



Blackboard

Blackboard Academic Suite™

Blackboard Building Blocks™ Introduction to the Building Blocks APIs and Runtime

Release 7
Blackboard Learning System™
Blackboard Community System™
Blackboard Content System™

Date Published: October 13, 2005

Copyright © 2005 by Blackboard Inc. All rights reserved.

Blackboard, the Blackboard logo, Blackboard Academic Suite, Blackboard Learning System, Blackboard Learning System ML, Blackboard Community System, Blackboard Transaction System, Blackboard Building Blocks, and Bringing Education Online are either registered trademarks or trademarks of Blackboard Inc. in the United States and/or other countries. Intel and Pentium are registered trademarks of Intel Corporation. Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries. Sun, Solaris, UltraSPARC, and Java are either registered trademarks or trademarks of Sun Microsystems, Inc. in the United States and/or other countries. Oracle is a registered trademark of Oracle Corporation in the United States and/or other countries. Red Hat is a registered trademark of Red Hat, Inc. in the United States and/or other countries. Linux is a registered trademark of Linus Torvalds in the United States and/or other countries. Apache is a trademark of The Apache Software Foundation in the United States and/or other countries. Macromedia, Authorware and Shockwave are either registered trademarks or trademarks of Macromedia, Inc. in the United States and/or other countries. Real Player and Real Audio Movie are trademarks of RealNetworks in the United States and/or other countries. Adobe and Acrobat Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Macintosh and QuickTime are registered trademarks of Apple Computer, Inc. in the United States and/or other countries. WordPerfect is a registered trademark of Corel Corporation in the United States and/or other countries. Crystal Reports is a trademark of Crystal Decisions in the United States and/or other countries. WebEQ is a trademark of Design Science, Inc. in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners. Patents pending.

© 2005 Blackboard Inc. All rights reserved. Made and printed in the USA.

No part of the contents of this manual may be reproduced or transmitted in any form or by any means without the written permission of the publisher, Blackboard Inc.

Worldwide Headquarters

Blackboard Inc.
1899 L Street, NW, 5th Floor
Washington, DC 20036-3861 USA
800-424-9299 toll free US & Canada
+1-202-463-4860 telephone
+1-202-463-4863 facsimile
www.blackboard.com

International Headquarters

Blackboard International B.V.
Keizersgracht 106
1015 CS Amsterdam
The Netherlands
+31 20 5206884 (NL) telephone
+31 20 5206885 (NL) facsimile
global.blackboard.com

Table of Contents

Introduction	4
Data Objects	5
Data Object Packages	6
Supported Data Objects	7
Using Data Objects	9
Strongly-Typed Enumerations	12
System Services	14
Service Manager	15
Services	16
Persistence	17
Persistence packages	18
Containers	19
Persistence Manager	20
Loaders and Persisters	21
IDs and Persistence	23
Using Data Objects and Persistence	26
Exception Handling	28
Runtime Environment	29
Blackboard Content System APIs	30
Blackboard Content System Data Object Packages	31
Blackboard Content System Supported Data Objects	32
Using Blackboard Content System Data Objects	34
Using Blackboard Content System File System Objects	35

Introduction

Overview

The Building Blocks APIs and Runtime allows developers to programmatically interact with the *Blackboard Academic Suite*. It is not a specific Application Program Interface (API), rather it is a set of patterns and objects that relate to one another as a common mechanism used to store and retrieve the data manipulated in the public APIs (such as Content, Announcement, Calendar). While the majority of a developer's time will be spent using the public APIs, knowledge of the Building Blocks APIs and Runtime and the patterns used within it is invaluable because all of the public APIs use the Building Blocks APIs and Runtime services and work within the patterns established by it.

The principle use of the Building Blocks APIs and Runtime is to create, modify, or delete data within the *Blackboard Learning System* or to create custom content for the *Blackboard Community System* and *Blackboard Content System*. It has been designed to support those operations in a flexible manner, which may be from one *Blackboard Learning System* to another or the export of data into an IMS compatible package.

Audience

This document is intended for developers interested in building solutions for the *Blackboard Academic Suite*. Familiarity with object-oriented programming in Java[®] and Java Server Pages (JSP) development is assumed. Additionally, the reader should be familiar with the features of the *Blackboard Academic Suite*.

More information on Java Server Pages may be found at:

- Java Servlet Specification, version 2.2
<http://www.javasoft.com/products/servlet/2.2/>
 - Java Server Pages (JSP) Specification, version 1.1.
<http://www.javasoft.com/projects/jsp/1.1/>
-

Using This Document

In many places the document will reference the Building Blocks API JavaDoc that serves as the authoritative reference. The JavaDoc is published as the *Building Blocks API Specifications*. The example below demonstrates how these will be denoted:

See package `blackboard.data.content`.

Code samples will appear in the following typeface: `code sample`.

Manual Updates

Please note that *Introduction to Building Blocks APIs and Runtime* is updated periodically. Check the Date Last Update at the beginning of the manual to ensure that it is the most recent copy. Any updates are listed in the [Appendix](#).

To report any comments or suggestions regarding this manual, please contact Blackboard Support.

Data Objects

Overview

Data objects are simple encapsulations of data entities found within *the Blackboard Learning System*. Data objects within the Building Blocks APIs and Runtime directly map to the entities a user would see represented in the *Blackboard Learning System* user interface. These objects contain no business logic and act primarily as attribute repositories. Access to these attributes is provided through “get” and “set” methods similar to those employed in the Java Beans™ model.

The data objects are independent of any storage or persistence mechanism—in other words, the data objects are completely separate from the containers used to store data.

All data objects are in the system sub-class `blackboard.data.BbObject`. This sub-class contains most of the functionality used by the map-based persistence framework.

In this section

This section includes the following topics:

- [Data Object Packages](#)
 - [Supported Data Objects](#)
 - [Using Data Objects](#)
 - [Strongly-Typed Enumerations](#)
-

Data Object Packages

Introduction

The data object definitions reside in the various sub-packages of `blackboard.data`. The sub-packages are grouped into various functional areas like Content or Calendar.

Data Object sub-packages

The data sub-packages are described in the table below.

Data object sub-packages	Classes contained
announcements	Contains objects associated with Announcements in the system.
calendar	Contains objects used by the Calendar subsystem.
content	Contains classes for building content within a course. Content includes Course Documents (more commonly referred to as Content Items), External Links, and Staff Information.
course	Contains classes for core learning system concepts such as courses, groups, and course membership.
gradebook	Classes for interacting with the course Gradebook.
user	Contains core classes pertaining to users.

Besides the above-mentioned sub-packages, the `blackboard.data` package contains a number of important classes, such as `BbAttributes` and `BbLink`. These classes include several interfaces and exceptions as well as the base object for all data objects: `BbObject`.

Supported Data Objects

Overview

The Building Blocks APIs and Runtime provides information about all of the data that is tracked by the *Blackboard Learning System*.

Data object descriptions

The following table describes a subset of the supported data objects and a brief description about the *Blackboard Learning System* entity that they are designed to represent.

Entity	Description
Announcement	<p>Messages posted by the Instructor of a course Web site to Students or by the System Administrator to users. Announcements may appear in the My Institution portal or on a course Web site's home page (The default is the Announcement page).</p> <p>See package <code>blackboard.data.announcement</code></p>
CalendarEntry	<p>A detailed item in a Calendar. There are several "Calendars" in the system, including the system Calendar, user Calendars, and course Calendars.</p> <p>See package <code>blackboard.data.calendar</code></p>
CourseDocument	<p>The most basic content item accessible in the Content Areas of a course. Content areas are defined by the CourseTOC object, which is new in <i>Blackboard Learning System</i> (Release 6).</p> <p>See package <code>blackboard.data.content</code></p>
ContentFolder	<p>A container for Content objects. Any object that subclasses Content may be placed in a ContentFolder, with some exceptions, such as ContentFile. The rules for determining what object can be placed in a folder are more completely documented in the JavaDoc.</p>
ExternalLink	<p>Hyperlinks to Web sites outside of <i>Blackboard Learning System</i>. In <i>Blackboard Learning System</i> (Release 6) External Links may appear in any Content Area of a course.</p> <p>See package <code>blackboard.data.content</code></p>
StaffInfo	<p>Course area dedicated to displaying profiles of instructors, teaching assistants, and course leaders. They are typically created by and for Instructors and assistants in a course.</p> <p>See package <code>blackboard.data.content</code></p>
StaffInfoFolder	<p>As with Course Documents, Staff Info objects may be organized into collections called folders. These folders may only appear in the Staff Information content area of a course. StaffInfoFolder and ContentFolder are mutually exclusive. Objects of a type that may be placed in one may not be placed in the other.</p> <p>See package <code>blackboard.data.content</code></p>

Entity	Description
Course	<p>A set of learning materials created by an Instructor. The Course is the principal administrative object within the system. All content is associated with a course. Several aspects of access control are dependent upon the course site currently being accessed.</p> <p>See package <code>blackboard.data.course</code></p>
CourseMembership	<p>A representation of a user's role within the context of a course. A user's role (for example, Instructor or Student) affects their capabilities within a course and can change from one course to another.</p> <p>See package <code>blackboard.data.course</code></p>

Using Data Objects

Introduction

All data objects have public, no-argument constructors and can be created like any other Java class.

Creating a data object

The following code demonstrates how to create an `Announcement` object.

```
Announcement ann = new Announcement();
```

Creation of a data object in this way will result in an object with default values for all of the object's attributes. To create a more useful `Announcement`, it is necessary to create one that contains more than default values. To do this, create a default `Announcement` object and call all of the appropriate "set" methods provided by the object.

Example

Since data objects map directly to entities within the *Blackboard Learning System*, a good example of how to construct an `Announcement` is to examine the announcement creation user interface within the *Blackboard Learning System*. Next map user interface elements to the appropriate attributes within the `Announcement` object and examine how to construct the `Announcement` from this mapping. The following image represents the Add Announcement page.

MY COURSES > BLUE > CONTROL PANEL > ANNOUNCEMENTS > ADD ANNOUNCEMENT

Add Announcement

1 Announcement Information

Subject:

Message:

Smart Text Plain Text HTML

2 Options

Always show this announcement on the course's main page. Yes No

Restrict dates to show this announcement:

Apr 23 2002

04 20 PM

Display Until

Apr 24 2002

04 20 PM

Given the user interface and attribute-to-method mappings an `Announcement` object can be constructed as shown below. Assume the above Web page is being processed within a JSP page and that reasonable names for the form elements exist.

The following code represents creating an announcement data object from HTML form values:

```
Announcement ann = new Announcement();

// begin retrieving form element values in order to fill out the announcement
ann.setTitle( request.getParameter( "subject" ) );

// announcement body is a FormattedText block (element made up of text and
// formatting). Retrieve message and message type values to construct it.
ann.setBody( new FormattedText( request.getParameter( "message" ),
    FormattedText.Type.fromExternalString( request.getParameter( "msg_type" ) ) ) );

// if display after checkbox is not selected, form value will be null
if ( request.getParameter( "display_after" ) != null )
    ann.setRestrictionStartDate( calDisplayAfter ); // assume calendar processing
else
    ann.setRestrictionStartDate( null ); // no "display after" specified

// if display until checkbox is not selected, form value will be null
if ( request.getParameter( "display_until" ) != null )
    ann.setRestrictionEndDate( calDisplayUntil ); // assume calendar processing
else
    ann.setRestrictionEndDate( null ); // no "display until" specified
```

Given the above code, an `Announcement` object can be created that matches the announcement object created by a user within the interface.

Data object IDs

A review of the `Announcement` JavaDoc shows that the above code does not actually call all of the available "set" methods in `Announcement`. More specifically, the code does not exercise any of the "ID" set methods available.

All data objects within the Application Framework are identified by a "key" value called an ID (this property is inherited through the data object base class `BbObject`). An object's ID is designed to uniquely identify that object from others within the Building Blocks APIs and Runtime. IDs are also used to represent relationships between data objects in a fashion similar to foreign key relationships in a relational data model.

An ID includes more information than just the database primary key—it also includes object type and container information. This allows IDs to be used for equality checks. For example, since both an `Announcement` and `Calendar Entry` can have '10' as the value of their primary key, the additional type check ensures they are properly differentiated. Additionally, data from different Virtual Installations may be loaded into the same Java Virtual Machine, so it's possible to have two `Announcements` loaded with the same primary key, therefore container information is used to differentiate the objects. For more information see [Virtual Installations](#).

Formally, an Object ID (OID) in the *Blackboard Learning System* is a collection of three pieces of information: `Pk`, `Type`, and `Container`, where `Pk` is the primary key of the object, `Type` is the associated logical type (not necessarily the Java language type), and `Container` is the physical data store for the object.

Further information on IDs and the important role they play within the Application Framework, including how they are created and how they are used, can be found in the [IDs and Persistence](#) topic.

Strongly-Typed Enumerations

Introduction

Although Java does not explicitly include support for enumerated types the framework includes a pattern to emulate the type and bounds checking that is typically associated with enumerated types. All enumerations used in the data objects, and elsewhere in the Building Blocks APIs and Runtime, are implemented using a pattern that provides for compile-time type checking for enumerated values.

Enumeration Classes

In this pattern, there is a primary enumeration class with the following properties:

- private constructors
- static final instances of the enumeration class that are defined for each element in the enumeration

For convenience, the enumeration classes are frequently declared as inner classes of a more general class that provides for scope and context. For example, the class `Course` contains an inner class called `Pace`. The values of the `Pace` enumeration class are used when calling the `getPaceType()` or `setPaceType()` methods on `Course`. The code sample below shows how this is done:

```
// create a course and then set the pace value
Course course = new Course();
course.setPaceType( Course.Pace.INSTRUCTOR_LED );

// now retrieve the pace value from the course object
Course.Pace pace = course.getPaceType();
```

Because the elements of the enumerations are declared as `static final`, comparisons of enumeration values can be performed using the `==` operator. The example below demonstrates switching on strongly typed enumeration values.

```
Course.Pace pace = course.getPaceType();
if ( pace == Course.Pace.INSTRUCTOR_LED )
{
    /* handle case one */
}
else if ( pace == Course.Pace.SELF_PACED )
{
    /* handle case two */
}
else
{
    throw new InternalError(
        "Unknown enumeration element encountered: " + pace.toString() );
}
```

It is good form to explicitly handle each case in the enumeration—the code should not “fall through” with an `else` and assume a certain state, unless not all cases in the enumeration require explicit handling. This reduces code readability and introduces the possibility for error if an enumeration is widened later.

Base Class

All strongly typed enumerations used with the Building Blocks APIs and Runtime data objects inherit from a single base class `BbEnum`. This base class provides a number of useful features including the ability to convert between an enumeration value and a `String` value. The below code sample demonstrates this capability, as well as other features provided by `BbEnum`:

```
// convert the course pace value to a string
String strPace = course.getPaceType().toExternalString();

// convert a string to a course pace value
Course.Pace pace = Course.Pace.fromExternalString(
    course.getPaceType().toExternalString() );

// retrieve the default course pace enumeration value
Course.Pace defPace = (Course.Pace) BbEnum.getDefaultElement( Course.Pace.class );

// determine if an enumeration value is the default enumeration value
boolean isDefault = defPace.isDefault(); // will return true

// retrieve the list of pace values and go through the list
Course.Pace[] paces = Course.Pace.getValues();
for ( int i = 0; i < paces.length; i++ )
{
    Course.Pace aNewPace = paces[i];
    // now do something with the pace value...
}
```

For additional information see `blackboard.base.BbEnum`.

System Services

Introduction

The Building Blocks APIs and Runtime provides access to several useful services through the Services Framework. These services provide many useful features, including user authorization and access to system configuration information. All access to supported services is provided through the service manager. This section includes a description of the service manager as well as some useful services it provides.

In this section

This section contains the following topics:

- [Service Manager](#)
 - [Services](#)
-

Service Manager

Introduction

Access to all system services is provided through the service manager, represented within the framework by the object `blackboard.platform.BbServiceManager`. The methods of the service manager used to access individual services are declared `public static`, and can therefore be immediately accessed at all times by a developer.

Function

The service manager provides easy access to several commonly requested services through convenience methods. The persistence service, shown in earlier code samples, is one of these commonly accessed services that can be retrieved. The example below demonstrates how to retrieve the persistence service from the service manager:

```
PersistenceService service = BbServiceManager.getPersistenceService();
```

Retrieving a service for which `BbServiceManager` does not provide a convenience method is done using the `lookupService()` method. Below is an example of service retrieval using this technique:

```
FileSystemService service = (FileSystemService)  
BbServiceManager.lookupService( FileSystemService.class );
```

Besides providing easy access to services, the services manager improves application performance by effectively caching service instances whenever possible.

Services

Introduction

Some of the useful services provided by the Service Manager include:

- Persistence Service
 - Configuration Service
 - Session Manager Service
 - Access Manager Service
-

Persistence Service

As mentioned earlier, the persistence service can be used to retrieve instances of the persistence manager configured to operate against containers with default configurations. Most applications retrieve a persistence manager instance through this service.

See `blackboard.platform.persistence.PersistenceService`

Configuration Service

The configuration service provides access to a number of system settings including the file system directory values for various key application locations.

See `blackboard.platform.config.ConfigurationService`

Session Manager Service

The session manager service provides access to a user's `BbSession` value. This object is useful when developing within a Web-based environment, for example, when developing a JSP page as part of an extension to the *Blackboard Learning System*. `BbSession` provides access to important information about the current request and the user making that request.

See `blackboard.platform.session.SessionManagerService`

Access Manager Service

The access manager service provides important information concerning user authorization. It provides access to the information necessary for determining whether or not a particular user has the ability to access a system resource.

See `blackboard.platform.security.AccessManager`.

Persistence

Overview

Building data objects and iterating through lists of them is interesting, but not very useful. It is more useful to create data objects and push them to a persistent data store for retrieval at a later point. It is also helpful to store a data object so that it will appear within the *Blackboard Learning System*. The Building Blocks APIs and Runtime provides this functionality through a process called persistence.

Generically, persistence refers to the pushing and pulling of data objects to and from a persistent data store. The act of pushing data objects is called persisting, while the act of pulling data objects from a store is called loading. The Building Blocks APIs and Runtime defines a persistent data store as a container.

Note: The persistence subsystem is initialized through a set of XML-based configuration files. Information on this is provided for information only. Editing this file is not permitted by the license agreement.

In this section

This section includes the following topics:

- [Persistence Packages](#)
 - [Containers](#)
 - [Persistence Manager](#)
 - [Loaders and Persisters](#)
 - [IDs and Persistence](#)
 - [Using Data Objects and Persistence](#)
 - [Exception Handling](#)
-

Persistence packages

Introduction

The base persistence objects reside in the package `blackboard.persist`. This package contains a number of sub-packages, grouped into functional areas that are the same as those for data objects. These sub-packages provide the means for loading/persisting the class of data objects that their names imply.

Sub-packages

The persistence sub-packages are described in the table below:

Persistence sub-packages	Interfaces contained
announcements	Contains interfaces for loading and persisting Announcements.
calendar	Contains interfaces for loading and persisting Calendar Entries.
content	Contains interfaces for loading and persisting content within a course. Content includes Course Documents (also known as Content Items), External Links, and Staff Information.
course	Contains interfaces for loading courses. This contains classes for enrollment.
gradebook	Contains interfaces for loading and persisting Gradebook items and scores.
user	Contains interfaces for loading users.

Containers

Introduction

A container, represented within the Building Blocks APIs and Runtime by the interface `blackboard.persist.Container`, abstracts the notion of a persistent data store. Therefore a `Container` represents a place to which data objects can be pushed (persisted) and retrieved (loaded) from at a later point in time. While one can imagine a great number of different kinds of containers, from file systems to ftp sites, the Building Blocks APIs and Runtime currently supports one type of container, a relational database. A relational database container is concretely represented by the class `blackboard.persist.DatabaseContainer`.

Because different containers offer diverse capabilities, including different mechanisms for identifying persistent objects, indexing objects, and tracking relationships between objects, interacting with different containers may require more than one technique. However, regardless of the capabilities offered by any one container, the processes responsible for interacting with it will always accept and return data object values just like the processes for all other containers.

In almost all cases, developers will not interact with Containers directly. Instead, the framework will manage the correct associations independently.

Persistence Manager

Introduction

The various sub-packages of `blackboard.persist` contain only interfaces describing how one would persist and load a data object to a container, but they do not contain actual implementations of these interfaces that can be used by a developer. The Building Blocks APIs and Runtime was designed to be flexible, allowing for more than one loader and persister implementation for each data object, for each container. Due to this flexibility, loader and persister objects are obtained through a factory. The persistence manager, `blackboard.persist.BbPersistenceManager`, serves the role of factory, and is responsible for providing callers with loader and persister implementation instances appropriate to a particular situation.

Container association

The loader and persister implementation classes returned by the persistence manager are situation specific. The situation, and thus the decision of which implementation instance to return, is governed by a number of factors, the most important being the container against which the persistence manager is operating. Persistence managers are directly associated with a particular container when they are created. Once created in this manner, the persistence manager will only provide loader and persister implementations specifically designed for use with that particular container.

The following code sample demonstrates how one would obtain a persistence manager instance associated with a database container.

```
BbPersistenceManager bbPm =  
    BbPersistenceManager.getInstance(  
        new DatabaseContainer( BbDatabase.getDefaultInstance() ) );
```

Persistence Service

The example above does not provide for efficient caching and reuse of the persistence manager, therefore, a better way to retrieve a persistence manager instance attached to the default database container is through the persistence service. The following code sample demonstrates how to obtain the default database persistence manager.

```
BbPersistenceManager bbPm =  
    BbServiceManager.getPersistenceService().getDbPersistenceManager();
```

The persistence service is always available for retrieval through the `BbServiceManager` and can be used to retrieve default instances of the persistence manager. Default instances are persistence manager instances configured to operate against containers with default configurations. The persistence service is always resident in memory, therefore, it can perform efficient caching, allowing it to provide the caller with the same persistence manager instance whenever called. This also allows the persistence manager to perform whatever caching is necessary, leading to increased performance.

For more information on services, see the topic [System Services](#).

Loaders and Persisters

Introduction

Loaders are the objects responsible for reading data from a container and converting the data into a live data object reference. They offer read-only operations on the data model, such as looking up an object by a key value, or searching for a list of objects based on given criteria. Persisters are the objects responsible for taking a live data object reference and writing the data into a container. They offer read-write functions such as insert, update, and delete.

Retrieving loaders and persisters

Loader and persister instances are obtained through the persistence manager. For this purpose, `BbPersistenceManager` exposes two key methods, `getLoader()` and `getPersister()`. Loaders are retrieved with a call to `getLoader()` while persisters are retrieved through a call to `getPersister()`.

Loaders and persisters are typed so that there is a specific (or possibly more than one) instance that targets a particular container and data object.

The following example demonstrates how to retrieve a loader that can load a `Course` object from the database described by the default database:

```
BbPersistenceManager bbPm =
    BbServiceManager.getPersistenceService().getDbPersistenceManager();
CourseDbLoader loader = (CourseDbLoader) bbPm.getLoader( CourseDbLoader.TYPE );
```

To assure that a loader instance capable of loading from a database container is retrieved, a persistence manager instance attached to the database container must first be retrieved. Once this is done, a `Course` loader is retrieved through the persistence manager by specifying the course database loader type (`CourseDbLoader.TYPE`) in the `getLoader()` call.

Once the `Course` loader instance is obtained, all of the methods described in the `CourseDbLoader` interface can be used to load a course(s) from the database. Because all loader and persister instances are loaded in the exact same fashion, one can see in the example above how to load any other type of object. For example, the code sample below shows how to obtain an `Announcement` database persister:

```
BbPersistenceManager bbPm =
    BbServiceManager.getPersistenceService().getDbPersistenceManager();
AnnouncementDbPersister persister =
    (AnnouncementDbPersister) bbPm.getPersister( AnnouncementDbPersister.TYPE );
```

All loader and persister interfaces contain a public static value called `TYPE` that is used as the key to retrieve a loader and persister instance of that type. It is this value, as shown above, that is provided to the `getLoader()/getPersister()` method that allows the persistence manager to retrieve a loader and persister instance of the appropriate type.

“Heavyweight” vs. “Lightweight”

By default, persistence operations (loading or persisting) deal with complete object graphs. For example, a `Lineitem` object, when loaded, will load all of its contained

Score objects. In many cases a loader may provide options to only retrieve the actual object, and not any dependent/contained objects.

Prior to *Blackboard Learning System* (Release 6), different versions of the data objects were provided to map the different sets of attributes that may be associated with light or heavy objects. In *Blackboard Learning System* (Release 6), those objects have been removed, since the state of all data objects is determined using `blackboard.data.BbAttribute` classes. This allows for smart evaluation of whether a particular attribute was loaded or not.

IDs and Persistence

Introduction

As discussed in the topic [Data Object IDs](#), all data objects within the Building Blocks APIs and Runtime are identified by a “key” value called an ID. An object’s ID is designed to uniquely identify that object from others within the framework. While ID values are part of a data object’s attributes, they play a more important role in persistence and the processes of persisting and loading objects from a container.

ID value

In order to be unique within a system an ID value, represented abstractly within the framework by `blackboard.persist.Id`, is a composite of several values including a container reference, a type specification, and a key value. The container reference describes the container from which the object identified by the ID was loaded (or should be loaded from in the case of an ID that describes a relationship). The type specification is a value of type `blackboard.persist.DataType` and describes the “type” of object the ID refers to, such as `Course`, `Announcement`, or `CourseDocument`. All data objects contain a public static field of type `DataType` that can be used when creating IDs. The key value represents an identifying value whose only requirement is that it is at least unique for a single data type within a given container. An example of an acceptable key value is the primary key value of a database table.

Concrete ID types

Concrete instances of `Id` are container specific because different containers implement the notion of an ID field in different ways. However, there is no need to know the concrete `Id` type being used with any object because all interactions with an `Id` derived value should be handled through methods provided by an `Id`. A developer should never cast an ID value provided by the Application Framework to a concrete type in order to manipulate it.

Generating ID values

The Application Framework will generate ID values for the developer. An example of this is through a load or persist operation. However, it is possible for a developer to generate his or her own ID values without knowing the concrete `Id` class for a `Container`. This is done via the `Container.generateId()` set of methods. There are different versions of the `generateId()` method but they only differ in the form with which the key value is provided: as one `String` or as two integers.

Assuming there is a string representation of the ID key value of a `Course` object (in a variable called `strCourseId`) the `Id` can be reconstructed. The example below of building an ID from a string value using a container demonstrates this:

```
BbPersistenceManager bbPm =
    BbServiceManager.getPersistenceService().getDbPersistenceManager();
Container container = bbPm.getContainer();
Id id = container.generateId( Course.COURSE_DATA_TYPE, strCourseId );
```

`BbPersistenceManager` also provides a convenient method for generating `Id` values since developers do not typically store references to the current `Container` they are operating against. Using these methods, which follow the same form as those exposed

by `Container`, the above code sample can be reduced. The example below of building an ID from a string value using the Persistence Manager demonstrates this.

```
BbPersistenceManager bbPm =
  BbServiceManager.getPersistenceService().getDbPersistenceManager();
Id id = bbPm.generateId( Course.COURSE_DATA_TYPE, strCourseId );
```

To use either of the above methods to generate an `Id` value, an ID key value in `String` form is required. To generate this, use the `toExternalString()` method on an existing ID value. Do not use the `toString()` method on `Id` since this method generates debug information, and its output cannot be understood by the `generateId()` methods.

How persisters use IDs

The ID of a data object plays an important role in the process of persistence and can be used by a persister to make decisions concerning how a particular data object should be handled for storage. For example, when an object is persisted to a database container, the ID value of the object and its validity (whether or not it is valid for that specific container) is used to determine whether an insert operation, or an update operation should be performed. If the ID of an object is not valid, an insert will be performed. If the ID is valid, an update will be performed.

Containers

The ID value of an object changes to reflect the state of that object in the last container the object operated against. In other words, the ID value of a data object, as returned by `getId()`, changes as the object is persisted to different containers. When a data object is created it is automatically assigned a default ID value. This default ID is not associated with a container and is therefore not valid for any container. When this data object is then persisted to a container that container is responsible for generating a new `Id` value and assigning this new `Id` value to the object after the persist operation succeeds.

Example

The following code shows the life cycle of a data object ID value through a series of operations.

```
// assume we have created two persistence manager instances (not using the default
// values returned through the persistence service) each of which is associated
// with a different database container - going against two databases at one time
//
// bbPm1 is associated with database container 1 (database 1)
// bbPm2 is associated with database container 2 (database 2)

Id id1, id2, id3, id4; // used to track different stages of id value

// retrieve an announcement persister from each persistence manager
AnnouncementDbPersister persister1 =
  (AnnouncementDbPersister) bbPm1.getPersister( AnnouncementDbPersister.TYPE );
AnnouncementDbPersister persister2 =
  (AnnouncementDbPersister) bbPm2.getPersister( AnnouncementDbPersister.TYPE );

// retrieve default announcement id value - assigned by the framework
Announcement ann = new Announcement();
id1 = ann.getId();
```

```
// now fill in all the attributes of announcement...
ann.setTitle( "Announcement Title" );
...

// now persist the announcement to container 1...
// Because the current announcement id value (default id value) is an invalid id
// for container 1, persister performs a database "insert" operation and generates
// a new container specific id value.
persister1.persist( ann );
id2 = ann.getId();

// update some properties (NOT id) and re-persist to container 1...
// Because current id IS valid for container 1, persister performs a database
// "update" operation. Current announcement id value is left as is.
ann.setTitle( "New Announcement Title" );
persister1.persist( ann );
id3 = ann.getId();

// take the same announcement as above and directly persist to container 2...
// Because the current announcement id value (id value assigned from the persist
// to container 1 operation) is an invalid id for container 2, persister performs
// a database "insert" operation and generates a new container specific id value.
persister2.persist( ann );
id4 = ann.getId();

// when done, the following id relationships exist:
// id1 is not equal to any other id value
// id2 and id3 are the same id value
// id4 is not equal to any other id value
```

Using Data Objects and Persistence

Introduction

With the information that has been presented about data objects and persistence, a full life-cycle example can be created demonstrating the process of loading and manipulating objects such that the changes appear within the *Blackboard Learning System*.

Example

The following code sample demonstrates how to update a course within the *Blackboard Learning System*. It assembles what has been discussed in this document concerning data objects and persistence along with a few more details to create a fully working example. For the below sample, assume these operations are being performed within a JSP page and the Id of a Course has been passed in on the URL using `Id.toExternalString()`. The example also assumes that other values of the Course have been passed in on the URL (or submitted via a form) with appropriate names.

```
BbPersistenceManager bbPm =
    BbServiceManager.getPersistenceService().getDbPersistenceManager();

try
{
    // process URL/form data
    String strCourseId = request.getParameter( "course_id" );
    String strTitle     = request.getParameter( "title" );
    String strDescription = request.getParameter( "description" );
    String strPace      = request.getParameter( "pace" );

    // generate objects from the strings retrieved above where necessary...
    // -> generate an actual Id value from the string course id provided
    // -> generate an actual Course.Pace value from the string pace provided
    Id courseId = bbPm.generateId( Course.COURSE_DATA_TYPE, strCourseId );
    Course.Pace pace = Course.Pace.fromExternalString( strPace );

    // retrieve both a course loader and persister
    CourseDbLoader loader = (CourseDbLoader) bbPm.getLoader( CourseDbLoader.TYPE );
    CourseDbPersister persister =
        (CourseDbPersister) bbPm.getPersister( CourseDbPersister.TYPE );

    // load the specified course and update it using the passed in values
    Course course = loader.loadById( courseId );
    course.setTitle( strTitle );
    course.setDescription( strDescription );
    course.setPaceType( pace );

    // save the changes we made to the course back to the container
    persister.persist( course );
}
catch ( KeyNotFoundException knfe )
{
    // handle appropriately
}
course load failure -- handle appropriately
}
catch ( ValidationException ve )
{
    // handle appropriately
}
```

```
}  
catch ( PersistenceException pe )  
{  
    // handle appropriately  
}
```

Exception Handling

Introduction

When using the Building Blocks APIs and Runtime, a number of exceptional situations can occur that must be managed.

Exceptions

The following table describes some of the most common exceptions.

Exception	Description
<code>blackboard.persist.PersistenceException</code>	Nearly all operations involving a container, including loading or persisting to a container, can result in a <code>blackboard.persist.PersistenceException</code> being thrown. While this exception can mean a specific problem has occurred, it sometimes acts as a catchall exception within the framework and can include other more specific exceptions (see <code>blackboard.base.NestedException</code>).
<code>blackboard.persist.KeyNotFoundException</code>	This exception is thrown by database container load routines when a requested object cannot be found. This may mean that an object with a given ID value could not be located or a provided ID value was invalid for the container. This exception inherits from <code>PersistenceException</code> and therefore can be handled in the same catch block.
<code>blackboard.base.ValidationException</code>	This exception is thrown when the contents of a data object are determined to be invalid. While object validation is typically done during a persist operation, validation can be manually invoked on any data object (see <code>BbObject.validate()</code> for more information). The validity of an object is not container specific. Validity typically involves field value ranges (for example, an integer attribute must be positive) or an attribute relationship (for example, Field A must have a value if Field B is set to X).

Runtime Environment

Overview

One important aspect of developing for the *Blackboard Academic Suite* is the way that APIs are used in the context of particular service requests. In the application, service requests usually correspond to HTTP requests from a browser. For example, Context is used in the application server to parse out information about a current user or a virtual installation without having to explicitly identify that data.

Virtual Installations

This feature allows several logical installations of *Blackboard Academic Suite* to co-exist in a single physical installation. The runtime environment enforces data segregation between the instances by partitioning the actual persistence containers. Thus, when code queries the Service Manager for the Persistence Manager, the instance returned is bound only to the Virtual Installation currently being accessed.

The runtime knows which Virtual Installation is being accessed through the use of Context. This is very important, as it negotiates the specifics of Container and Persistence Manager, discussed above.

Context

The state of any user interaction with the system has two components: request data, which is particular to a specific HTTP request, and session data, which will span a set of requests. Session data include information about the user currently logged in while request data can encompass a much wider range of information.

Formally, Context represents the combination of Session data and data in a single client request. This is because some types of information cannot be assumed session-wide, such as the course currently being accessed. Consider the case of a user who opens a second browser window to access a course. Assuming that the course is session-wide would lead to confusion as the user moves from browser to browser. Thus, course is an explicit request parameter. Context is simply a unified point of access for this.

Additionally, Context is used by most of the Building Block APIs to transparently negotiate items. Context must be set before any API methods are called.

Using Context

The `blackboard.platform.context.ContextManager` object is used to set and release context. The `Context` object also will contain information about the current user, course being accessed, and so forth. See `blackboard.platform.context.Context` for more information.

```
ctxMgr = (ContextManager)BbServiceManager.lookupService( ContextManager.class );
Context ctx = ctxMgr.setContext(request);

//... work with the APIs

if( ctxMgr != null ) {
    ctxMgr.releaseContext();
}
```

Note: Servlet and JSP code can use the `<context>` tag from the `bbData` tag library.

Blackboard Content System APIs

Overview

The Blackboard Content System APIs can be used to programmatically access and manipulate aspects of the Blackboard Content System. The Blackboard Content System must be installed in order to access the APIs. Building Blocks can be created for the Blackboard Learning System, Blackboard Community System, or Blackboard Content System which take advantage of these APIs. The APIs allow for a wide variety of access to everything from the creation and management of files and folders to their metadata, properties, and permissions. The APIs also enable programmatic access to many of the tools within the Blackboard Content System, such as the Learning Objects catalog, Portfolios, and Workflows.

Blackboard Content System Data Object Packages

Introduction

The data object definitions reside in the various sub-packages of `blackboard.cms`. The sub-packages are grouped into various functional areas like `FileSystem` or `Portfolio`.

Data Object sub-packages

The data sub-packages are described in the table below.

Data object sub-packages	Classes
<code>bookmark</code>	Contains objects associated with Bookmarks in the system.
<code>filesystem</code>	Contains objects used by the filesystem subsystem. This is one of the primary sub-packages within the Blackboard Content System, and contains the objects which control files and directories as well as the "CSContent" object which is part of the Blackboard Content System permissions framework.
<code>lrngobj</code>	Contains objects for interacting with the Learning Object Catalog.
<code>metadata</code>	Contains classes for manipulating metadata templates.
<code>portfolio</code>	Contains classes for interacting with the Portfolio system.
<code>workflow</code>	Contains classes for interacting with the Workflow system.

Blackboard Content System Supported Data Objects

Setup

In order to correctly access many of the objects and methods used by the Blackboard Content System APIs, several permissions must be enabled in the `bb-manifest.xml` file. For more information on that file, refer to the *Building Block Developer Guide*.

In the permissions section of that file, ensure the following exists:

```
<permission type="java.lang.RuntimePermission" name="db.connection.*" />
<permission type="attribute" name="user.authinfo" actions="get,set" />
<permission type="attribute" name="user.personalinfo" actions="get,set" />
<permission type="persist" name="user" actions="create,modify,delete" />
<permission type="persist" name="userrole" actions="create,modify,delete" />
<permission type="persist" name="course" actions="create,modify,delete" />
```

Data object descriptions

The following table describes a subset of the supported data objects and a brief description of the Blackboard Content System entity that they are designed to represent.

Entity	Description
CSEntry	This is the base class for objects (file and folders) in the filesystem. It contains the access control entries, location, size, and metadata for objects. See package <code>blackboard.cms.filesystem</code>
CSFile	This class represents files on the filesystem. It contains the method for setting and getting the actual data within a file, as well as, the type of file. See package <code>blackboard.cms.filesystem</code>
CSDirectory	This class represents a directory in the filesystem. Methods are available for managing quotas for each directory. These quotas are the total space allotted by the owner for all files and sub-directories contained in the directory. See package <code>blackboard.cms.filesystem</code>
CSCustomDirectory	This class encapsulates "Special" directories in the filesystem. More specifically, the directory contents of a <code>CSCustomDirectory</code> cannot be changed, as they are managed explicitly by Blackboard Content System. For instance, the <code>/users</code> , <code>/courses</code> directories are type <code>CSCustomDirectory</code> See package <code>blackboard.cms.filesystem</code>

Entity	Description
CSContext	<p>This key object controls the context used for determining permissions to entries, as well as, controls management of transactions. A context is the top level state manager for a series of Blackboard Content System calls. For every set of actions grouped together into one transaction, a new context is created and used. A context instance contains transactional state for that request (for example, commit or rollback). All permission checking is done through calls on the context object.</p> <p>See package <code>blackboard.cms.filesystem</code></p>
CSAccessControlEntry	<p>The CSAccessControlEntry class provides a representation of an Access Control Entry within the Blackboard application. An object exposing the individual permissions of a file/directory for a specific user or list of users. Methods exist to both read and set the permissions.</p> <p>See package <code>blackboard.cms.filesystem</code></p>
LOItem	<p>The LOItem class provides a representation of a Learning Object Item within the Blackboard application. Learning Object Items usually refer to a physical file or folder within the Blackboard Content System, with additional metadata and status information related as metadata.</p> <p>See package <code>blackboard.cms.lrnobj</code></p>
LOCategory	<p>The LOCategory class provides a representation of a Learning Object category within the Blackboard application.</p> <p>See package <code>blackboard.cms.lrnobj</code></p>
Portfolio	<p>The Portfolio class provides a representation of a Portfolio within the Blackboard application.</p> <p>See package <code>blackboard.cms.portfolio</code></p>
Workflow	<p>The Workflow class provides a representation of a Workflow within the Blackboard application. A Workflow must have a type associated with it.</p> <p>See package <code>blackboard.cms.workflow</code></p>

Using Blackboard Content System Data Objects

Introduction

Blackboard Content System data objects behave in the same way as Blackboard Learning System data objects. Refer to the [Blackboard Content System Supported Data Objects](#) section for instructions on how to instantiate and use the methods of these objects.

Creating a data object

The following code demonstrates how to create a Portfolio object.

```
Portfolio portTest = new Portfolio();
```

Creation of a data object in this way results in an object with default values for all of the object's attributes. To create a more useful `Portfolio`, it is necessary to create one that contains more than default values. To do this, create a default `Portfolio` object and call all of the appropriate "set" methods provided by the object.

Data object IDs

Blackboard Content System objects use the same ID framework as Blackboard Learning System objects.

Persistence and Managers

The Blackboard Content System uses a simplified approach for loading, deleting, and saving objects. Each of the core data objects for Portfolio, Metadata, Learning Objects, Bookmarks, and Workflows has an associated Manager.

For example, the `Portfolio` has a `PortfolioManager` class which has methods to `deleteById`, `loadById`, `loadAccessibleByCourseId`, `loadByOwner`, `save`, and more. The methods within the managers are static, so they can be accessed directly.

For example, to load a Portfolio, make a change to it, and then save it:

```
Portfolio portLoaded = PortfolioManager.loadById(portId);
portLoaded.setTitle("Updated Title");
PortfolioManager.save(portLoaded);
```

Using Blackboard Content System File System Objects

Introduction

Blackboard Content System filesystem objects work in a slightly different way than the other Blackboard API objects. For filesystem access, a Content System Context must first be established. This context is used to group transactions, as well as handle permission checking.

Creating a Content System Context

A context is the top level state manager for a series of Blackboard Content System calls. For every set of actions grouped together into one transaction, a new context is created and used. A context instance contains a transactional state for that request (for example, commit or rollback). All permission checking is done through calls on the Context object.

Without a Context, whenever an exception is thrown, the state of the transactions involved would be unknown. It would not be possible to know which transaction(s) needed to be committed or rolled back. The context object is used to keep track of that state and allow or block commit attempts. Thus, the context object can be used to find out if the current transaction can be committed or not.

It is extremely important when using the context object to ensure that you properly use a try{ }, catch{ }, finally{ } block to commit() or rollback() each transaction. Not doing so properly may cause deadlocks, performance degradation and unexpected behavior.

It is recommended to always use CSContext in the following way:

```
CSContext ctxCS=null;
try {
    ctxCS = CSContext.getContext();

    //your code here

} catch (Exception e) {
    ctxCS.rollback();
} finally {
    if (ctxCS!=null) {
        ctxCS.commit();
    }
}
```

The CSContext.getContext() call also has the ability to load context "as a specific user." This gives the programmer the ability to determine if that user has sufficient privileges to operate on the designated CSEntry. For some operations, the standard user context may be inadequate to perform the desired operation. In that case, there is a CSContext.isSuperUser() method that can be used. Setting this to "true" and then performing the operation is useful under certain conditions.

Using the Content System Context

Once a CSContext is established. It is used to create new CSFiles and CSDirectories. It is also used to load existing files and directories using the findEntry() method. The CSContext is further used to check if permissions for the

current loaded context `canRead()`, `canWrite()`, `canManage()`, and `canDelete()` a specific `CSEntry`.

Using the Content System Context to manage File System objects

The `CSText` object is used to load, save, create, and manage the `CSEntries` in the file system.

The following is an example of how the context would be used to load a directory and manipulate it.

```
// Loads the CSDirectory object /users/dcane
CSDirectory csDirMyUserDir = ctxCS.findEntry("/users/dcane");

// Gets the contents of the directory as a List
List listDirectoryContents = csDirMyUserDir.getDirectoryContents();

// Iterate through the list and print out information on the contents
Iterator iterDirectoryContents = listDirectoryContents.iterator();
while (iterDirectoryContents.hasNext())
{
    // the List of a directory contains CSEntry items (which can be other dirs or files)
    CSEntry csEntryItem = (CSEntry) iterDirectoryContents.next();
    out.println("Hello there, I have found: " +csEntryItem.getFullPath());
}
```

The `CSText` also does some special loading for course directories, user directories, and `eReserve` directories. These loaders will return `CSDirectory` objects when you pass in the corresponding user or course.

For example, to load a course directory in the Blackboard Content System when the Blackboard Learning System `blackboard.data.course` object is present, run the following:

```
CSDirectory courseDir = ctxCS.getCourseDirectory(theCourse);
```

Understanding Permissions and Principals

The access control system for files and folders in the Blackboard Content System uses the concept of principals tied to permissions. A "principal" is an entity with the base class of `CSPrincipal`. This interface defines the concept of a principal, encompassing any entity to which file system access permissions may be granted. Every principal must be uniquely identified by a principal ID. All Blackboard Content System user and group objects extend this object.

These objects are found in the `blackboard.cms.filesystem.security` package.

Subclasses of this include: `CourseGroupPrincipal`, `CoursePrincipal`, `CourseRolePrincipal`, `PortalRolePrincipal`, `PortfolioPrincipal`, `SystemRolePrincipal`, `UserPrincipal`. These encapsulate the primary ways people are associated with permissions.

A look at one in more detail helps explain this concept. If the goal was to give all Students in the "math10" course the access control of Read – the developer would first use the above information to retrieve a `CSEntry` with the desired file, and then determine the `CourseRolePrincipalID` used to represent the class of users associated in a course with the specific role.

```
// Load the file
```

```
CSEntry csFileToChange = ctxCS.findEntry("/user/dcane/foo.txt");

// determine the principal ID
String strCourseRolePrincipal = CourseRolePrincipal.calculatePrincipalID(courseMath101,
courseRoleofStudent);

// get the Access Control Entry for that principal ID (even if it does not exist)
CSAccessControlEntry csACE =
csFileToChange.getAccessControlEntry(strCourseRolePrincipal);

// then update the permission for "read"
csACE.updatePermission(true, false, false, false);
```

Remember, this all needs to happen inside of a try{} block where context is established, and that the current context has permissions to update that entry's permissions.
