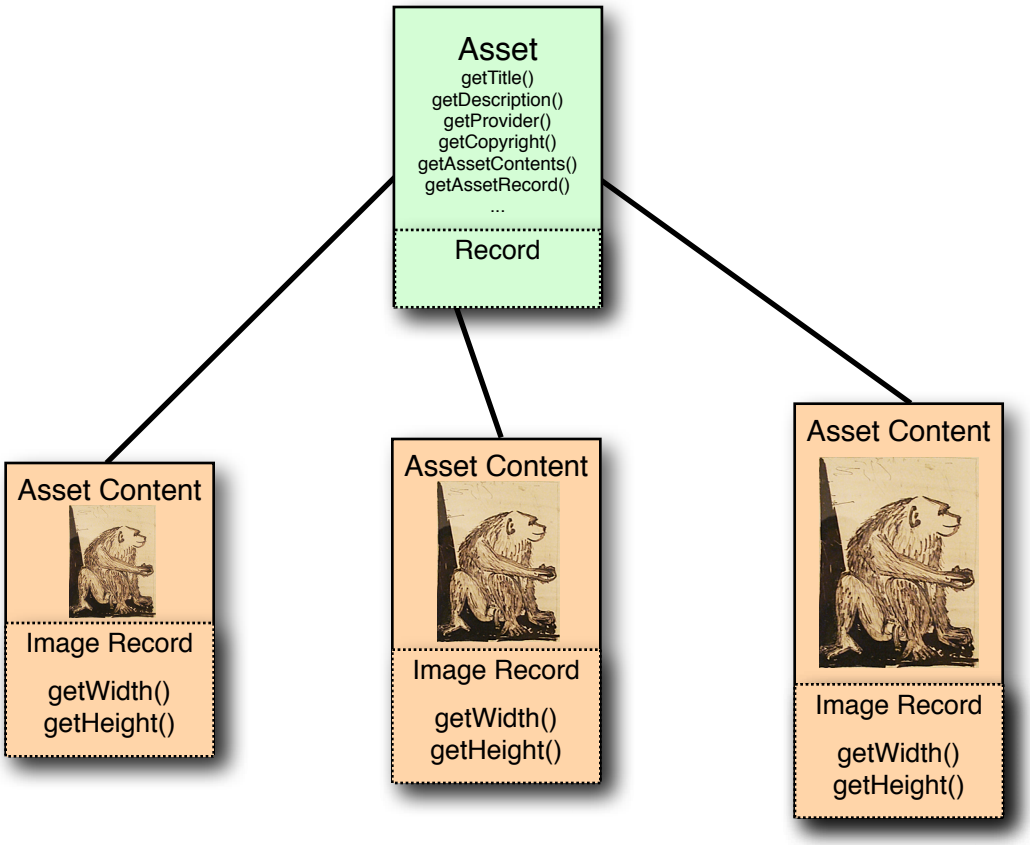
To begin sorting this mess out, first the distinction has to be made between the Asset and the various digital means the content may be conveyed.

Asset Content

An Asset represents something in digital form. The digital form is expressed through the AssetContent interface. AssetContent hangs off the Asset and captures information about the digital content including its data. An Asset may have zero of more AssetContents.

The AssetContent is identified with a Type. The Type may indicate that the content represents an image, video, audio, a document, or something more specific. It is used specifically to achieve interoperability around the format of the media while allowing the OSID Consumer to select the desired media.

For example, an image may be delivered in a large size, a smaller size suitable for a web page, or a thumbnail suitable for displaying many image samples. There can be an AssetContent for each image related to the same Asset.



Multiple forms of content can be associated with an Asset.

Other examples of using multiple content may include supporting multiple formats, providing varying quality, or enabling different types of accessibility. The image record can dive deeper and describe the data format in more detail including issues of compression, processing or codecs.

This leaves the AssetRecord to describe something about what the Asset represents since the actual data is captured by the AssetContent. For example, if the Asset was an image of *Le Singe*, then perhaps the AssetRecord could describe the actual painting itself. If a second photograph of *Le Singe* were taken, then there are a couple of possibilities.

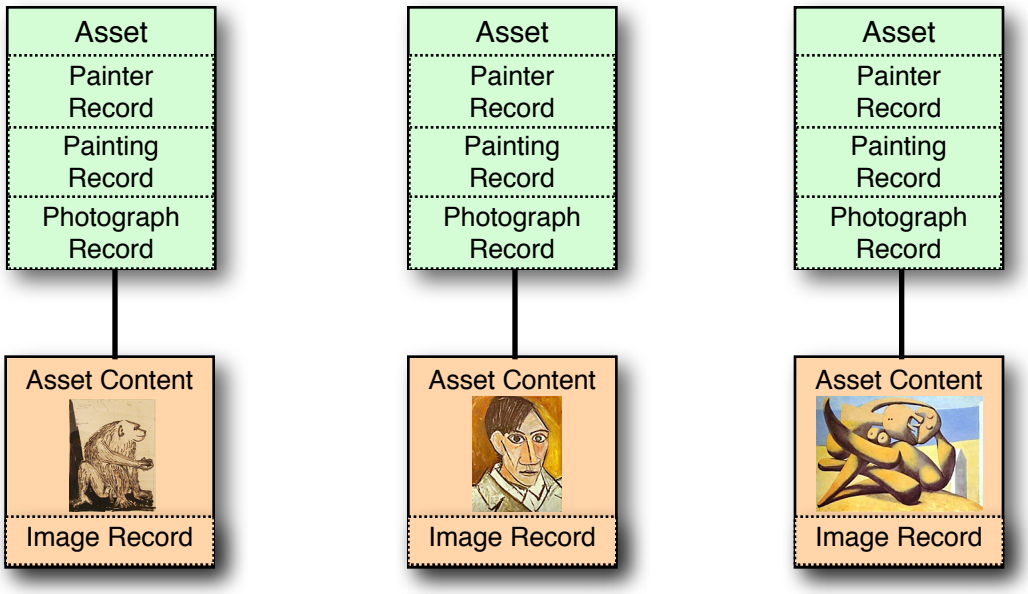
- Store the image of the second photograph as an additional AssetContent. Data about the painting is kept in a single asset. The downside is that if there is different core Asset data for each photograph, such as the photographer or copyright, then this could be problematic.

- Create a separate Asset for the second photograph. The downside here is that any record describing the painting is duplicated.

V3 provides a means for separating the asset from the subject the asset depicts.

Asset Credits

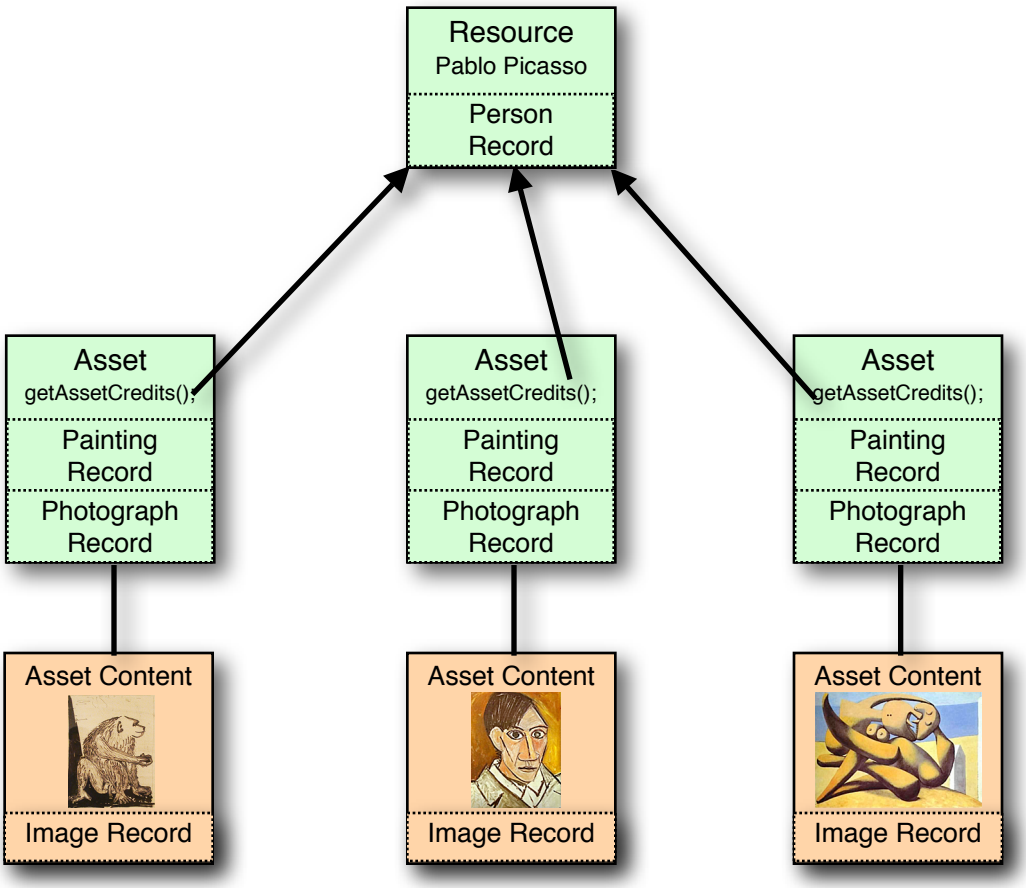
Take the following example of a Picasso repository. Each Asset in the repository is a photograph of a Picasso painting. A first pass design might define an OsidRecord for the painter, the painting, and perhaps data bout the photograph.



A repository of Assets with various records.

In this example, the painter record is the same among the three assets. This record might also contain biographical information about the painter, that needs to be replicated among the assets. OSID Consumers must have knowledge of the painter record Type to access this data.

The V3 Repository OSID defines an AssetCredit interface to relate a person, such as the creator, to the Asset. The person definition takes the form of a Resource interface. A Resource is a simple OsidObject used where an identifiable identity is desired. In this case a Resource with an OSID Id, a display name of *Pablo Picasso*, and his description can be implemented. Any specific data, such as date of birth or hair color, would be defined in a ResourceRecord.



AssetCredits are a way to relate people (Resources) to Assets.

Resources, like all OsidObjects, can be searched, created, updated and deleted. Because the Repository OSID defines a Resource relationship to Asset, Resource queries can be joined with Asset queries.

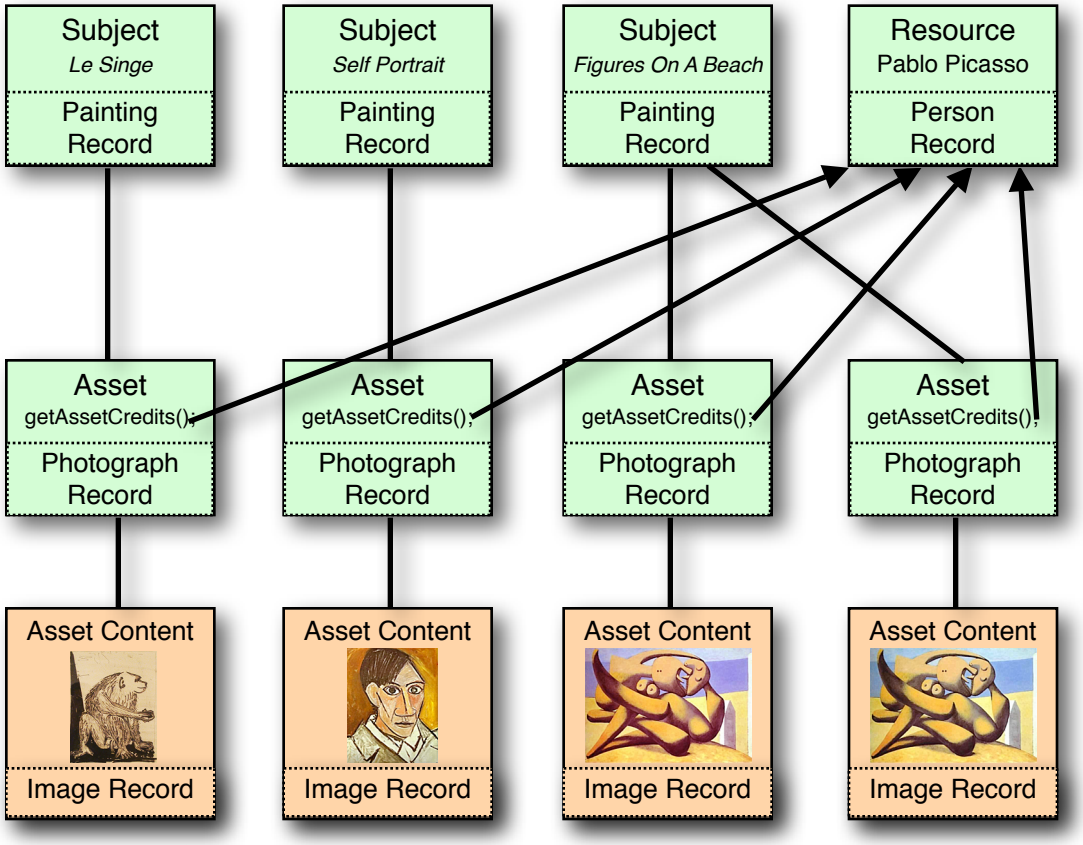
Another advantage of normalization is to increase interoperability. An OSID Consumer may have no knowledge of a painter record Type. In Resource, the display name and description is defined in core specification. Any OSID Consumer will be able to access it and be able to perform cross-references by OSID Id.

Subjects

Another photograph of *Figures On A Beach* was added to the repository. Now there are two painting records with duplicate information. Besides having maintain duplicate data, searches for paintings by Picasso will result in two search hits for *Figures On A Beach*. In a large repository, this is less than ideal. It would be much better to see a

unique search result for each Picasso painting, then once a painting is selected to see its variants.

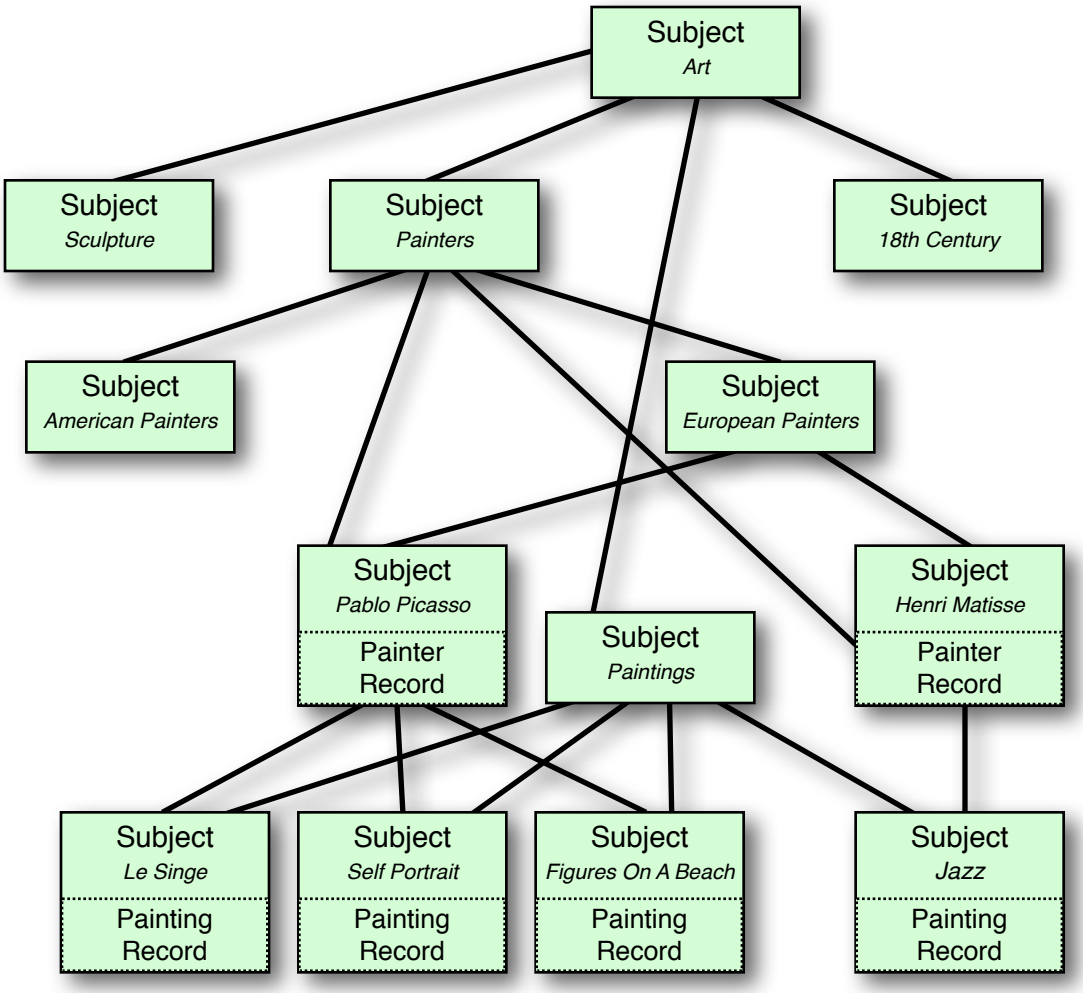
Subjects can be used to normalize subject matter apart from Assets.



Subjects are a way to relate Assets to subject matter.

Similar to Resources, the Subject can normalize data as well as promote interoperability by replacing property-stuffed records to Id-able objects.

Subjects are also hierarchical allowing OSID Consumers to traverse subject matter prior to retrieving Assets.



It's not a mess, it's a Subject Hierarchy.

As shown in the example above, Subjects can represent the painter as well as the painting. Earlier, the painter was represented in a Resource and a mapping through AssetCredit.

Searching for Assets

A search application may do the following to eliminate pages and pages of redundant results caused by having many versions of many assets on the same subject.

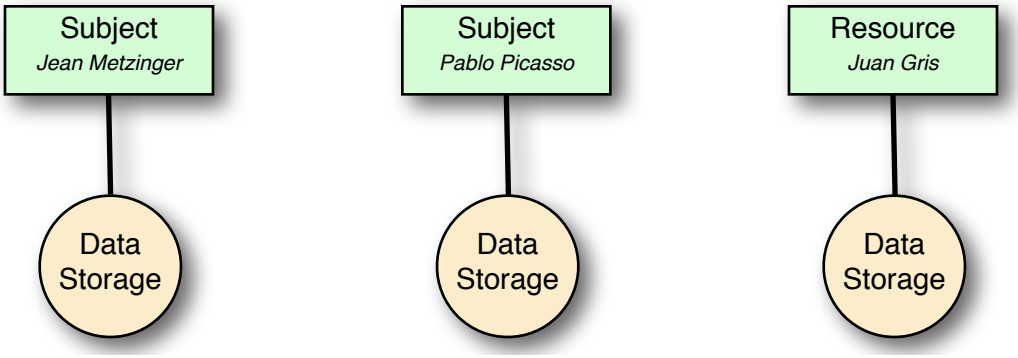
1. Perform a keyword search using the SubjectSearchSession to retrieve a list of pertinent Subjects.

2. Perform a lookup of the desired Subject(s) to generate a list of Assets. This list may be filtered based on Repository or provider.

3. Examine the available AssetContent for the selected Asset(s).

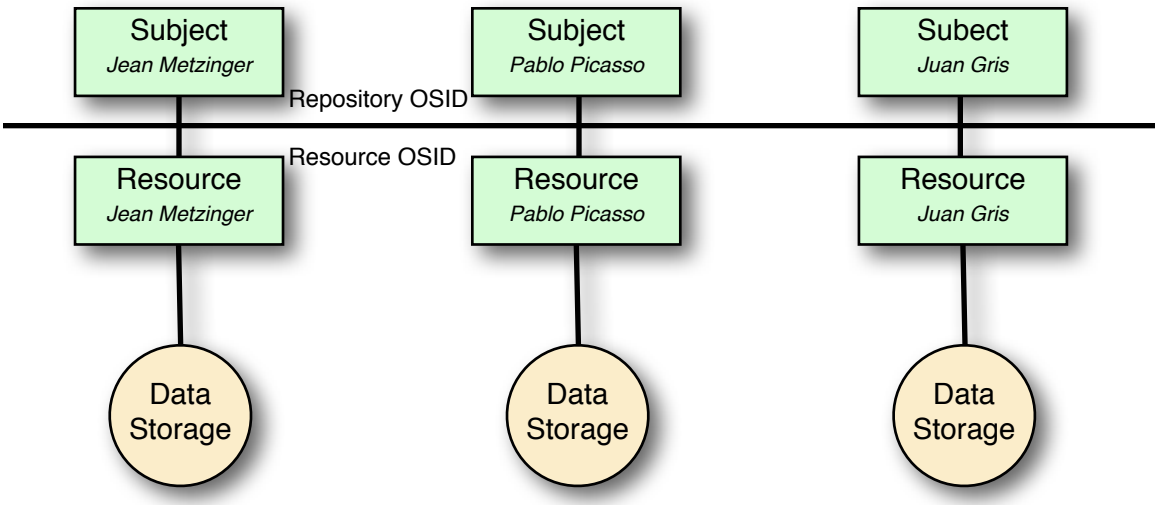
Delightful Ambiguities

An interface can be seen as a direct means of access to an underlying object implementation.



A basic interface design.

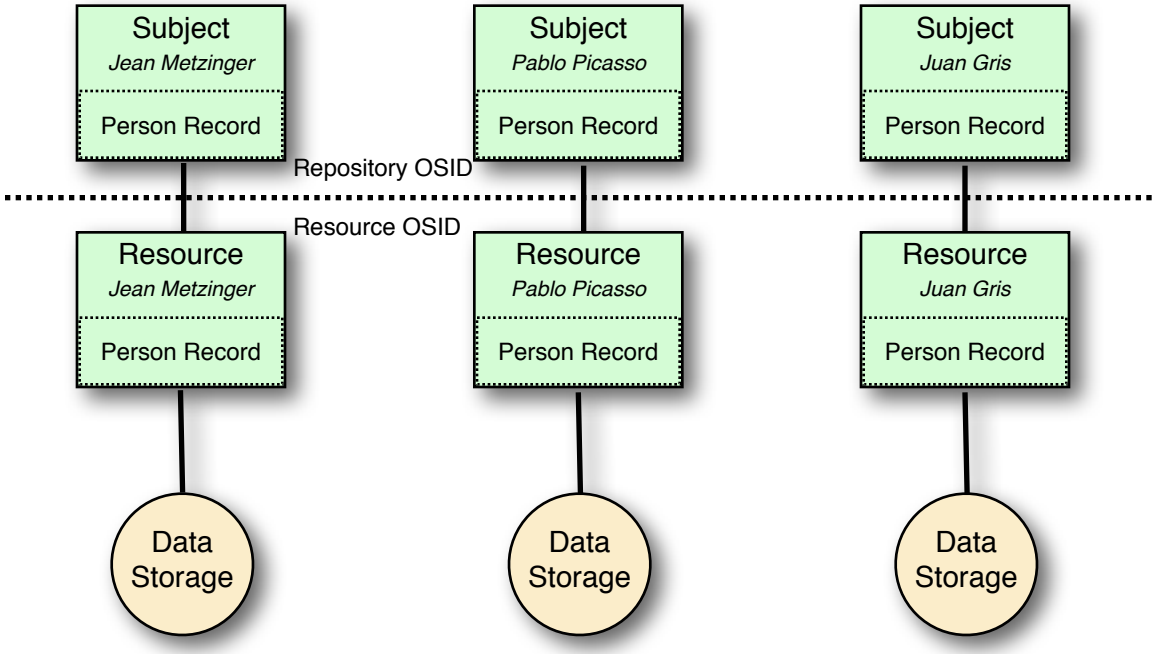
This can lead to questions about which interface to apply. In this case, there can be an ambiguity between the Resource and Subject interfaces. This may seem unfortunate since the Resource and Subject interfaces are almost identical. Then again, if they are so similar then they are easy to adapt.



Adapting Resources to Subjects.

An application geared toward searching a repository might examine Subjects. Another application geared toward managing directories of people may interact solely with the Resource OSID.

At times it may be warranted to provide an connection between the two OSIDs and this can be accomplished in one of two ways.



A Subject adapter floats the underlying Resource Record through a Subject Record to allow for the creation and update of Resources through the Subject interfaces.

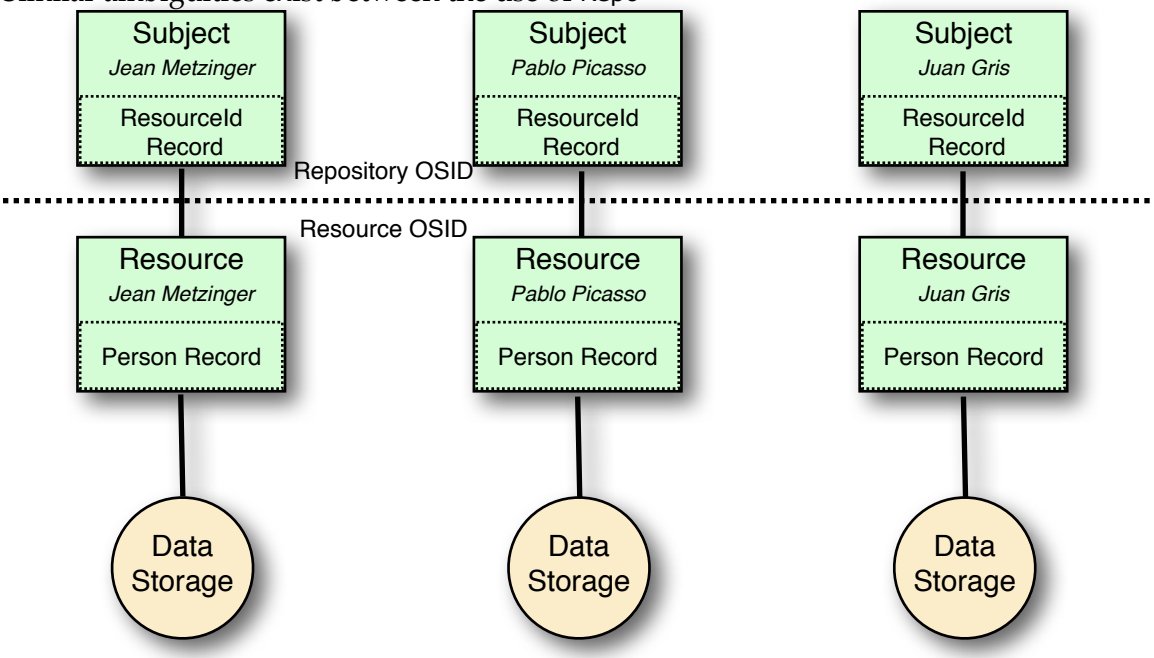
In the example above, the Subject contains all of the data found in the underlying Resource. Consequently, operations available in the ResourceAdminSession would be available in the SubjectAdminSession. The OSID Consumer of the Repository OSID would not be aware that a Resource OSID existed underneath.

Why bother with a Resource OSID? If, for example, a Repository OSID Provider wished to support musical content, it would have to figure out how to provide and/or maintain a directory of musicians. If such a directory were available, the Repository OSID Provider can federate the underlying Resource OSID and make the Resources identified throughout the entire federation immediately available to its own Assets and Subjects.

Another approach is to help the OSID Consumer orchestrate the Repository and Resource OSIDs.

The degree of one-stop shopping by combining OsidObjects needs careful consideration. While it is often tempting to overload an application to perform a wide variety of tasks, an application (or application module) geared toward a specific purpose, such as managing a directory of people vs. managing digital content, can be easier to use and develop.

Similar ambiguities exist between the use of Repo



A Subject Adapter floats the Resource Id through the Subject so the Resource can be accessed using a Resource OSID directly. Another variant is to simply use the Resource Id as the Subject Id.

sitory and Subject. One OSID Provider may use a Repository hierarchy to express an ontology that an OSID consumer is expecting to exist in a Subject hierarchy. Again, a third party may inject an OSID Adapter to remap a Repository to a Subject.

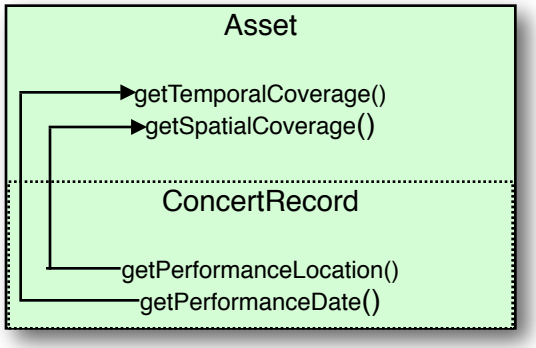
Asset Coverage

The core Asset specification defines methods general to digital content leaving more specific data to the records. Access to the records requires knowledge of a record Type specification. For searching Assets, keywords may be supplied by an OSID Consumer that may be applied to various data elements defined in an OsidRecord without knowledge of its Type specification.

So far so good. Some keyword queries may require more precision than a string can offer. Examples are time and space. An OSID Consumer may wish to perform a query to find Assets relevant within a certain time period, without understanding the specifics of

where or how that time period is defined. Asset defines temporal and spatial coverage to enhance generic searches.

An OSID Provider may allow an OSID consumer to apply this information to an Asset, or it may prefer to map other methods, perhaps those specified in an OsidRecord.



An Asset can float temporal and spatial coverage from other data in the Asset.

Intellectual Property

OSID principles regarding Intellectual Property:

- The OSIDs cannot be used in such a way as to prevent uses of content protected under *Fair Use* doctrine.
- The OSIDs must not interfere with a user's ability to understand the rights and restrictions existing upon content as conveyed by a provider.

Assets describe intellectual property information pertinent in handling digital content. The first set of methods convey whether or not known copyright protection exists, or the content is free to use without any strings attached. A search may include a query term to restrict results to find only those assets which are in the public domain to avoid any entanglements with copyright law.

- isPublicDomain(): tests if this is a public domain asset with no rights restrictions
- isCopyrightStatusKnown(): tests if there is any valid information available concerning copyright issues available in the asset.
- getCopyright(): a copyright statement

- getCopyrightRegistration(): a copyright registration number if the content has been registered with a copyright authority

- getPublishedDate(): the date of asset publication, if applicable. The published status affects copyright terms in many countries.

For assets protected under copyright, there may be usage restrictions on what can be done with the asset. This is not a DRM scheme. The purpose is to convey information to a user of the asset what the provider says they can or cannot do with it once they have it in hand. If the OSID Provider wishes to restrict access to an asset, then an authorization should be performed beforehand.

- getLicense(): gets a terms of use for an asset with a valid copyright

- canDistributeVerbatim(): conveys whether or not copies of this asset can be distributed as-is to any other party

- canDistributeAlterations(): conveys whether or not modifications to this asset can be distributed to any other party (e.g. derivatives)

- canDistributeCompositions(): conveys whether or not this asset may be included as part of another asset and distributed to any other party

The license may include particular terms or conditions placed upon the redistribution of the asset. For example, provider may restrict the posting of an asset to personal blogs. In this case, canDistributeVerbatim() would be false and the exception for bloggers would be documented in the license.

The purpose of these methods is to allow an OSID Provider to convey usage information to repository users in an interoperable manner. These methods are not designed to be the be-and-end-all of copyright management. Once an Asset has been retrieved, then the user has it on their computer. If the OSID Provider grants no authorization access to the Asset, these methods are unavailable. Therefore, the OSID can only convey what can be done with the content *after* it has been accessed and downloaded.

Asset Alternatives & Accessibility

Assets may relate to other assets in such a way as to provide alternative representations. These representations may be different renderings of the same data, such as a larger font or a different color map, or they may include different data altogether such as an audio track in lieu of a visual. Assets may also be created with different levels of quality or fidelity of the same media.

The core of the solution lies in the ability to relate assets to each other in the form of alternatives using the AssetAlternateSession. For any given Asset Id, an OSID Consumer may request a list of assets that have been designated as alternates.

In imaging, it is often desired to have different versions of the same image produced for different output devices with varying aspect ratios, color maps, resolution, sharpening, data formats, etc. To searching subject matter, it is desirable to return one asset for each criteria match instead of every version that may exist. The ability to focus the search is available in the AssetSearch interface.

15 Transaction Trouble

V3 carries forward the Transaction interface from V2. Unlike V2 where the transaction interface is applied to the manager, a transaction interface is available from OsidSessions. The V3 OsidManager is stateless.

The OSID interfaces are not transactionally-oriented as they are designed to be consumer-friendly. OSIDs don't define method sequences such as prepareToSearch(), executeSearch(), pickupSearchResults(), finishedWithSearch().

One case might be to update a series of OsidObjects. The session methods take one object form at a time resulting in an underlying operation for each update. The OSID Consumer desiring the most efficient means could do the following:

```
Transaction transaction;
if (session.supportsTransactions) {
        transaction = session.startTransaction();
}

for (Id id : idsToUpdate) {
    ObjectForm form = session.getFormForUpdate(id);
    form.setSomething(somethings[id]);
    session.updateObject(id, form);
}

if (session.supportsTransactions) {
        transaction.commit();
}
```

Code example of using Transaction to update a bunch of objects.

What exactly happens all depends on the OSID Provider. If the OSID Provider does not support transactions, then each update occurs separately. If the OSID Provider supports transactions, then the question becomes how it performs error handling.

The OSID Provider can validate the form and return an error on updateObject() and if supporting transactions it should probably validate as much as possible. However, some errors won't be detected until the OSID Provider performs the operation in the underlying system. In which case, the only place to inform the OSID Consumer of the problem is from commit().

The OSID Provider may or may not have the means of implementing ACID style transactions. In other words, if a failure occurs for the final update, the OSID Provider may have no means of rolling back to the state before the transaction began. This issue is left as a characteristic of the implementation.

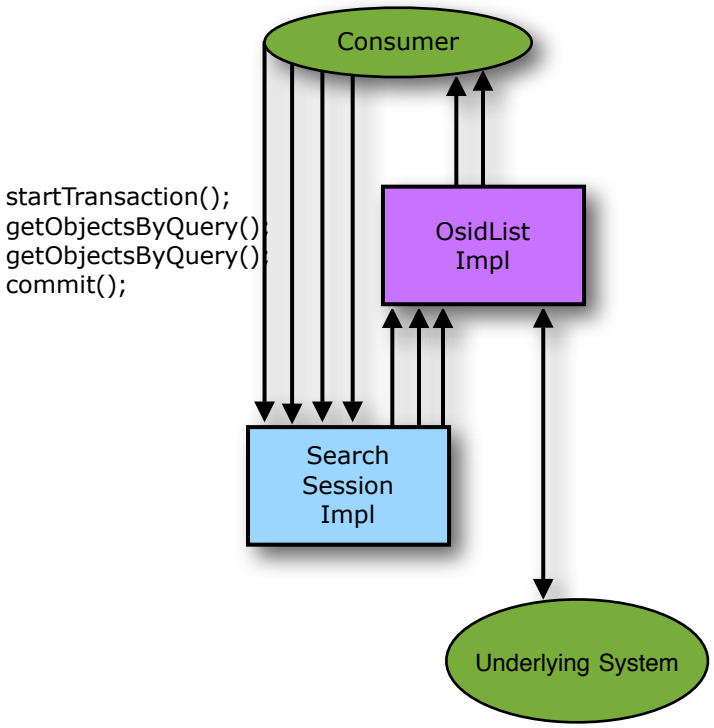
Why not have arrays instead of using transaction? One of the uses of Transaction within an OsidSession is to hide the awkwardness of handling arrays. The OSID Provider, inside a Transaction block, builds the array on behalf of the OSID Consumer and processes the array once commit() is invoked.

Transactions across methods that return values can be a bit trickier from the provider point of view. In the case of search, it might be desirable to invoke several queries within a single transaction to perform an OR with unique results.

```
Transaction transaction;
if (session.supportsTransactions) {
        transaction = session.startTransaction();
}

ObjectQuery query = session.getObjectQuery();
query.matchDisplayName("Fred", matchType, true);
ObjectList objects = search.getObjectsByQuery(query);
query.matchDescription("cartoon", stringMatch, true);
objects = search.getObjectsByQuery(query);

if (session.supportsTransactions) {
        transaction.commit();
}
```

Code example of using Transaction to combine search queries.

The wrinkle in using Transaction is that the method returns an OsidList. An implementation of such a scheme might do the following:



Attempting the illustrate the control flow in a transactional-based search.

The OSID Consumer invokes the four methods illustrated above. For the first getObjectsByQuery(), The Search Session implementation instantiates its OsidList implementation with the data from the first query and maintains a reference to it. When the OSID Consumer invokes the query the second time, it feeds that query data to the same OsidList object. When the OSID Consumer commits the transaction, the OsidList implementation communicates with the underlying system to execute both queries.

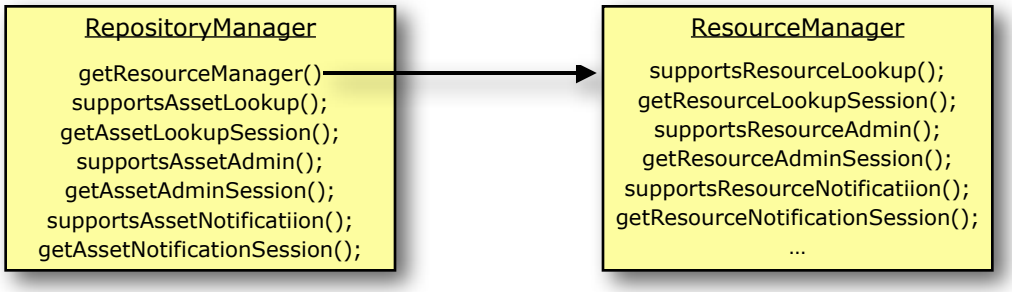
Until the first results are made available to the OsidList implementation from the underlying system, the methods such as hasNext() and getNextObject() block. As results are available to the OsidList implementation, the OSID Consumer can begin accessing them.

This illustrates that an OsidList is simply an interface for retrieving things, and its implementation may be significantly more complex than a simple collection. The heavy lifting that is pushed into the OsidList implementation serves to simplify the implementation of the sessions. Even if Transactions is not implemented, it may be a good idea to push queries and retrievals into the OsidList to enable the session methods to return immediately and allow OSID Consumers to retrieve objects as they are available without having to wait for the entire download to complete.

16 OSID Orchestration

When a V2 OSID referenced an object defined in another OSID, its `Id` was used. For method returns, the OSID Consumer required knowledge of the orchestration among OSID Providers to resolve the `Id` into an object. V3 performs this orchestration.

When a V3 OSID uses as part of its definition an external `OsidObject`, it provides access to the `OsidManager` under which the `OsidObject` can be queried and managed. The V3 Repository OSID, for example, leverages a Resource to capture identities of people related to an `Asset`. The `RepositoryManager` includes a method to access a `ResourceManager`.



An example of OSID orchestration.

Orchestrating the OSIDs in this way encapsulates the actual relationship between the Repository and Resource OSIDs. The OSID Consumer need only identify the identity of the Repository OSID implementation. The OSID Provider is free to the implement the Resource OSID in any way it sees fit.

One possibility is that there is no distinct resource service. The available resources are defined and managed within the repository implementation. In this case, the Repository OSID Provider maps the interfaces defined in the Resource OSID to its own internal data and implementation.

This orchestration is only defined when one OSID, as part of its specification, uses definitions from another OSID. An application that may wish to couple the Repository and Calendaring OSIDs may have to either have the orchestration performed in an OSID Adapter or at the OSID Consumer layer.

Not all cases of definition borrowing result in such an orchestration. Some OSIDs, like Hierarchy, only provide interface definitions.

17 Caveats

Casting

In V2, the Agent and filing OSIDs defined an interface hierarchy and required the OSID Consumer to cast one OSID object to another, This requirement has been eliminated in V3 and replaced with explicit retrievals for any object related to another. As mentioned earlier, it is common in Java to move up and down both the object and interface hierarchy as if they were equivalent whether using explicit casting, generics or polymorphism. The problem this exposes for the service layer is that it mixes up interface and implementation. Any form of cast performed on an OSID object is by definition an implementation of an out of band agreement between a particular OSID Consumer and OSID Provider. All is not bad here in that casting can be used in certain OSID Adapter implementations where it is necessary to change the behavior of an underlying implementation but such an OSID Adapter is not free to move independently from OSID Provider to OSID Provider.

Since we do not want to tie applications to OSID Providers, applications should not cast or assume any interface or object hierarchy in the OSIDs that are not explicitly indicated by a Type. Top-level interfaces are defined in the OSIDs as a means of ensuring a degree of consistency in the specification and simplifies reading the various OSIDs. An implementation may do what it wants. To examining the Filing case, an Entry contains methods in common with both a File and a Directory. An implementation of Filing may not retrieve all the information necessary to fulfill the contract for a File and a Directory. Casting an Entry to a File may result in an error. Conversely, polymorphing a File back into an Entry may result in an Entry object that the OSID Provider would never have generated. The OSIDs attempt to avoid this scenario by requiring retrievals through the associated sessions and never exposing Entry directly for the sake of saving a few method definitions.

To summarize, if an application programmer feels the need to cast (in any of its forms), then it should be considered a red flag. An exception to this rule might be found in the Java exception mechanism where it is acceptable to catch OsidException, to imply all OSID exceptions. On the other hand, the OSID doesn't specify what an exception is so this hierarchical relationship does not conflict with the OSIDs.

Nulls & Method Overloading

To eliminate some ambiguity, V3 specifies a no null rule. Nulls are not permitted as method arguments or returns. In programming, nulls are often used as a shortcut to avoid defining additional methods. V3 uses several patterns, albeit at the cost of some verbosity, to keep the specification clear and precise.

Method parameters are factored in such a way that there are no optional arguments. This is not performed through method overloading in that overloading is a not available in many languages. Creates and updates use a separate interface (the OsidForm) to allow the OSID Consumer to set exactly those parameters which are desired.

An interface may specify that a method may return zero, one or more objects. In the case of a many object returned, an OsidList is generally used to encapsulate the objects. An OsidList may be returned with no objects in lieu of a null. In the case of a single object specification, the method is generally required to return a NOT_FOUND error however in these cases there is an accompanying test method to avoid the error.

18 Loading OSIDs

In V2, the OsidLoader is an implementation of a factory pattern to instantiate OSID Providers. In V3, the OsidLoader is expressed strictly in the terms of an interface that may support more than one way of getting at an OSID Provider.

The OsidLoader has been renamed OsidRuntimeManager and is also responsible for finding appropriate implementation configurations and other non-OSID related resources that may have been installed using the Installation OSID.

Although difficult to ascertain from the interface definition itself, the tea ceremony inside an implementation of OsidRuntimeManager uses both the Configuration and Installation OSIDs. Here's how a Java version would work:

4. An OSID Consumer requests an OSID Provider by supplying the manager class name.

5. OsidRuntimeManager finds the class. It may find the class using a default classpath, a configured search path, or use the Installation service to see what OSIDs are installed.

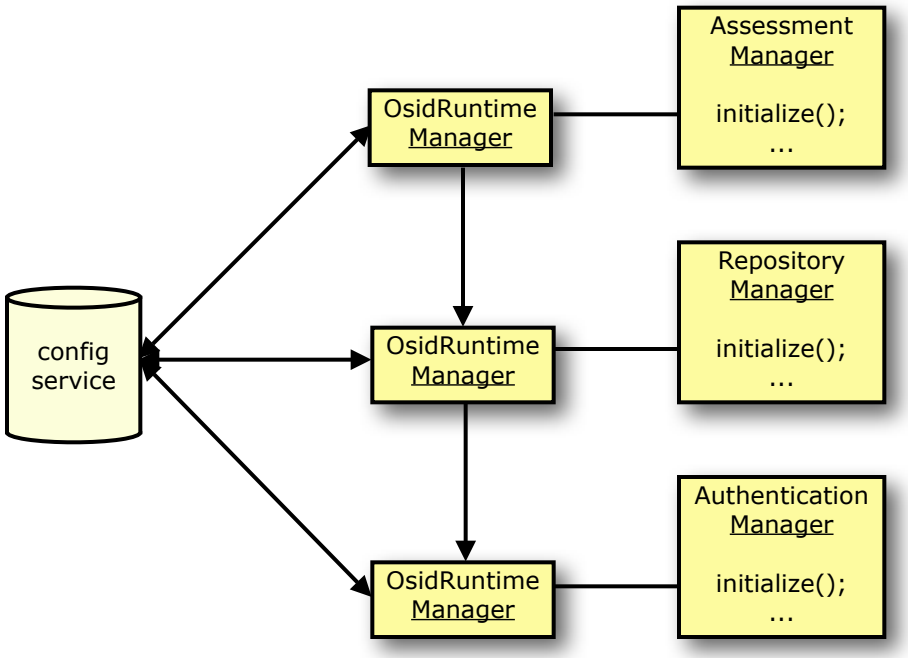
6. The class is instantiated with the interface of the requested service.

7. The OsidRuntimeManager makes copy of itself for the class being loaded.

8. The OsidManager's initialize() method is invoked. The argument to initialize() is the new OsidRuntimeManager.

9. The manager performs any startup initialization. If the manager loads any other OSIDs, it uses the runtime manager it was given. This instance of the runtime manager may have been given a different configuration, or a configuration handle to use when the initialization method asks for its configuration parameters.

Each new OsidManager gets its own copy of an OsidRuntimeManager so that configurations may be assigned to each OSID implementation which are centrally managed in the runtime environment through the Configuration OSID. Each new OSID instantiated is another link in the chain.



A confusing diagram. Separate OsidRuntimeManagers are created and passed to child OSID implementations. Each manager may have a separate configuration key, among other runtime data, used to retrieve a configuration.

The OSID Runtime may be implemented in such a way that each OsidRuntimeManager has awareness of the OsidManager that created it, forming a linked chain of runtimes each with knowledge of the OSID implementation instantiated. Therefore a configuration hierarchy may be created so that an OSID Provider can receive a different configuration based on what OSIDs are *on top* of it. While never exposing the OSID Consumer directly to the OSID Provider and maintaining configuration management in the runtime environment, the behavior of the OSID Provider may be modified based on a specific OSID Consumer, or a stack of OSID Consumers.

For example, *when edu.mit.osidimpl.installation is the OSID Consumer of com.harvestroad.repository.hive, the repository should only return zip installation files for getAssets() because the person who wrote the installation implementation believes all repositories revolve around him* can be corrected for in the repository implementation, or into an OSID Adapter in between.

While never exposing the OSID Consumer directly to the OSID Provider and maintaining configuration management in the runtime environment, the behavior of the OSID Provider may be modified based on a specific OSID Consumer, or a stack of OSID Consumers.

The instantiation of the OsidRuntimeManager is not defined in the OSID (its a utility in the developer kit) and this is where the chain begins.

This is what an OsidManager might do:

```
void initialize(OsidRuntimeManager runtime) {

   if (runtime.supportsConfiguration()) {
       ValueLookupSession session = runtime.getValueLookupSession();
       String[] serverNames = session.getValues(serverParameterId);
       String[] authZImpl   = session.getValues(authZImplParameterId);
   }

   AuthorizationManager manager = runtime.getManager(OSID.AUTHORIZATION, authZImpl[0]);
}
```

Code example of using the OSID Runtime to retrieve implementation configuration parameters and launch another OSID.

The OSID Provider implementation first uses the OSID Runtime environment for access to configuration parameters. This indirection allows for the change of configuration management of the OSID Provider. A developer's OSID Runtime may access configuration parameters from a local properties file while an enterprise may wish to centralize configuration management. The OSID Runtime can select a configuration keyed to the OSID Consumer providing for the ability to customize the OSID Provider based under what it is running. The details of the configuration interface is defined in the Configuration OSID.

The OSID Runtime is also used to instantiate other managers. The OSID Runtime manages issues of finding and loading OSIDs, which may or may not use the language environment's search path. This is somewhat analogous to Java's ClassLoader hierarchy, where breaking the hierarchy by not using the loader of one's environment, such as instantiating a new OsidRuntime directly (like using the System ClassLoader directly) can result not finding OSIDs of their supporting libraries properly.

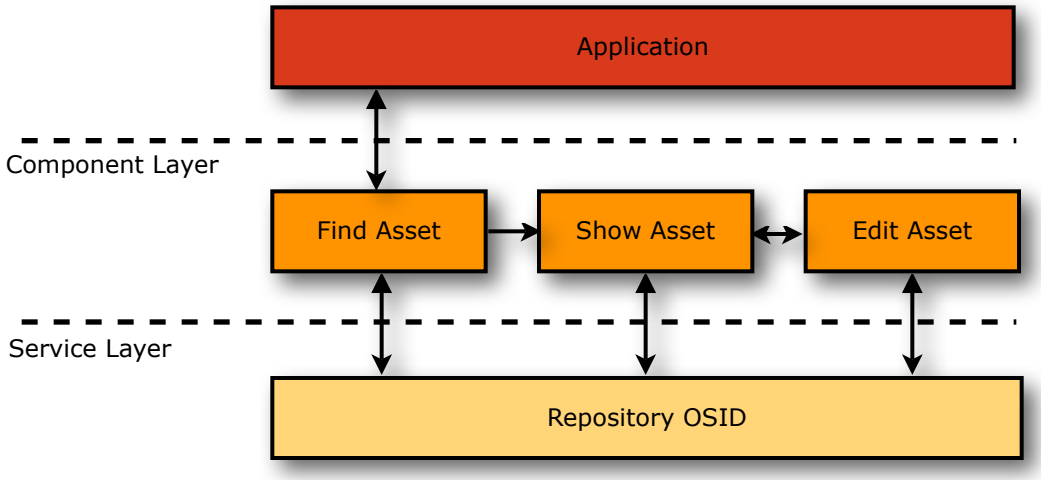
This may sound a bit like the V2+ Provider OSID. This interim OSID has been divided among the V3 OsidRuntime, the V3 Installation OSID, and the V3 Configuration OSID, all of which play a role in the V3 OSID Runtime environment.

19 Components

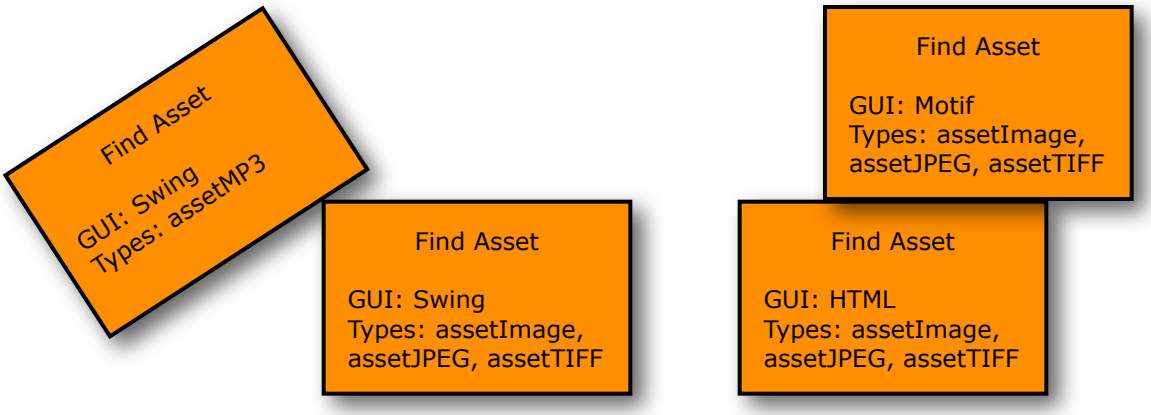
The OSIDs are positioned as an interface to a service layer. Sizable applications tend to be broken down into multiple layers. The service interface describes how a OSID Consumer interacts with a service which is generally not how an application programmer wants to manage GUI components.

The service layer is not complete without data descriptions but the OSIDs stay clear of locking into particular data specification by leaving this open to interface extensions. This requires type agreements between the OSID Consumer and OSID Provider. Adding new type support within an OSID Provider can be accomplished with the use of federating OSID Adapters. A similar modular framework should be available to the application programmer.

There has been prior work in defining a modular component layer that would ride above the service layer and encapsulate the GUI management that pertains to the OSIDs. There may be a component for a particular application platform and look&feel that performed an asset search. It would encapsulate the search mechanism and any related types that together with a Repository OSID provides a complete solution for an application. Such a module can be replaced for a different UI, or adapted as would be done for the OSIDs for the purposes of federation or adding type support.



The component layer buffers the application from service details.



Example components toolbox.

The OSIDs are designed to address a particular problem. The places in the OSIDs that are open for flexibility should be addressed by the component layer to ease the risk of bloating applications.While to date there is no component layer offered from O.K.I., it is necessary to fully grasp the OSIDs and understand what goes where.

20 New Services

Cataloging

The Cataloging OSID defines a simple interface for associating an OSID `Id` with a catalog. Many OSIDs define sessions and objects for the purpose of cataloging. A `Repository` is an example. The Cataloging OSID exists for those OSID Consumers (or other OSID Providers) to factor out cataloging operations into a separate service.

Configuration

The Configuration OSID provides a means for an OSID Consumer to manage configurations and profiles. The OSID Consumer may be an end-user application needing to retrieve configurations for itself or on a per-user basis.

The OSID Consumer may also be another OSID Provider.

Locale

The Locale OSID is a simplified variant of Dictionary that is more tuned to translating strings and other localized items across various locales.

Installation

The Provider OSID was developed post-V2 as part of the VUE project to provide a means for the searching, installation and access to OSID Provider implementations. Much of this was developed under the constraint of the rest of the V2 specification.

In V3, Provider is renamed Installation. Installation is reduced in scope to handle searching and retrievals of OSID implementations from remote package repositories and local installations. The access to installed OSID Providers has been merged with the OsidLoader.

Journaling

Journaling provides a means of tracking changes to an OSID object. Journaling captures any metadata associated with an object creation or change and allows objects to be versioned. The Journaling OSID is highly abstract, and intended for use as an assist to a

OSID Consumer that wishes to factor out journaling operations. Some OSIDs offer a journaling session.

Metering

To be designed. A service to track and manage usage and allocation.

Provisioning

Provisioning defines a service that performs allocations or assignments among Resources.

Relationship

To be built. An OSID to capture data about relationships among OsidObjects.

Resource

The Resource OSID defines a service that manages objects referenced throughout the OSIDs. In V2, this was part of the Agent OSID. The Resource completes the grammar of the OSID language by describing the direct or indirect object of the sentence. In provisioning, for example, *a librarian may provision a book*. The book is represented by a Resource and the librarian is represented by an Agent. The agent is the authenticated entity that performed the action. The Agent, in turn, may map to another Resource that describes the person who has the *role* of librarian as defined by the Authorization OSID.

Spatial

To be designed. An OSID to manage spatial relationships and provide a Spatial interface for use with asset spatial coverage.

Topology

The Hierarchy OSID has been simplified to define only the service required to present a hierarchy of objects defined outside the scope of the hierarchy service. In other words, the Node has been eliminated and replaced with a single Id. This works better for OSID Consumers who are using Hierarchy as an assist to structure objects known in their domain. The V3 Hierarchy is not cyclic.

Topology is a more general form of the V2 Hierarchy OSID that defines both a Node and an Edge. It can be used to define arbitrary relationships as well as capture the nature of relationships. They can be hierarchies, rings or general topographical maps.

Transport

Transport defines a means to move data. This service is positioned as a utility to OSID Consumers who wish to abstract implementation specific protocol APIs and provide a means for containing implementation issues such as server location and fault tolerance.

Type

The Type OSID manages Type definitions across all OSIDs. Its primary purpose is to provide a standard means for removing hard-coded Type definitions and to provide a mechanism for the sharing and dissemination of Type definitions.

Additional OSIDs in consideration include Simulation, Payment, and Budgeting.

21 Interface Navigator