Extending the ATG Platform Rel 10.1.2

Student Guide

D81525GC10
Edition 1.0
May 2013
D82003

ORACLE

Author
Karin Layher

Technical Contributors and Reviewers
Jay Cross

Tom Hardy

Editors
Richard Wallis

Malavika Jinka

Graphic Designer
Seema Bopaiah

Publishers
Srividya Rameshkumar

Veena Narasimhan

Contents

1

Developing Component Classes

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Course Goals

After completing this course, you should be able to:

- Create custom ATG component classes that can
  - Manage repositories and transactions
  - Handle form input and output
- Model complex data relationships in repositories
- Fine-tune the configuration of repository caching to improve performance
- Create custom servlet beans (droplets) to display content

Student Introductions

Introduce yourself to your classmates and instructor by answering these questions:

- What is your name?
- Where do you work? What is your role there?
- How will your organization be using ATG Web products?
- How much experience do you have with Java or with object-oriented design and development?
- When did you take the prerequisite courses? Have you had a chance to apply what you learned in them?
- Why are you taking this course?

Agenda

Session	Lesson
Day One: Custom ATG Components	Lesson 1: Developing Component Classes Lesson 2: Repository API Lesson 3: Custom Form Handlers Lesson 4: Managing Transactions
Day Two: Complex Repository Relationships	Lesson 5: Complex Repository Configuration Lesson 6: Advanced SQL Repository Configuration Lesson 7: Derived Properties Lesson 8: SQL Repository Caching Lesson 9: Custom Droplets

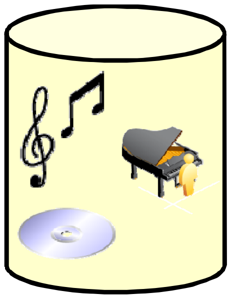
Road Map

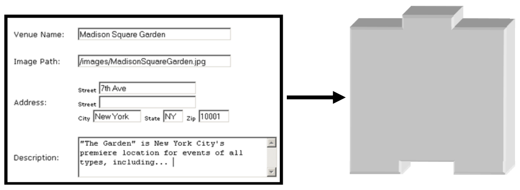
- Review
- Developing component classes

ORACLE

Review: Core Concepts

In *Foundations of ATG Application Development*, you learned about the following core concepts:


Repositories


Venue Name: Madison Square Garden
Image Path: /images/MadisonSquareGarden.jpg
Address: Street 7th Ave
Street
City New York State NY Zip 10001
Description: "The Garden" is New York City's premiere location for events of all types, including...

Form handlers

Nucleus components

User profiles

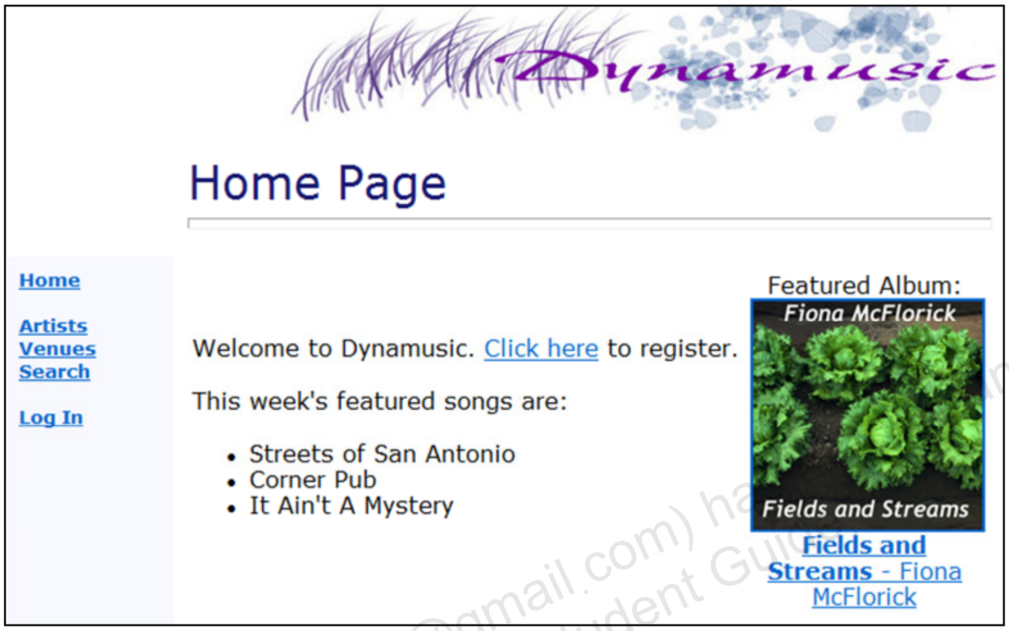
You created Java components from Java classes to perform functions such as data storage and retrieval, personalization with business rules, and business logic.

You configured repositories by using ATG's Data Anywhere Architecture to manage structural data and objects, such as profiles, artists, albums, and songs.

You edited the User Profile object to contain fields specific to the Dynamusic site.

You created dynamic webpages to control the presentation by using JSP technology and the DSP tag library.

Review: Dynamusic.com



Welcome to Dynamusic. Click here to register.

This week's featured songs are:

- Streets of San Antonio
- Corner Pub
- It Ain't A Mystery

You may remember Dynamusic.com, a fictional company that enables users to download MP3s. The site generates revenues from monthly subscription fees, referral fees for album purchases from a retail site, and advertising for concerts and venues. You added functionality to Dynamusic as you learned about the tools that ATG provides. You continue to use this site for your practices in this course.

Dynamusic features include the ability to:

- Browse artists, albums, and songs
- Search for songs based on title or description
- Browse and search for concerts and venues
- Read and write album, song, and concert reviews
- Select preferred genres (jazz, classical, and so on) that can be used for song recommendations, as well as for finding other users with similar interests.
- Share user profile information, including a user "description," with other users. Dynamusic forwards an email preserving user privacy.
- Upload songs to share with interested users (for registered users who are amateur musicians)

Review: Dynamusic Functionality

In building Dynamusic 1.0, you did the following:

- Created a module to hold and encapsulate the Dynamusic application, including the Java EE application
- Built different types of components. Some examples are:
  - Generic: `FeaturedSongs`
  - Servlet Bean: `ArtistLookup`
- Extended the profile to add the `shareProfile` and `viewedArtists` properties
- Created the Events Repository with two item types: `venue` and `concert`
- Created dynamic pages for the Songs Repository to enable users to browse through artists, albums, and songs as well as to add new songs

Modules enable you to encapsulate functionality and pages into one place. You can then layer modules on top of one another by specifying dependencies. In *Foundations of ATG Application Development*, you created a module called Dynamusic that depended on the Dynamusic-Base module. Dynamusic held the Java EE application, including the web application with the JSPs. You used the module to build new functionality as well as to make changes to the underlying functionality.

When working with components, you learned how to create components in Eclipse and in a text editor by using `.properties` files. Some example components that you built were:

- `FeaturedSongs`: To hold the songs you are featuring on Dynamusic
- `ArtistLookup`: A servlet bean that looks up a repository object by using the object's ID

Dynamusic Revision 2.0

Dynamusic wants to release version 2.0 of the website.

- Many new features are needed to make the application more robust.
- Primary goal: Attract new users and retain existing users.

Road Map

- Review
- Developing component classes

Objectives

After completing this lesson, you should be able to:

- Describe the types of custom component classes you need to create in your application
- Create custom component classes
- Integrate custom components into Nucleus
- Use ATG's application-logging facilities in a component class

Custom Component Classes

- Although ATG provides many useful components and classes, your application probably needs functionality that is not provided by ATG.
- ATG can easily incorporate the functionality provided in components by using custom Java classes.

Dynamusic Custom Component Classes

Dynamusic needs custom classes for many functions, such as:

- Managing low-level repository operations
  - Maintaining item interrelationships
- Handling complex data input forms
- Uploading mp3 files and saving associated repository information
- Triggering periodic “cleanups” of the Events Repository for past events

Types of Components

Most components in ATG can be classified into one of the following categories:

- Service: A component that provides functionality on which one or more systems depend (for example, an MP3 parsing service)
- Form Handler: A component to validate and manage form input data
- Servlet Bean: A droplet to handle page display logic

This lesson focuses on services. Form handlers are covered in the lesson titled "Custom Form Handlers." Writing custom servlet beans is covered in *ATG Personalization and Scenario Development*.

An example of a service component is a scheduled service. Your server-wide application may have tasks that need to be performed on a periodic or recurring basis. For example, a component in the application may need to clear a cache every 10 minutes, send email at 2:00 AM every day, or rotate a set of log files on the first day of every month.

Dynamo includes a Scheduler service, `atg.service.scheduler.Scheduler`, which keeps track of scheduled tasks and executes the appropriate services at specified times. Administrators can see a list of all scheduled tasks by looking up `/atg/dynamo/service/Scheduler` in the Component Browser. For more information about scheduled services, refer to the "Core Dynamo Services" chapter of the *ATG Programming Guide*.

Java Development in ATG

- To use your custom classes, you must add your classes to your module's CLASSPATH.
- Each module can specify one or more directories or .jar files to add to ATG's CLASSPATH in the manifest file (`<module-folder>/META-INF/MANIFEST.MF`).

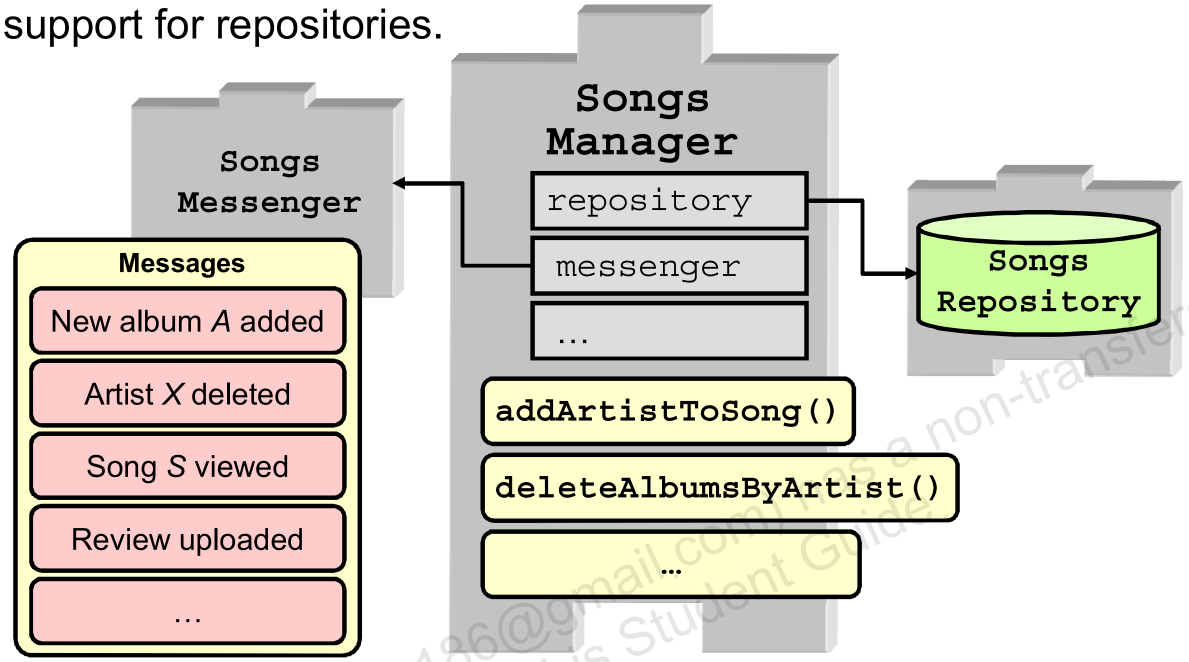
```
Manifest-Version: 1.0
ATG-Required: DAS DPS DSS
ATG-Config-Path: config/
ATG-Class-Path: classes/ lib/classes.jar
```

ATG components are based on Java classes. If your module includes components or other objects based on custom classes that you supply, you must add the ATG-Class-Path line to your manifest.

Classes can either be in a `.jar` file or be individual `.class` files. The example in the slide shows both. When a project is ready to deploy, classes are typically packaged in a `.jar` file. During development (and during this course), classes are left as separate files in the `classes/` directory.

Dynamusic Repository Support Classes

The repository “manager” component provides back-end support for repositories.



A good common practice is to create a “manager” or “tools” component that provides a centralized location to handle all the data associated with a repository. This component maintains pointers to relevant other components and provides helpful utility methods.

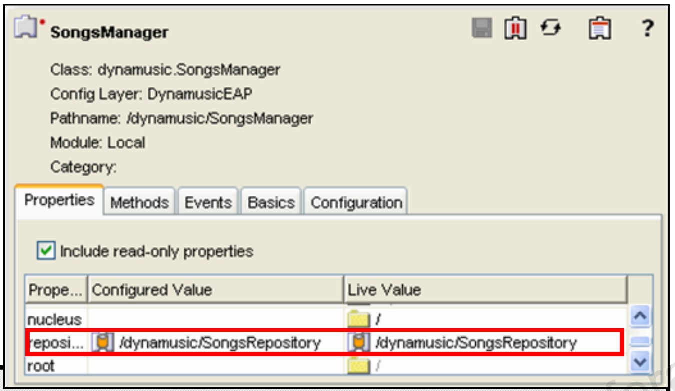
Repository Managers are designed to provide the context for methods that operate on a repository. Methods on the manager class encapsulate the low-level details involved in common business logic. You can use this component to reach methods needed by form handlers, custom scenario actions, servlets, and so on. In the next few lessons, you will see how a `SongsManager` component can be used to manage access to the Songs Repository by maintaining complex item relationships and managing transactions.

“Manager” components such as these are often accessed by custom form handlers. In the lesson titled “Repository API,” you will see how the `UploadSongHandler` uses the `SongsManager` component.

A messenger typically exists to supplement the manager. If there are methods in the manager that should send a JMS message, it might call methods from the messenger to send a specific message. Messaging is covered in the *ATG Personalization and Scenario Development* course, but you should start thinking about the role that messaging plays in your application design now. Messages are how your application communicates with other systems and, in particular, with the Scenario Manager.

JavaBeans

- ATG components are JavaBeans.
- Properties are implemented with `get` and `set` methods.



SongsManager

Class: dynamusic.SongsManager
Config Layer: DynamusicEAP
Pathname: /dynamusic/SongsManager
Module: Local
Category:

☑ Include read-only properties

Prope...	Configured Value	Live Value
nucleus		/
reposi...	/dynamusic/SongsRepository	/dynamusic/SongsRepository
root		/

SongsManager.java

```
public class SongsManager {
  private Repository mRepository = null;
  public Repository getRepository() {
   return mRepository;
  }
  public void setRepository(Repository pRepository) {
   mRepository = pRepository;
  }
}
```

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This sample code (and the rest of the implementation of the `SongsManager` component used in the remaining lessons) can be found in `examples/lesson01/SongsManager.java`.

Full documentation for the JavaBeans specification can be found at Oracle's Java site at http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html.

The most important aspects of JavaBeans that are relevant to ATG are the following:

- Each property must have a corresponding set and/or get method.
  - `public type getProperty()`
  - `public void setProperty(type)`
- Boolean properties can be implemented with `isProperty()` rather than `getProperty()`.
- Each class must include a default (no-argument) constructor. The no-argument constructor for `SongsManager` is not shown in the slide example because it is not written explicitly. Recall that, in Java, if no constructors are written as part of the class, a default constructor (a constructor with no arguments that does nothing) is provided automatically. In the example, no initialization of the object is required, so no explicit constructor is required. The implicit constructor is sufficient to meet the JavaBeans requirement of having a no-argument constructor.

GenericService

- ATG provides a base class that provides smooth integration into the ATG Nucleus: `atg.nucleus.GenericService`
- Properties include
  - `loggingDebug`
  - `loggingError`
  - `loggingInfo`
  - `loggingWarning`
  - `name`
  - `nameContext`

The logging level set by the developer

The name of the component in Nucleus

ORACLE

GenericService Methods

Useful methods of `GenericService` include:

- `resolveName()`: Finds a component by its Nucleus path name
- `isRunning()`: Tells what the current status of this component is
- `doStartService/doStopService()`: Are run when the component is started or stopped by Nucleus
  - These are methods to be overridden by the subclass.

A `GenericService` component can find out information about the environment in which it is running. It also has `doStartService` and `doStopService` methods to perform any setup or cleanup operations. For instance, if you had a component that established a connection with a third-party system, `doStartService()` might create and maintain a socket connection, and `doStopService()` would cleanly terminate the connection.

Nucleus Naming Services

There are two ways a component or object can locate another component.

The simpler and more common way is through a linked property. When a property refers to another component path name (such as "`/dynamusic/TopSongs`"), Nucleus resolves the component reference into an object reference, and stores that in the property. When the property is used, the object reference is there and ready for use.

Occasionally, however, a component needs to reference another component for which it has no property and thus no object reference. In that case, Dynamo provides the `resolveName(String)` method, which takes a Nucleus path name and returns an object reference. This is a useful method, but it should be used only when necessary because it is very slow.

The preferred way for one component to refer to another is by using a linked property so that the name resolution occurs only when the property is set rather than every time it is required.

Application Logging

- Custom components should take advantage of ATG's logging system.
- The logging methods are:
  - `logDebug`
  - `logError`
  - `logInfo`
  - `logWarning`

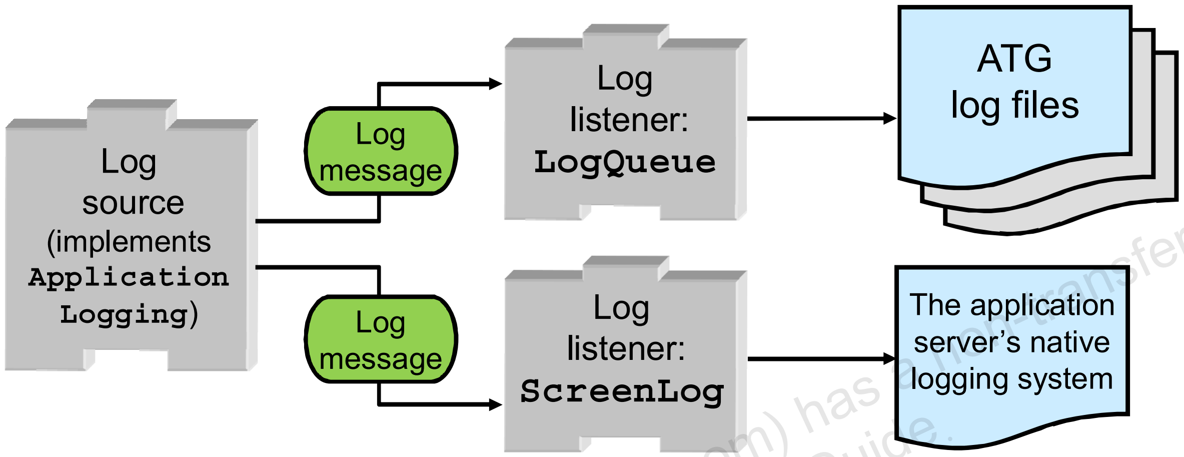
You used the logging capabilities of built-in components in the *Foundations of ATG Application Development* course. As a result, you learned that logging is very useful. Your components should make use of the logging functionality of `GenericService` (or other component superclasses).

Each of the logging methods can take a string, a `Throwable` (such as an exception), or both. See the *API Platform Reference* for details.

The logging-related facilities of `GenericService` are actually an implementation of the `atg.nucleus.logging.ApplicationLogging` interface. If your component extends a non-ATG base class, and you are therefore unable to use `GenericService` as a base, you can still take advantage of ATG logging by implementing this interface yourself.

Logging Implementation

ATG application logging is implemented by using the Java event listener model.


Log source (implements ApplicationLogging)
Log message
Log message
Log listener: LogQueue
Log listener: ScreenLog
ATG log files
The application server's native logging system

Each log source (such as a custom component that extends `GenericService`) has a property pointing to a set of log listeners. The log message is sent to the log listeners, which in turn pass the log on to whatever output they are configured to use. By default, two log listeners are configured:

- **`/atg/dynamo/service/logging/LogQueue`:** Outputs to the log files
- **`/atg/dynamo/service/logging/ScreenLog`:** Passes the logging to the logging infrastructure of the application server by using Jakarta Commons Logging (JCL) as a bridge

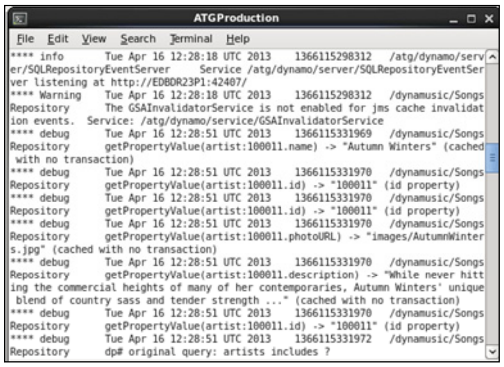
By default, the `LogQueue` outputs to `<atgdir>/home/logs` or `<atgdir>/home/servers/<servername>/logs`.

ATG log files are divided by message type:

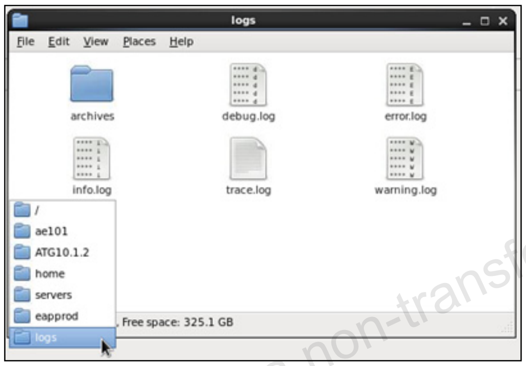
- Debug messages into `debug.log`
- Info messages into `message.log`
- And so on

Logging Messages in the ATG Console

- You can find the ATG logging messages on the console or in the specific ATG server's logs directory.



```
**** info      Tue Apr 16 12:28:18 UTC 2013    1366115298312   /atg/dynamo/serv
er/SQLRepositoryEventServer     Service /atg/dynamo/server/SQLRepositoryEventSer
ver listening at http://EDBDR23P1:42407/
**** Warning   Tue Apr 16 12:28:18 UTC 2013    1366115298312   /dynamusic/Songs
Repository     The GSAInvalidatorService is not enabled for jms cache invalidat
ion events.  Service: /atg/dynamo/service/GSAInvalidatorService
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331969   /dynamusic/Songs
Repository     getPropertyValue(artist:100011.name) -> "Autumn Winters" (cached
 with no transaction)
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331970   /dynamusic/Songs
Repository     getPropertyValue(artist:100011.id) -> "100011" (id property)
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331970   /dynamusic/Songs
Repository     getPropertyValue(artist:100011.id) -> "100011" (id property)
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331970   /dynamusic/Songs
Repository     getPropertyValue(artist:100011.photoURL) -> "images/AutumnWinter
s.jpg" (cached with no transaction)
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331970   /dynamusic/Songs
Repository     getPropertyValue(artist:100011.description) -> "While never hitt
ing the commercial heights of many of her contemporaries, Autumn Winters' unique
 blend of country sass and tender strength ..." (cached with no transaction)
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331970   /dynamusic/Songs
Repository     getPropertyValue(artist:100011.id) -> "100011" (id property)
**** debug     Tue Apr 16 12:28:51 UTC 2013    1366115331972   /dynamusic/Songs
Repository     dp# original query: artists includes ?
```



- ATG logging messages that are sent to the `ScreenLog` component are passed to the application server for handling via the JCL bridge.

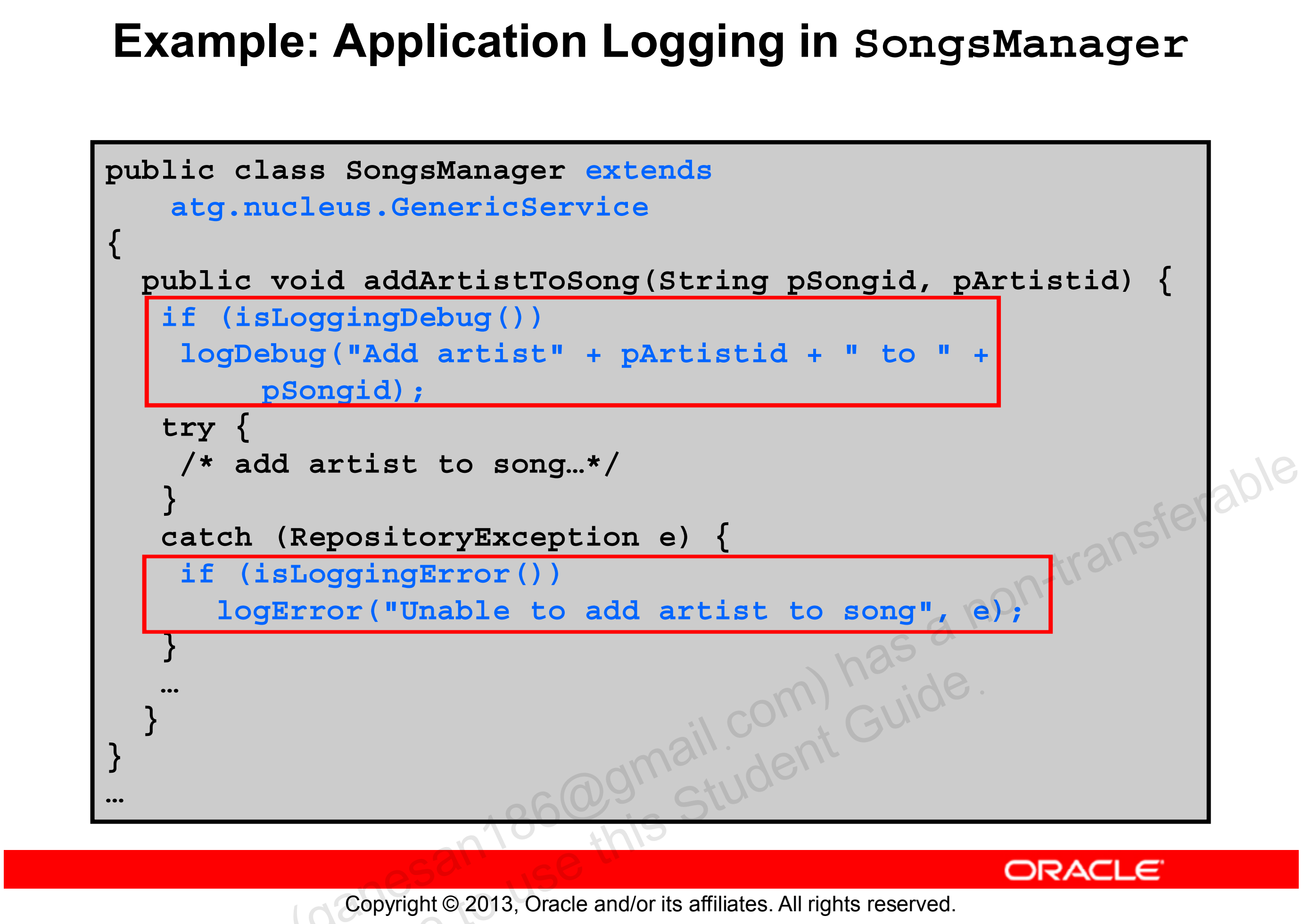
The two screenshots in the slide show the logging messages on the ATG server console and the location of the log files in the ATG file structure from your practice environment. You can find the files in your `<atgdir>/home/servers/<myServerName>/logs` folder.

ATG logging messages sent to the `ScreenLog` component are passed to the application server for handling via the JCL bridge. For example, JBoss uses log4j to manage logging output, configured by the following XML file:

`<jbossroot>/server/<servername>/conf/log4j.xml`

Oracle WebLogic Server can also be configured to use log4j. You can configure your application server to send the messages to a particular location, and you can control which messages are logged.

Where the logging files go in WebLogic depends on how you compile your EAR. If you compile your EAR in the default development mode, your application uses ATG's home directory structure. If you compile the EAR in stand-alone mode (as is typical in a production environment), the ATG applications log their messages to files in the `ATG-Data` directory. The name and location of this directory are configurable, but it is located by default in ATG's WebLogic domain (`MW_home/user_projects/domains/atg_education/ATG-Data`, for example). The `ATG-Data` directory corresponds to the `home` directory in an ATG installation.



Example: Application Logging in `SongsManager`

```
public class SongsManager extends
    atg.nucleus.GenericService
{
  public void addArtistToSong(String pSongid, pArtistid) {
   if (isLoggingDebug())
    logDebug("Add artist" + pArtistid + " to " +
        pSongid);
   try {
    /* add artist to song…*/
   }
   catch (RepositoryException e) {
    if (isLoggingError())
      logError("Unable to add artist to song", e);
   }
   …
  }
}
…
```

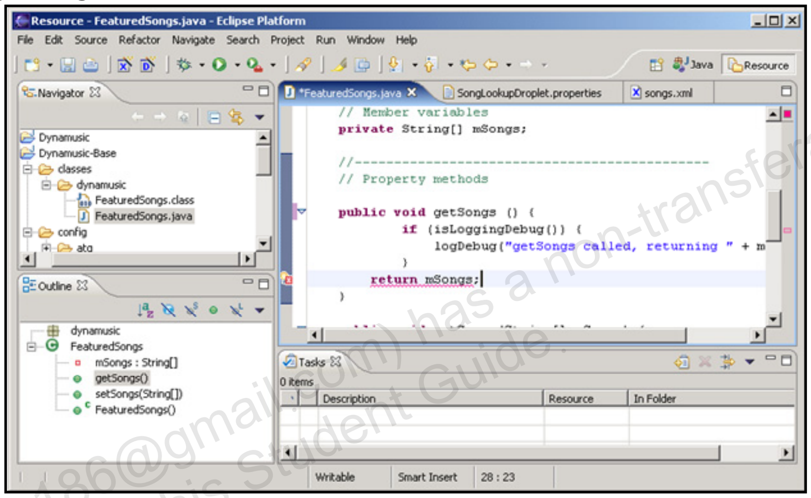
ORACLE

As mentioned in previous slides, one way to add application logging to your application is to extend `atg.nucleus.GenericService`. You implement the particular logging method that you want to use.

The example in the slide defines two logging methods: one with a debug message and one with an error message. The code in the slide shows the best practice for coding logging messages. Before sending a log message, a component should check whether logging is enabled for that log message's level (for example, `loggingDebug=true`) to avoid unnecessary overhead.

Working with ATG, Java, and Eclipse

- It is recommended that you use Eclipse as an IDE.
- Eclipse provides a Java editor that has:
  - Context-sensitive help
  - Syntax highlighting
  - Automatic compilation
  - Debug support



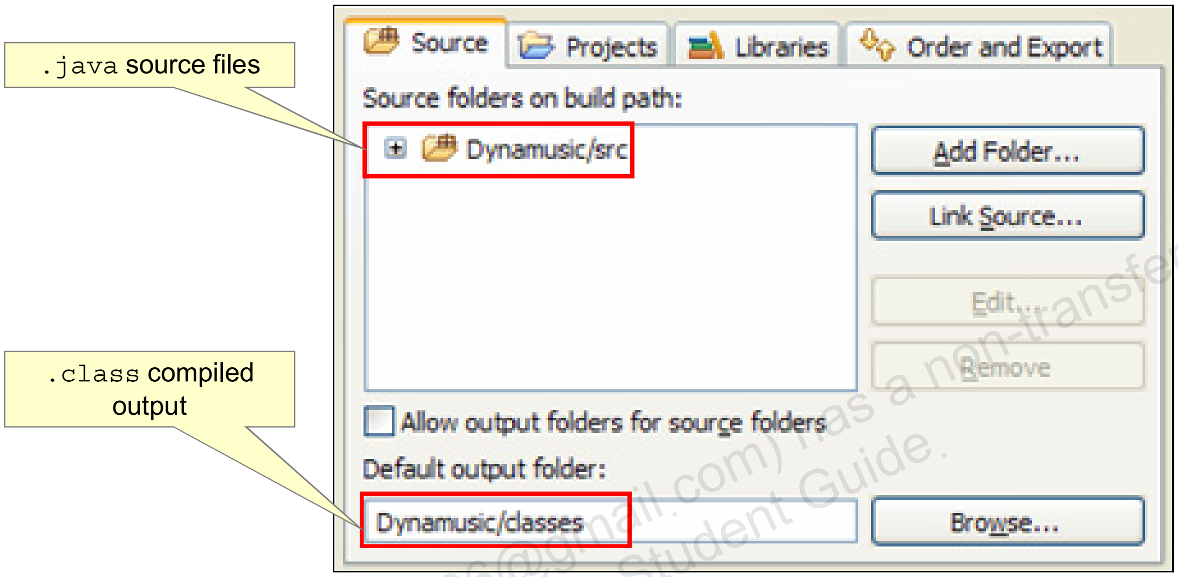
```
// Member variables
private String[] mSongs;

//--------------------------------------------
// Property methods

public void getSongs () {
        if (isLoggingDebug()) {
            logDebug("getSongs called, returning " + m
        }
    return mSongs;
}
```

Eclipse has a built-in Java editor that provides many powerful features for Java development, such as syntax highlighting, source code analysis, automatic compilation, generation of stub code for JavaBeans, and so on. Covering the details of Eclipse is beyond the scope of this course, but you are encouraged to explore on your own by using the Eclipse help utility. You will be using Eclipse in your practice environment.

Note: ATG provides a plug-in for handling ATG objects in Eclipse. This plug-in ships with ATG 10.1.2 and can be installed in Eclipse. The plug-in is a beta version. For instructions on how to install the ATG plug-in in Eclipse, see “Appendix D: Development Tools for Eclipse” in the *ATG Installation and Configuration Guide*.

Configuring Eclipse with ATG

You must specify the Java build path.



Source folders on build path:

Dynamusic/src

☐ Allow output folders for source folders

Default output folder:

Dynamusic/classes

The practice instructions walk you through setting up the Java build path for your DynamusicEAP module.

Demonstration: Creating a Java Class in Eclipse

Quiz

Which one of the following statements best describes the ATG logging system?

a. To take advantage of ATG's logging system, you extend `atg.nucleus.GenericService`.

b. If your custom class extends a non-ATG base class, you can take advantage of ATG logging by implementing the `atg.nucleus.logging.ApplicationLogging` interface.

c. Before sending a log message, a component should check whether logging is enabled for that log message's level.

d. All of the above.

Answer: d

Answer b: The logging-related facilities of `GenericService` are actually an implementation of the `atg.nucleus.logging.ApplicationLogging` interface. If your component extends a non-ATG base class, and you are therefore unable to use `GenericService` as a base, you can still take advantage of ATG logging by implementing this interface yourself.

Summary

In this lesson, you should have learned how to:

- Create component classes that extend ATG's `GenericService` base class for full integration with Nucleus
- Write a component class that takes advantage of the ATG application logging facility

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *API Platform Reference*
  - `atg.nucleus` package
  - `atg.nucleus.logging` package
- *Programming Guide*
  - "Nucleus: Organizing JavaBean Components"
- Sample code included in ATG
  - Many delivered modules include an `src/java` directory.
- For more information about JavaBeans, see http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html.
  - Includes a tutorial, technical specs, and documentation

ORACLE

Practice 1 Overview: Creating Custom Components

This practice covers the following topics:

- Setting up for Java development
  - Editing an ATG module's manifest file
  - Creating an Eclipse project for an existing ATG module
- Creating a custom component
  - Writing a custom Java class
  - Creating a new ATG component

In the *Foundations of ATG Application Development* course, you created a new module called `Dynamusic` that required the provided `Dynamusic-Base` module.

In this course, all of the `Dynamusic` functionality is rolled into a new module called `DynamusicEAP` for you to use. `DynamusicEAP` contains the Java code, components, repositories, and Java EE application with JSPs that you need to get started. As you complete the practices, you will be editing `DynamusicEAP`.

To test your functionality in the web application, you can access Dynamusic through the following URL:

http://localhost:7103/dynamusic/

2 Repository API

Objectives

After completing this lesson, you should be able to use the ATG Repository API to manipulate repository items within Java code, including retrieving, creating, updating, deleting, and querying for items.

Road Map

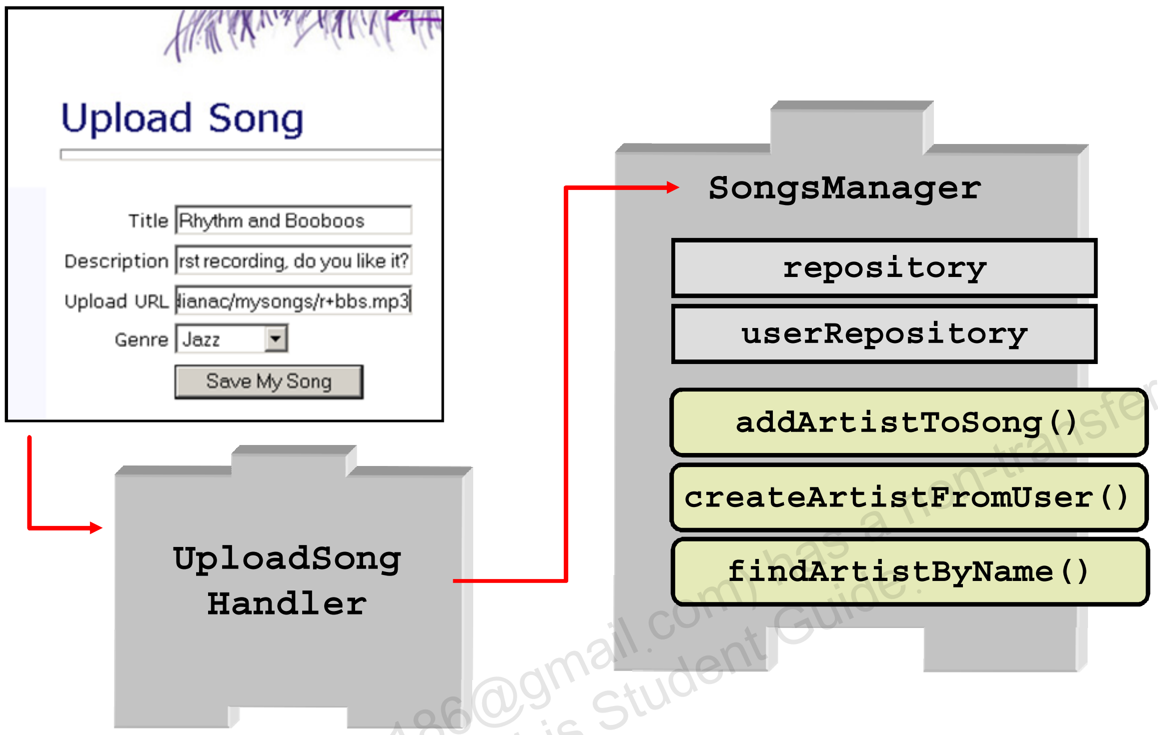
- Repository Overview
- Mutable Repositories

ORACLE

Dynamusic Repository Access

- In previous course examples, Dynamusic repository access has used standard servlet beans and form handlers.
- "Manager" components provide specialized access to repositories that is not supported by the standard components.

Example: `SongsManager`



One of Dynamusic's features is that amateur musicians who are Dynamusic members can upload songs to share with other members.

Dynamusic requires that all songs in the `SongsRepository` be associated with artists. However, when the song is uploaded from a user, there is no artist associated with it (because the user is the artist). Therefore, two things must be done after creating the song item in the repository:

1. A new artist item must be created based on information about the user.
2. The new artist must be set as the value for the song's `artist` property.

This is not something that can be handled by the default `RepositoryFormHandler`, so a custom form handler is needed (`UploadSongHandler`). It is good practice, however, not to do repository access directly from the form handler. Instead, there should be a centralized component that manages all access to the repository: `SongsManager`. `SongsManager` needs to provide methods to accommodate the needs of `UploadSongHandler`, such as `createArtistFromUser()` and `addArtistToSong()`.

The next lesson explores the creation of `UploadSongHandler`, but the foundation is established in this lesson by discussing what `SongsManager` provides.

All the code examples shown in the lessons in the course are available in the `<coursedir>/examples/` folder structure (by lesson number). `<coursedir>` is located on your practice environment in `/ae101/Training/Extending`.

Repository API: Overview

The ATG Repository API is a set of interfaces that are used for generic data access.

Interface	Key Methods
`Repository`	`getItem(),getView()`
`RepositoryItem`	`getPropertyValue()`
`RepositoryView`	`executeQuery()`
`MutableRepository`	`addItem()` `createItem()` `getItemForUpdate()` `updateItem()` `removeItem()`
`MutableRepositoryItem`	`setPropertyValue()`

More information about the interfaces and methods listed in the slide can be found in the *API Platform Reference* in the `atg.repository` package.

- `Repository`: Is used to gain access to a given `RepositoryItem` when the ID is known; is also used to obtain a `RepositoryView`
- `RepositoryItem`: Returns the property values for an individual item from the repository
- `RepositoryView`: Used to execute queries to return an array of `RepositoryItem` objects
- `MutableRepository`: Provides methods used to insert, update, and delete `RepositoryItem` objects
- `MutableRepositoryItem`: Is used to change or set the values of a `RepositoryItem`

The API provides generic data access calls that mask the underlying data store from the developer. The ATG developer can use the same method calls for LDAP and SQL data stores.

Retrieving a Repository Item by ID in `SongsManager`

```
public void addArtistToSong(String pSongid, String
    pArtistid) throws RepositoryException {
    Repository repository = getRepository();
    try {
      RepositoryItem artistItem =
      repository.getItem(pArtistid,"artist");

      if (isLoggingDebug())
          logDebug("adding artist: " +
             artistItem.getPropertyValue("name"));
    }
    catch (RepositoryException e) {
      if (isLoggingError()) {
          logError("Unable to add artist to song", e);
      }
      throw e;
    }
}…
```

`SongsManager.addArtistToSong()` takes the ID of a song and an artist, and then modifies `song.artist` to point to that artist. Because the artist item is not modified, the method can simply use the `getItem` method to get the item associated with the ID passed in. The next few slides explain how to work with an item to be modified.

Note the use of the try-catch block. Most Repository API methods throw a `RepositoryException`. Java requires that you either catch all exceptions or declare your method as throwing the exception. This method does both. It catches the exception so it can do some processing (such as logging a helpful error message or some other error handling), but it also throws the error back up the stack so that any method calling this one knows the exception occurred and handles it accordingly.

Class: `atg.repository.Repository`

`public RepositoryItem` **`getItem`**`(String pId, String pDescriptorName) throws RepositoryException`

Class: `atg.repository.RepositoryItem`

`public Object` **`getPropertyValue`**`(String pPropertyName)`

Road Map

- Repository Overview
- Mutable Repositories

`MutableRepository`

- `Repository` and `RepositoryItem` provide a read-only view of repository elements.
- `MutableRepository` is an extension that enables repository elements to be:
  - Created
  - Updated
  - Deleted
- `MutableRepository` also provides appropriate cache synchronization (either change notifications or locking)
  - Depends on caching mode

A `MutableRepository` offers a read-write view of a repository, keeping performance and transactional integrity in mind.

`MutableRepository` objects are copies of the real objects that can be altered by the application. Because changes are not written back on each change, performance is not affected. On a call to update the repository, a new transaction is started in which its contents are written back to the repository. (Transactions are covered in detail in a later lesson.)

Updating an Existing Item

1. Get the item by using `getItemForUpdate()`.
2. Change values by calling the item's `setPropertyValue()` method.
3. Save changes to the data store by using `updateItem()`.

Using `MutableRepository` in `SongsManager`

```
public void addArtistToSong(String pSongid, String
    pArtistid) throws RepositoryException {

    MutableRepository mutRepos = (MutableRepository)
      getRepository();
    try {
      RepositoryItem artistItem =
          mutRepos.getItem(pArtistid,"artist");

 1    MutableRepositoryItem mutSongItem =
          mutRepos.getItemForUpdate(pSongid,"song");

 2    mutSongItem.setPropertyValue("artist",artistItem);

 3    mutRepos.updateItem(mutSongItem);
}
…
```

ORACLE

The code example in the slide highlights the following actions:

1. Get the item for update.
2. Change the item's properties.
3. Save the changes.

Class: `atg.repository.MutableRepository`

```
public MutableRepositoryItem getItemForUpdate (String pId, String
pDescriptorName) throws RepositoryException
```

```
public void updateItem(MutableRepositoryItem pItem) throws
RepositoryException
```

```
public void removeItem(String pId, String pDescriptorName) throws
RepositoryException
```

```
public MutableRepositoryItem createItem(String pDescriptorName)
throws RepositoryException
```

Class: `atg.repository.MutableRepositoryItem`

```
public void setPropertyValue(String pPropertyName,
Object pPropertyValue)
```

Creating a New Item

1. Create a `MutableRepositoryItem` by using the `MutableRepository.createItem()` method.
2. Set the item’s properties by using the item’s `setPropertyValue(String pPropertyName, Object pPropertyValue)` method.
3. Store the item in the data store by using the `MutableRepository.addItem()` method.

Creating a New Item in `SongsManager`

```
public String createArtistFromUser(String pUserid)
    throws RepositoryException {

    MutableRepository mutRepos = (MutableRepository)
      getRepository();
    RepositoryItem user =
      getUserRepository().getItem(pUserid, "user");
    String username =
      (String)user.getPropertyValue("firstName") + " " +
      user.getPropertyValue("lastName");
    MutableRepositoryItem mutArtistItem =
      mutRepos.createItem("artist");
    mutArtistItem.setPropertyValue("name", username);
    …copy all relevant properties…
    mutRepos.addItem(mutArtistItem);
    return mutArtistItem.getRepositoryId();
}
```

`SongsManager.createArtistFromUser()` takes a user ID, creates a new artist based on information about that user, and then returns the ID of the newly created artist.

Although not shown in the code example in the slide, this method can perform exception catching and error logging (as in the example of the `addArtistToSong` method).

Deleting an Item in `SongsManager`

```
public void deleteSong(String pSongId) throws
    RepositoryException {

  MutableRepository mutRepos =
     (MutableRepository) getRepository();

  mutRepos.removeItem(pSongId, "song");

}
```

When you call `removeItem()` on an object, it also removes the object from Java memory, no longer allowing access to the object's properties after its deletion.

Querying the Repository

- The most straightforward way of executing queries in a repository is to use Repository Query Language (RQL) in the `RqlStatement` class.
- You can create an `RqlStatement` from a string by using `parseRqlStatement()`.

```
RqlStatement statement =
     RqlStatement.parseRqlStatement(rql string);
```

- RQL statements are executed against a `RepositoryView`.

The full name of the class is `atg.repository.rql.RqlStatement`.

Note: You cannot execute RQL from a String directly. First you must convert the String to an object of type `RqlStatement` by using the static method `parseRqlStatement`.

RQL statements are executed against a `RepositoryView`. A `RepositoryView` is an object that allows queries against a specific item type. When you create the `RepositoryView` object, the item type is specified, and only items of that type are returned.

Parameterized RQL

- The syntax for passing parameters to an RQL query is different for the API than for the `RQLQueryForEach` droplet.
  - `?n` (where "*n*" is the parameter number)
- Parameters are passed in an array of objects.
  - "*n*" is the index in the array.
- Example: "Have any artists with this name already been created?"

```
public String createArtistFromUser(String pUserid) … {
 …
 String username = (String)user.getPropertyValue("firstName") + " " +
   user.getPropertyValue("lastName");
 RepositoryItem artistItem = findArtistByName(username);
 if (artistItem==null) {
   …
```

Executing RQL Queries in `SongsManager`

```
public RepositoryItem findArtistByName(String
    pArtistName) throws RepositoryException {
    Repository repos = getRepository();
    RqlStatement findArtistRQL;
1   RepositoryView artistView = repos.getView("artist");
2   Object rqlparams[] = new Object[1];
    rqlparams[0] = pArtistName;
    findArtistRQL =
3     RqlStatement.parseRqlStatement("name = ?0");
    RepositoryItem [] artistList =
      findArtistRQL.executeQuery (artistView,rqlparams);

    if (artistList != null) {
      return artistList[0];
    }
    return null;
}
```

If a single user uploads multiple songs, you should not create multiple artists. Instead, you use the artist that has already been created. Therefore, `createArtistFromUser()` first checks for an existing artist with the matching name.

Note: Realistically, this should probably best be handled by adding an `artistid` property to the user profile and storing the artist ID associated with each user. But then you would not be able to see the RQL query in action.

The code example in the slide shows three steps:

1. Get a repository view, which is a context in which to execute queries. A view limits the query to returning the item type specified when constructing the view (in this case, `artist`).
2. Create an array of parameters to use in the query. The RQL statement refers to each parameter's index position, which allows you to combine more than one parameter in the query. In the example in the slide, there is only one parameter called `pArtistName`, the name of the artist to search for. Its index position is 0.
3. Execute the query, substituting the parameters as needed.

RQL statements can also be executed by using the Component Browser (in the ATG Admin UI), which has a text box for XML statements (browse to the relevant `GSARepository` component). This is useful for testing.

Named Queries

- You can also define a query in the SQL repository definition file.
- In this example from `songs.xml`, a named query returns all the songs that belong to a particular genre:

```
…
<item-descriptor name="song">
  …
  <named-query>
    <rql-query>
     <query-name>genreSongs</query-name>
     <rql>genre = ?0</rql>
    </rql-query>
  </named-query>
</item-descriptor>
…
```

Note: Instead of using XML, you can also create a named query programmatically by using the Query Builder API. This is out of the scope of this course.

Another feature of named queries is the ability to define direct SQL queries, including calling stored procedures. For more information about named queries, see the *ATG Repository Guide*.

Executing Named Queries

There are two ways to execute a named query:

- Programmatically: Using the `NamedQuery` object
- From a JSP: Using the `NamedQueryForEach` droplet

Using a Named Query in `SongsManager`

```
public RepositoryItem[] getJazzSongs() {
  …
    RepositoryView songView =
      getRepository().getView("song");
    NamedQueryView nView = (NamedQueryView)songView;

    Query namedQuery =
      nView.getNamedQuery("genreSongs");

    Object rqlparams[] = new Object[1];
    rqlparams[0] = "jazz";

    ParameterSupportView pview =
      (ParameterSupportView)nView;
    return pview.executeQuery(namedQuery, rqlparams);
  …
}
```

Take great care with casting a `RepositoryView` item to a `NamedQueryView` (and later a `ParameterSupportView`). Named queries and parameterized queries are not supported by all implementations of the Repository Interface. Those implementations that do support them will return a “view” that implements `NamedQueryView` and `ParameterSupportView`, respectively. For those implementations that do not support named queries and parameterized queries, if you attempt to cast them as shown in the slide, a Class Cast Exception results. Unless you know that a repository implementation supports named queries, you should check to make sure before trying to cast. The most common implementation of Repositories is SQL, which does support parameterized, named queries.

Using a Named Query in `popsongs.jsp`

```
<dsp:droplet
   name="/atg/dynamo/droplet/NamedQueryForEach">
   <dsp:param name="repository"
     bean="/dynamusic/SongsRepository" />
   <dsp:param name="queryName" value="genreSongs" />
   <dsp:param name="itemDescriptor" value="song" />
   <dsp:param name="inputParams" value="pop" />
...
   <dsp:oparam name="output">
     <li>
      <dsp:a href="song.jsp">
         <dsp:param name="itemId" param="element.id"/>
         <dsp:valueof param="element.title"/>
     </dsp:a>
   </dsp:oparam>
  …
</dsp:droplet>
```

ORACLE

The `NamedQueryForEach` droplet works just like the `ForEach` droplet, with the additional ability to specify the repository, item type, and query name that you want to invoke.

Quiz

Which of the following statements describe the repository API? (Select all that apply.)

a. You can execute a named query both programmatically and from a JSP.

b. You can execute RQL queries in your Java code but not in a JSP.

c. `MutableRepository` enables repository elements to be created, updated, and deleted.

d. It is a best practice to use a centralized component that manages all access to the repository rather than to access the repository directly from a form handler.

Answer: a, c, d

Answer b: You can execute RQL queries in your Java code, or use an `RQLQueryForEach` droplet to execute the query in your JSP.

Summary

In this lesson, you should have learned how to:

- Use the Repository API to retrieve, create, modify, delete, and query for items in a repository
- Define and use a named query

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Repository Guide*
  - “Repository API”
  - “Repository Queries”
  - “SQL Repository Queries”
- *API Platform Reference*
  - `atg.repository` package

Practice 2 Overview: Using the Repository API

This practice covers the following topics:

- Implementing a `deleteAlbumsByArtist` method by using Repository API calls
- Testing by using a provided JSP, passing in the ID of an artist

3

Custom Form Handlers

ORACLE

Objectives

After completing this lesson, you should be able to create custom form handlers by:

- Extending `RepositoryFormHandler` and adding application-specific functionality
- Extending a generic ATG base form handler class

Form Handlers

- Form handlers are components that handle input from an HTML form.
- A form handler may need to:
  - Handle data in a variety of formats
  - Check input for validity
  - Handle errors
  - Supply data to manager components
  - Exercise methods in manager components that persist data or perform a function
  - Pass input to a servlet for processing
  - Redirect to other pages based on form input

ORACLE

Custom Form Handlers

- ATG comes with several form handlers, such as:
  - `RepositoryFormHandler`
  - `ProfileFormHandler`
- Applications often require more complex form handling.
- Custom forms can:
  - Extend form handlers with existing functionality
  - Extend a generic form handler

Road Map

- Extending `RepositoryFormHandler`
- Creating custom form handlers based on ATG's `GenericFormHandler`
- Best practices

Extending `RepositoryFormHandler`

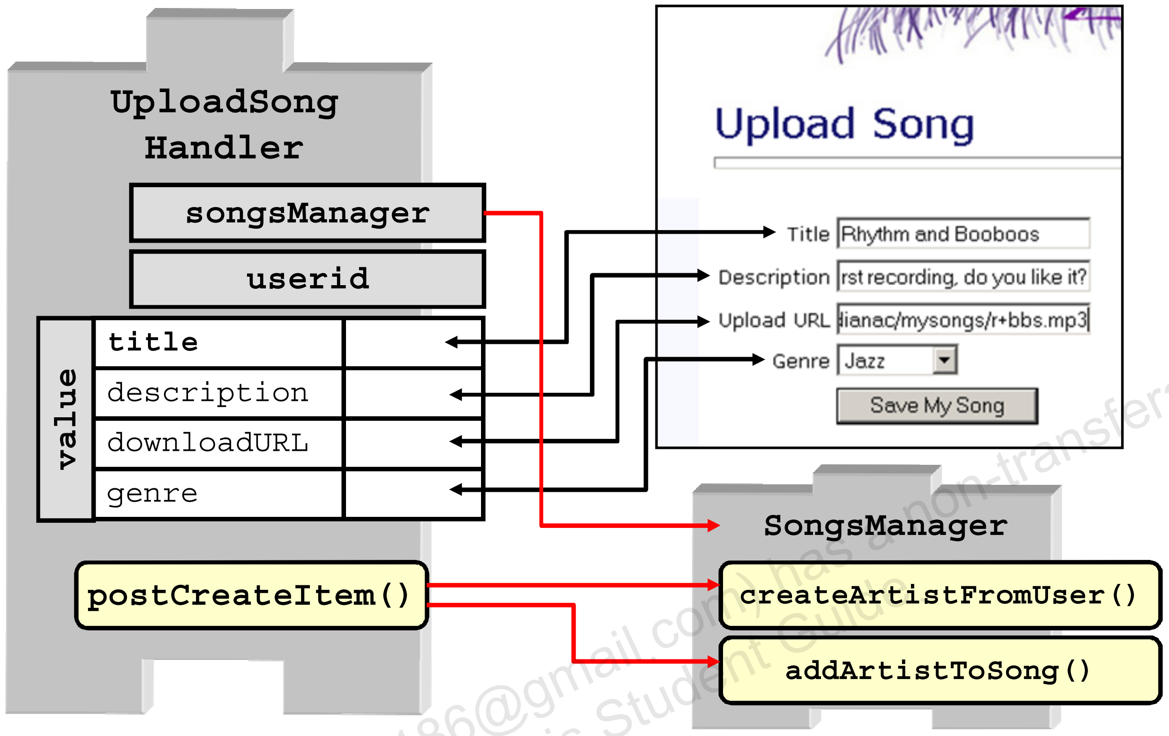
- `RepositoryFormHandler` is a powerful class.
  - It includes logic that keeps the Java code from being dependent on a specific schema or repository definition.
  - Do not override out-of-the-box methods to create, update, or delete items.
- Designed to be extended, `RepositoryFormHandler` includes a method "hook" that can be overridden by subclasses:
  - `pre/postCreateItem`
  - `pre/postDeleteItem`
  - `pre/postUpdateItem`

Class: `atg.repository.servlet.RepositoryFormHandler`

```
protected void preCreateItem(DynamoHttpServletRequest pRequest,
DynamoHttpServletResponse pResponse)
protected void postCreateItem(DynamoHttpServletRequest pRequest,
DynamoHttpServletResponse pResponse)
protected void preDeleteItem(DynamoHttpServletRequest pRequest,
DynamoHttpServletResponse pResponse)
protected void postDeleteItem(DynamoHttpServletRequest pRequest,
DynamoHttpServletResponse pResponse)
protected void preUpdateItem(DynamoHttpServletRequest pRequest,
DynamoHttpServletResponse pResponse)
protected void postUpdateItem(DynamoHttpServletRequest pRequest,
DynamoHttpServletResponse pResponse)
```

Note: `DynamoHttpServletRequest` and `DynamoHttpServletResponse` are in the `atg.servlet` package.

Example: `UploadSongHandler`



A feature of Dynamusic is that amateur musicians who are Dynamusic members can upload songs to share with other members.

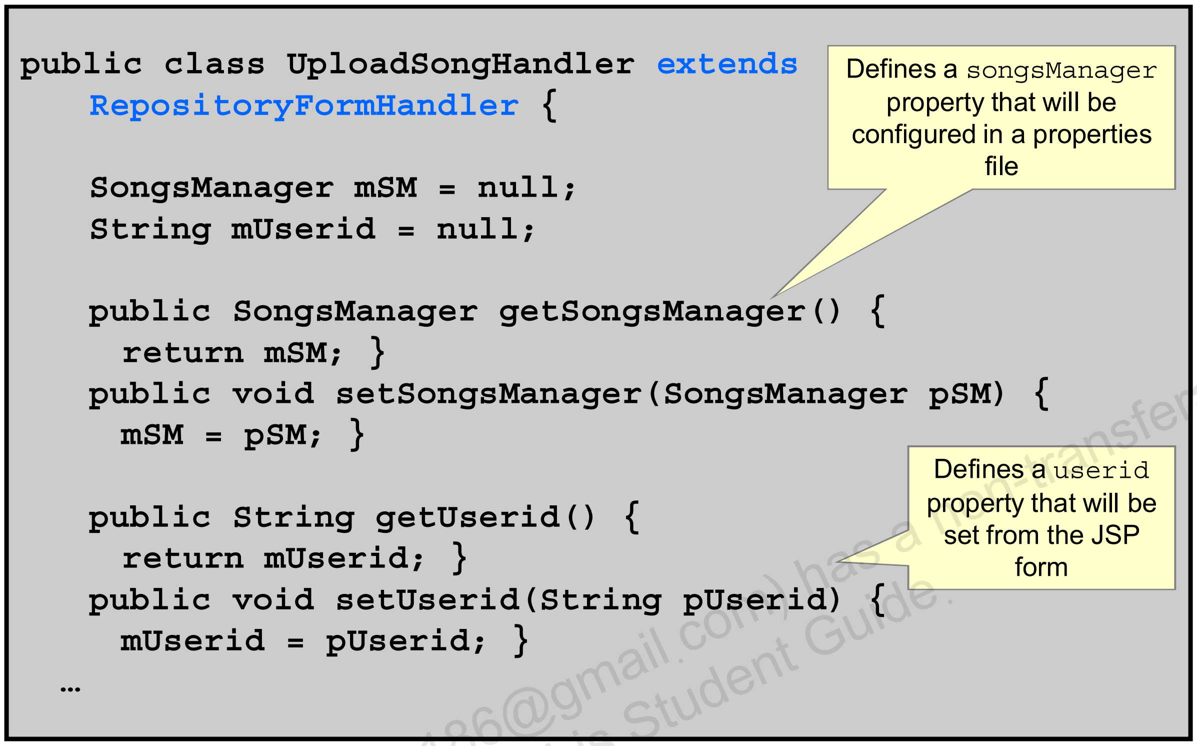
In Dynamusic, all songs in `SongsRepository` must be associated with artists. But when the song is uploaded from a user, there is no artist associated with it (because the user is the artist). Therefore, after a member creates the song item in the repository, two actions must be performed:

1. A new artist item needs to be created based on information about the user.
2. That new artist must be set as the value for the song's artist property. This is not something that can be handled by the default `RepositoryFormHandler`, so a custom form handler is needed.

In the previous lesson, you saw the development of some supporting methods in the `SongsManager`. In this lesson, you see how `UploadSongHandler` uses those methods.

Because `UploadSongHandler` extends `RepositoryFormHandler`, it does not need to create properties to hold the song values (that is done automatically in a map, as seen previously), and it does not need to handle creation of the item in the repository. But *after* the repository item has been created, the `postCreateItem` method is called, which creates a new artist for the current user (if necessary) and links the newly created song to the artist.

Extending `RepositoryFormHandler` in `UploadSongHandler`



```
public class UploadSongHandler extends
    RepositoryFormHandler {

    SongsManager mSM = null;
    String mUserid = null;

    public SongsManager getSongsManager() {
      return mSM; }
    public void setSongsManager(SongsManager pSM) {
      mSM = pSM; }

    public String getUserid() {
      return mUserid; }
    public void setUserid(String pUserid) {
      mUserid = pUserid; }
  …
```

Defines a songsManager property that will be configured in a properties file
Defines a userid property that will be set from the JSP form

All the sample code in this lesson is available in `<coursedir>/examples/lesson03/UploadSongHandler.java` and `uploadsong.jsp`.

The following code is not shown but is present in the code example:

```
import atg.droplet.*;
import atg.service.email.*;
import atg.repository.*;
import atg.servlet.*;
```

Extending `postCreateItem()` in `UploadSongHandler`

```
protected void postCreateItem (DynamoHttpServletRequest
    pRequest, DynamoHttpServletResponse pResponse)throws
    javax.servlet.ServletException,java.io.IOException {

    SongsManager sm = getSongsManager();
    String userid = getUserid();
    String artistid = null;

    String songid =
    getRepositoryItem().getRepositoryId();

    try {
      artistid = sm.createArtistFromUser(userid);
      sm.addArtistToSong(songid,artistid);
    }
    catch (RepositoryException re) { … }
  }
```

In the example in the slide, the out-of-the-box `createItem` method has already been run to create the new song repository item. That item is accessible through the class's `getRepositoryItem` method. The `postCreateItem` method shown in the slide gets `SongsManager`, the user's ID, and the song's ID. It then creates the artist from the user ID and adds the artist to the song.

Using `UploadSongHandler` in `uploadSong.jsp`

```
<dsp:input
   bean="/dynamusic/UploadSongHandler.value.title"
   name="title" type="text"/>
…

<dsp:input bean="/dynamusic/UploadSongHandler.userid"
   type="hidden" beanvalue="Profile.repositoryId"/>

<dsp:input
   bean="/dynamusic/UploadSongHandler.createSuccessURL"
   type="hidden" value="success.jsp"/>

<dsp:input bean="/dynamusic/UploadSongHandler.create"
   type="Submit" value="Save My Song"/>
…
```

Most of `uploadsong.jsp` looks like the uses of `RepositoryFormHandler` that you are accustomed to, but one value is different: `userid`. This property needs to be set—not by the user explicitly but by the page based on the user's ID. This is used to create an artist item based on the user.

Note: Another way to handle this situation is to include the Profile in `UploadSongHandler.java` so that it does not need to be passed in as a value.

Quiz

When extending the `RepositoryFormHandler` class, you must override the out-of-the-box methods to create, delete, or update the items.

a. True

b. False

Answer: b

`RepositoryFormHandler` includes logic that keeps the Java code from being dependent on a specific schema or repository definition. Therefore, you should not override out-of-the-box methods to create, update, or delete items. Instead, you can overwrite the related pre- or post-methods, such as `preCreateItem` or `postDeleteItem`.

Road Map

- Extending `RepositoryFormHandler`
- Creating custom form handlers based on ATG's `GenericFormHandler`
- Best practices

Creating Custom Form Handlers

- Custom form handlers should typically extend `GenericFormHandler`, which:
  - Is a subclass of `GenericService`
  - Includes standard form error handling
- Custom form handlers supply their own
  - `get` and `set` methods to implement properties
  - `handle` methods to process property values

The full class name is `atg.droplet.GenericFormHandler`.

You can build a base class that implements the `DropletFormException` interface, or you can extend either `EmptyFormHandler` or `GenericFormHandler`. This course focuses on `GenericFormHandler`, but the principles apply to whichever approach you choose.

Note

`EmptyFormHandler` has no error processing other than a `handleFormException` method that you can override. You use this base class when you want to handle errors differently than `GenericFormHandler` does. Note that none of the provided form handlers actually do error checking. Instead, they just catch and handle whatever exceptions are thrown.

Another option is to extend `TransactionalFormHandler`, which extends `GenericFormHandler`. `TransactionalFormHandler` adds transaction management capabilities while processing form input. You learn more about transactions in a later lesson. For more information about `TransactionalFormHandler`, refer to the “Form Handlers and Handler Methods” section of the “Working with Forms and Form Handlers” chapter of the *ATG Programming Guide*, or see the *API Platform Reference*.

Property Handler Methods

Property handling is performed by `handleX` methods:

- In addition to `setX` and `getX` methods, an ATG JavaBean property can have a `handleX` method.
- `handleX` is called immediately after `setX`.

Form handling is usually performed as follows:

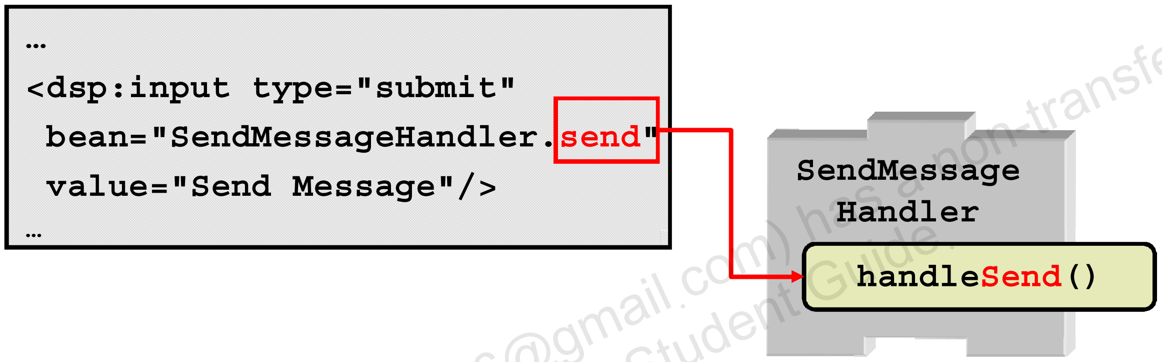
1. The user enters data on a form.
2. The user presses a Submit button.
3. `handleX` methods for individual properties are called. (The order is subject to priority, as you will soon see.)
4. The `handleX` method for the Submit button's property is called.

You can have a property that has *only* a `handleX` method. Submit buttons are usually linked to these types of properties.

Note: The `handleX` methods are a feature of ATG.

Property Handler Methods

- `handleX` methods can be associated with any component property linked to the form.
- A `handleX` method is usually associated with a property linked to the Submit button.
- Example:

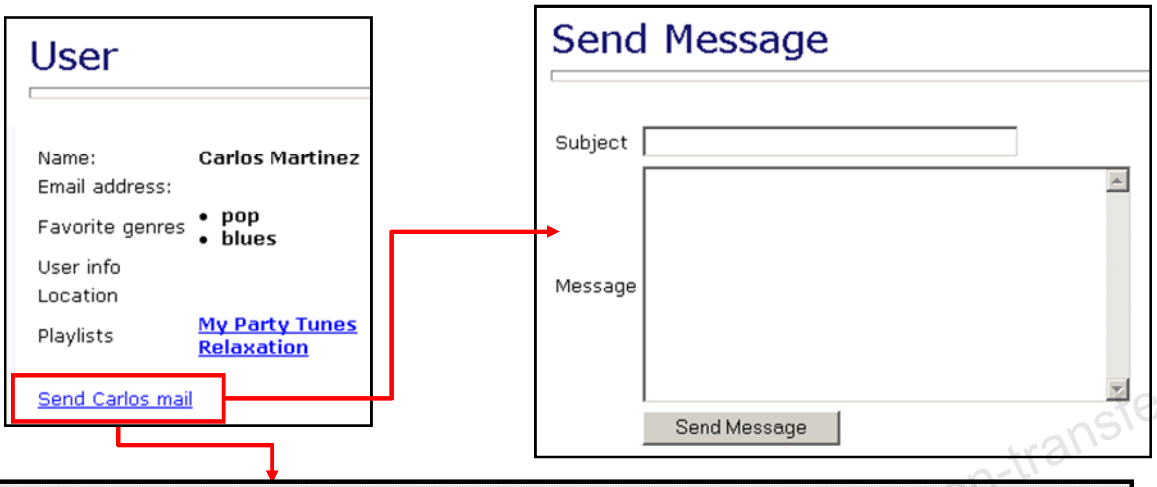


```
…
<dsp:input type="submit"
 bean="SendMessageHandler.send"
 value="Send Message"/>
…
```

Note that the `send` property is associated with the `handleSend` method. This is true for all submit properties. For instance, the `update` property of `RepositoryFormHandler` is associated with the `handleUpdate` method.

The `SendMessageFormHandler` example continues in the next few slides.

`SendMessageHandler` Example



```
…
<dsp:a href="sendmessage.jsp">
 <dsp:param name="userid" param="itemId"/>
 Send <dsp:valueof param="element.firstName"/> mail
</dsp:a>
…
```

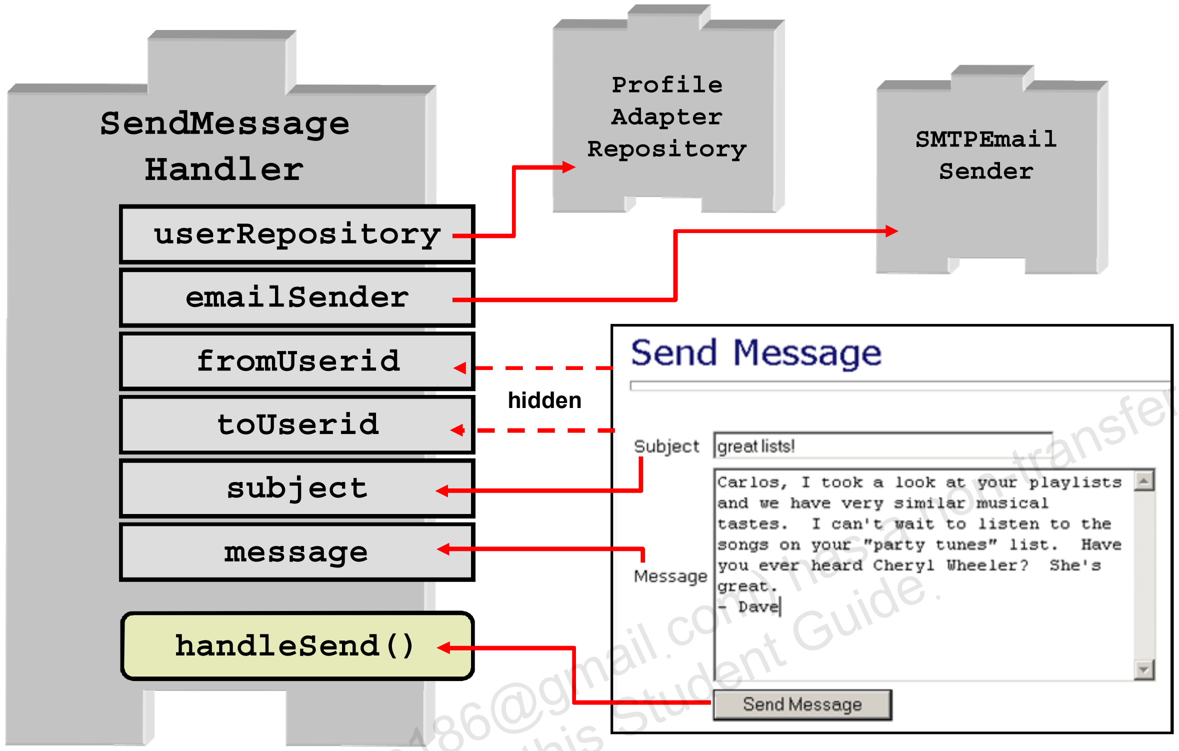
Although ATG provides `EmailFormHandler`, Dynamusic might need more functionality than what that form handler provides.

Examples

- To maintain anonymity, Dynamusic forwards the email without attaching the sender's email address, constructing a specially coded "from" address.
- The receiving user might have opted out of sharing the profile or receiving email from other members.
- Dynamusic might want to do some processing on the contents of the email to check for disallowed content.

In the slide, you see how to get from looking at the profile page of a sample user (Carlos) to seeing the page with the form to send that user a message. The user's profile page has a link called "Send Carlos mail." This is simply a link to another JSP that passes Carlos's user ID and his first name as parameters. `sendmessage.jsp` has two input fields for the message's subject and the message itself. The Submit button is called Send Message.

`SendMessageHandler` Example: Behind the Scenes



Carlos, I took a look at your playlists and we have very similar musical tastes. I can't wait to listen to the songs on your "party tunes" list. Have you ever heard Cheryl Wheeler? She's great.
- Dave

The example in the slide shows how the different components work together to create the message to send:

- `SendMessageHandler` gathers information for the message from other components and the JSP form. It needs to know the name of the user repository (`ProfileAdapterRepository`) and the component that handles the sending of email (`SMTPEmailSender`). It gathers the user IDs of the sender and the receiver of the message from the JSP form as hidden input values, which do not appear in the JSP form and are not editable by the sender.
- The `subject` and `message` properties of `SendMessageHandler` are linked to the appropriate text input fields in the JSP form.
- The Send Message button is the Submit button for the form and is linked to the `handleSend` method of `SendMessageHandler`.

Extending `GenericFormHandler` in `SongMessageHandler`

```
import atg.droplet.*;
import atg.service.email.*;
import atg.repository.*;
import atg.servlet.*;

public class SendMessageHandler extends GenericFormHandler
  {
 … member variables…
 … get/set methods for properties …
 public boolean handleSend(
      DynamoHttpServletRequest pRequest,
      DynamoHttpServletResponse pResponse)
          throws java.io.IOException,
          javax.servlet.ServletException {
 …process and send mail…
 return true;
}
}
```

A handler method accepts a request object and a response object as arguments. The method can get parameter inputs from the request object and redirect to another page by using the response object.

`DynamoHttpServletRequest` and `DynamoHttpServletResponse` are ATG extensions for standard `HTTPRequest` and `HTTPResponse` objects.

A handler method can throw an `IOException` or `ServletException`. Nucleus catches these exceptions.

Handler methods return a Boolean value:

- **True:** ATG continues processing and serving the page.
- **False:** ATG stops processing the page. This is used with redirection, which is covered later in the lesson.

This example code is in `<coursedir>/examples/lesson03/SendMessageHandler.java`, as is the corresponding page, `sendmessage.jsp`.

Form Errors

- Subclasses of `GenericFormHandler` have access to ATG's form error processing.
- Exceptions are saved as properties of the form handler:
  - `formError`
  - `formExceptions`
  - `propertyExceptions`
- Exceptions are handled in the page by calling the `ErrorMessageForEach` servlet bean.

Properties of the form handler for error handling are:

- **`boolean formError`:** Is set to true if any errors occurred when the form was processed
- **`Vector formExceptions`:** Holds the exceptions that occurred when the form was processed
- **`Dictionary propertyExceptions`:** Contains an entry for each property name set by the form. If an exception occurred for that property, the entry value contains the exception.

See the *API Platform Reference* for more information about `GenericFormHandler` methods.

Adding Form Errors

- Form error handling is a two-step process:
  1. Check for errors.
  2. Add exceptions to the `formExceptions` Vector.
- Rather than use a standard Java exception, it is generally more useful to provide a more descriptive custom error.
- To add your own custom errors, use `addFormException()`, passing in a new `DropletException` object.

One way to do form input validation is to use "converter" attributes in the `dsp:input` tags. If you do this and the input does not conform to the expected value, the converter may throw a `DropletException`.

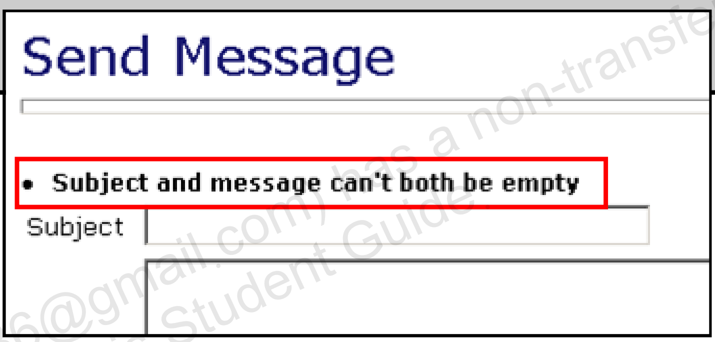
If the page developer uses the `required` converter (for example, `<dsp:input type="text" bean="profile.firstName" required="true"/>`), and if the user does not supply a value, the converter throws a `DropletFormException`, which is then added to the `formExceptions` attribute.

You can also create custom converters to do form validation and generate exceptions for non-conforming input.

Note: `DropletFormException` extends `DropletException`.

Adding a Form Error in `SongMessageHandler`

```
public boolean handleSend( … )… {
  …
    if (subject.equals("") && message.equals("")) {
      if (isLoggingDebug())
          logDebug("subject and message both null");

      addFormException(new DropletException
          ("Subject and message can't both be empty"));
      return true;
    }
  …
}
```


Send Message

• Subject and message can't both be empty

The example in the slide shows how to add a `DropletException` to the form. If both the subject and message fields are empty, the form exception “Subject and message can’t both be empty” is added.

Note: This is an example where you might want to use Java resource bundles to internationalize your error messages. For more information about resource bundles and internationalization, refer to the “Internationalizing an ATG Web Site” chapter of the *ATG Programming Guide*.

Providing Success and Error Redirection in `SongMessageHandler`

```
public boolean handleSend( … )… {
   …
    if (subject.equals("") && message.equals("")) {
      addFormException(new DropletException
           ("Subject and message can't both be empty"));
      if (getErrorURL() != null) {
           pResponse.sendLocalRedirect
               (getErrorURL(), pRequest);
           return false;
      } //end if errorURL is not null
      return true;
    } //end if subject & message are empty
    …finish processing form…
    if (getSuccessURL() != null) {
      pResponse.sendLocalRedirect
           (getSuccessURL(), pRequest);
    return false;
}
…
return true;
```

The response object that is supplied as a parameter to the handler method is of type `atg.servlet.DynamoHttpServletResponse`. That class has two methods relevant to the code example in the slide:

- **`sendRedirect(String url)`:** This is a standard Java method that should be used when directing to a page outside your ATG application.
- **`sendLocalRedirect(String url, DynamoHttpServletRequest)`:** This method is specific to ATG and should be used when redirecting to a page within your application. It handles relative page references within the web application and includes the session ID to maintain session tracking.

Whenever your code redirects to another page, be sure to return false for the handler method. This tells ATG's form processing not to continue processing and delivering the form, but instead to move on to handling a new page.

Form Handler Scope

- If the component is request-scoped, the following actions occur each time a user accesses a page:
  - A new instance of the form handler is created.
  - The new instance is destroyed after that single request.
- If the component is session-scoped, the following actions occur:
  - A new instance is created the first time a user accesses the form.
  - The instance uses the configured values (the first time) or the live values from the session (subsequent times).

Using a request-scoped form handler component is a good way to reset a form.

If you want property values to persist from page to page, you can set the form handler to be session-scoped. For example, suppose that your form is very long and you want to split it across two pages. The user fills in the first page and then clicks a button to go to the second page, but this button does not call the final submit handler. On the final page of the form, the final submit handler is called and then the values are processed.

Order and Priority

- Setting and handling values in a specific order may be important.
- Input fields can have a priority attribute that determines handling order:
  - Lower number = lower priority
- The syntax for defining priority is:

```
<dsp:input type="text" priority="10"/>
```

- By default, all form elements are set to `priority="0"` except submit buttons, which have `priority="-10"`.

Road Map

- Extending `RepositoryFormHandler`
- Creating custom form handlers based on ATG's `GenericFormHandler`
- Best practices

Form Handler Best Practices

Good form handlers should:

- Provide properties for a success URL and error URL
  - Page designers set URL values by using hidden fields in the form.
- Validate the form data
  - If validation is successful, redirect to the location specified by the success URL property.
  - If validation is unsuccessful, redirect to the error URL.
- *Contain no business logic*
  - The business logic should always be done in manager or tools classes.

The dynamic page that is the form is also most likely the page that is used to display any form errors to the user.

Form handlers should contain no business logic. Even the validation of form field values should be externalized in either a manager component or a validator component as appropriate to the application.

Forms and handlers that are written following the best practices in the slide help to ensure that the page designer can flexibly use the form without the need to modify any Java code. This is also the design pattern used for ATG’s built-in form handlers.

Additional Form Handler Conventions

- Every form should have an associated form handler.
- Each custom form should have its own form handler.
- Form handlers should:
  - Be either request-scoped or session-scoped (with preference given to request scope).
  - Reference only those components that are scoped at the same (or more general) scoping as the form handlers themselves
    - Examples: session > session or session > global
  - Never reference properties or methods in other form handlers
  - Extend an existing form handler, whenever possible, making use of pre- and post-handler methods and reusing the component

Every form should have an associated form handler. This form handler could be one of the default form handlers (such as `RepositoryFormHandler` or `ProfileFormHandler`) or a custom form handler.

There should be a one-to-one relationship between a custom form and a custom form handler. You should not use the same custom form handlers across multiple custom forms. You can, however, have multiple forms in a single JSP.

Form handlers should be either request-scoped or session-scoped with preference given to request scope. In general, if the form is used very frequently, it can be session-scoped to avoid instantiating new copies. However, when session scoping is used, tighter maintenance of instance variables must be used and is the exception.

Form handlers should never reference properties or methods in other form handlers. If multiple form handlers share some information, that information is generally persisted for the session and should be managed out of a common, shared manager component.

Whenever possible, extend an existing form handler class and then reuse the existing form handler component with your new class. For example, if you extend the profile form handler class, you can still use `/atg/userprofiling/ProfileFormHandler`. You update the component properties to use your new class.

Quiz

Which of the following statements best describe custom form handlers? (Select all that apply.)

a. A good form handler should provide properties for a success URL and error URL, and validate the form data.

b. The `handleX` method is called before the `setX` method.

c. When extending `GenericFormHandler`, the standard form error handling is not included.

d. You can display error messages generated by the form by calling the `ErrorMessageForEach` droplet in your JSP.

Answer: a, d

Answer b: The `handleX` method is called after the `setX` method.

Answer c: `GenericFormHandler` extends `GenericService`, which includes standard form error handling.

Summary

In this lesson, you should have learned how to create:

- A form handler that extends the functionality of the `RepositoryFormHandler`
- A new form handler by using the `GenericFormHandler` base class

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Page Developer's Guide*
  - "Forms"
  - "Form Handlers"
- *ATG Programming Guide*
  - "Working with Forms and Form Handlers"
- *API Platform Reference*

Practice 3 Overview: Extending Form Handlers

This practice covers the following topics:

- Creating a new `ArtistFormHandler` (based on the `RepositoryFormHandler`) that deletes all albums by an artist before deleting the artist item
- Creating a new form handler (based on `GenericFormHandler`) that implements a weekly trivia contest, testing whether users enter the correct value

4 Managing Transactions

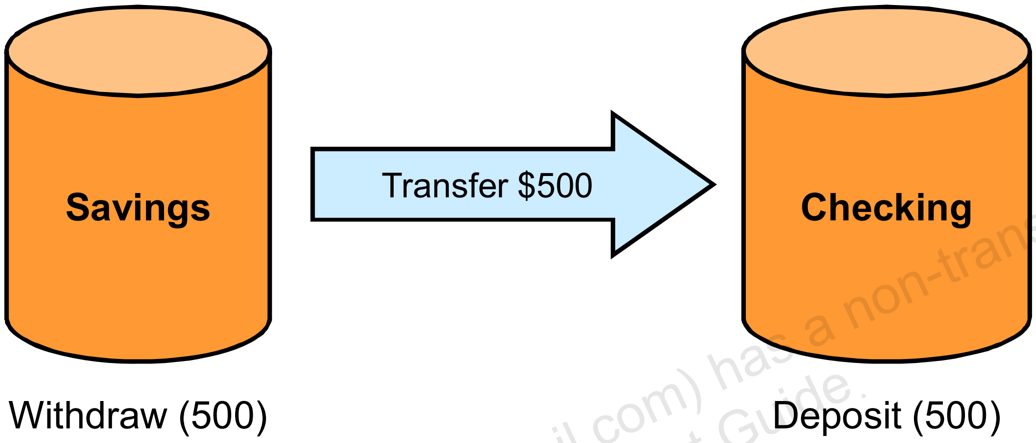
Objectives

After completing this lesson, you should be able to:

- Describe how ATG leverages the Java Transaction API to manage transactions
- Know the techniques that are available for controlling transactions within ATG
- Control transaction boundaries programmatically

Transactions: Overview

The classic example of a transaction is the transfer of money between two bank accounts.



The bank transfer involves two operations: a debit and a credit. These two operations must act as one atomic operation. To maintain the integrity of the system, they must either both succeed or both fail.

Transaction Manager

A Transaction Manager provides services to support the management of transactional resources.

- One transaction may span multiple resources (for example, Oracle and MySQL Server JDBC connections).
- Related processes may involve multiple transactions or may need to share the same transaction.

The Transaction Manager is a service provided by the application server (such as WebLogic Server or JBoss). The service implements the `javax.transaction.TransactionManager` Java interface. The detailed operation of the Transaction Manager is normally hidden from the developer.

Using a Transaction Manager can simplify programming because each method that requires a transactional resource can abstract from all other parts of the system.

The programmer needs to be concerned only about setting up the transactional boundaries within the code. The decision about whether to create a new transaction, suspend a transaction, or join the current transaction is the job of the Transaction Manager.

The Transaction Manager makes the final decision on when to commit or roll back a transaction.

Java Transaction API (JTA)

- JTA is a set of interfaces that standardize transaction processing.
- ATG has full support for JTA.

Interface	Purpose
`UserTransaction`	A component that provides methods used by Java developers to control transactions (for example, begin, commit, and rollback)
`TransactionManager`	A service provided by the application server
`Transaction`	An object that keeps track of the resources involved in a single transaction

JTA stands for "Java Transaction API," which is part of the Java extension package (`javax`). Java EE applications servers must provide Transaction Management services. JTA defines how application servers must provide these services.

Note: The Java EE specification does not require that the Transaction Manager be exposed to applications running in the application server. Therefore, your Java EE applications should use declarative transaction demarcation (covered shortly) or should get a reference to the `UserTransaction` component using JNDI (see the JTA documentation from Oracle for details).

ATG exposes `TransactionManager` and `UserTransaction` as Nucleus components. These components can be difficult to use directly. Fortunately, ATG provides the `TransactionDemarcation` class that enables you to "demarcate" program areas so that you can focus on your task and not be overly concerned about the transactional state of other related processes.

Transaction Completion

- All transactions end in either a commit or a rollback.
- JTA assumes that multiple resources may be involved in a transaction.
  - If a single resource is enlisted, the transaction object simply calls commit or roll back on the single resource.
  - If multiple resources have been enlisted, the transaction object uses a two-phase commit protocol to end the transaction.

ORACLE

Two-phase commit is an important feature of JTA and is discussed in the next slide.
Consider the bank transfer example in which savings and checking accounts are managed by two separate databases. If a withdrawal from the savings account results in the account being overdrawn, that database fails during the commit process. However, the checking account most likely would not fail. Two-phase commit enables the database to detect the failure before the commit stage.

Transaction Completion: Two-Phase Commit

The two-phase commit protocol consists of a prepare phase and a commit phase.

- Phase 1: Prepare
  - Each resource votes to commit or roll back.
  - If any resource votes to roll back, all resources are instructed to roll back.
- Phase 2: Commit
  - If all resources vote to commit, the transaction object issues a commit against each resource.

Consider the bank transfer example as a candidate for two-phase commit. Suppose that you are transferring money from a savings account stored in Oracle Database to a checking account stored in MySQL. You want to ensure that both operations take place. With two-phase commit, before actually committing, the transaction asks each resource (Oracle Database and MySQL) if it is able to complete the operation (PREPARE). If both resources vote “yes,” the transaction asks both to commit and the money is then transferred. If either resource votes “no,” the transfer does not occur.

Although the JTA requires that an application server support the two-phase commit protocol, many databases and JDBC drivers may not support this protocol. ATG provides `atg.service.jdbc.FakeXADataSource`, a class that allows JDBC 1.x drivers to be used with ATG. If your database vendor provides a JDBC driver that is JTA compliant (that is, it implements `javax.sql.datasource`), your `JTDatasource` can reference that component instead.

You can see examples of both types of data source components in your practice environment. DynamusicEAP connects to an Oracle database using `JTDatasource`. The DynamusicEAPTest module that you use when testing repository changes with the `startSQLRepository` utility is configured to use `FakeXADataSource`. You can find both components in the `/atg/dynamo/service/jdbc/` Nucleus namespace.

ATG Transaction Support

- ATG exposes JTA services as Nucleus components.
  - `/atg/dynamo/transaction/UserTransaction`
  - `/atg/dynamo/transaction/TransactionManager`
- ATG also provides `TransactionDemarcation`.
  - A helper class that is used to control transaction boundaries

`TransactionManager` and `UserTransaction` are both interfaces that are defined in the `javax.transaction` package. For more information about the `UserTransaction` and `TransactionManager` interfaces, go to http://www.oracle.com/technetwork/java/javaee/jta/index.html.

The `UserTransaction` interface defines a subset of the methods that must be implemented by `TransactionManager`. The `TransactionManager` interface defines methods for suspending and resuming transactions.

Java EE does not require that application servers expose `TransactionManager` to the programmer. When coding in a "pure Java EE" model, you do not have programmatic access to the `suspend` or `resume` methods.

Even within an ATG application, programmatic use of the `suspend` and `resume` methods is discouraged. ATG provides a helper class, `atg.dtm.TransactionDemarcation`, to assist with programming transaction boundaries. `TransactionDemarcation` is discussed later in this lesson.

`TransactionDemarcation`

- Is a helper class that simplifies the management of transaction boundaries
- Exposes the power of JTA transaction boundaries to the ATG programmer
- Has the following key methods:
  - `begin()`
  - `end()`

`TransactionDemarcation` is in the `atg.dtm` package. The key methods are:

- **`begin` method:** Is used to indicate the involvement of this process in the current transaction. Depending on arguments to this method, `TransactionManager` may decide to start a new transaction, have this process join in an existing transaction, or produce an error.
- **`end` method:** Is used to indicate that the current process has completed its part of the transaction. Depending on the mode, this may result in a commit operation, or it may defer the commit to some other process.

```
public void
begin(javax.transaction.TransactionManager pTransactionManager,
int pTransAttribute) throws TransactionDemarcationException
```

- **`pTransAttribute` parameter:** Specifies the transactional mode (which is covered later)
- **`public int end()` method:** Returns a status code indicating what happened to the transactions during the demarcated area. See the API for a list of status codes and their meanings. This method can also be called by supplying a Boolean parameter indicating whether the transaction should roll back.

Using Transactions

To use transactions effectively, a developer must understand:

- The six transaction demarcation modes
- Programmatic control of transactions
- ATG's default behavior when no transactions are specified

Transaction Demarcation Modes

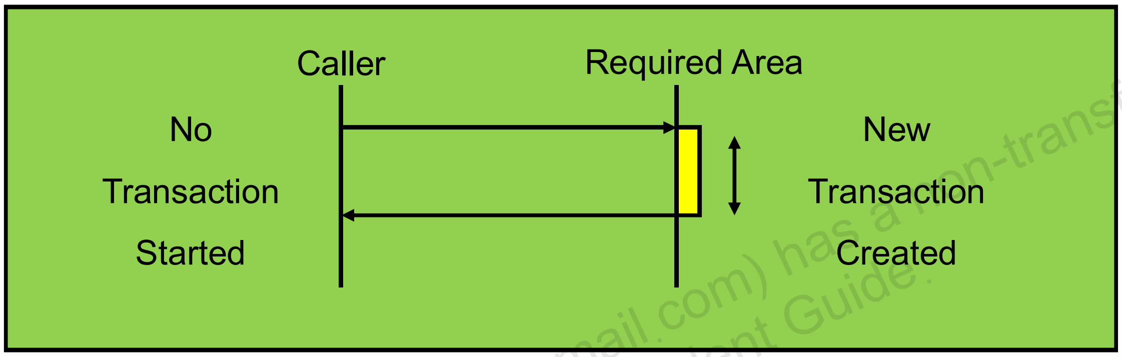
- *Transaction demarcation* is a term that is used to define the start and end points (the *boundaries*) of a transaction.
- There are six transaction demarcation modes that can be used to determine the boundaries:
  - `REQUIRED`
  - `SUPPORTS`
  - `MANDATORY`
  - `REQUIRES_NEW`
  - `NOT_SUPPORTED`
  - `NEVER`
- Transaction boundaries are set by using constants on the `TransactionDemarcation` object.

In addition to allowing transaction boundaries to be set programmatically, ATG allows these boundaries to be set declaratively in EJBs and in JSPs by using the transaction droplets. These methods are not covered in this course. For details, see the “Transaction Management” chapter in the *ATG Programming Guide*.

`REQUIRED` Transaction Mode: No Transaction Started

Example 1: No transaction started

- If no transaction exists when the demarcated area is entered, a new one is created.
- The transaction ends when the demarcated area ends.



Consider the bank transfer example from the beginning of this lesson.

The `transfer` method might be considered the "calling" method, shown on the left side of the diagram in the slide. The `deposit` method can then refer to the area on the right side of the diagram. In the example, the `transfer` method has not started a transaction. The deposit method has been marked, indicating that a transaction is `REQUIRED`. Therefore, a new transaction is created.

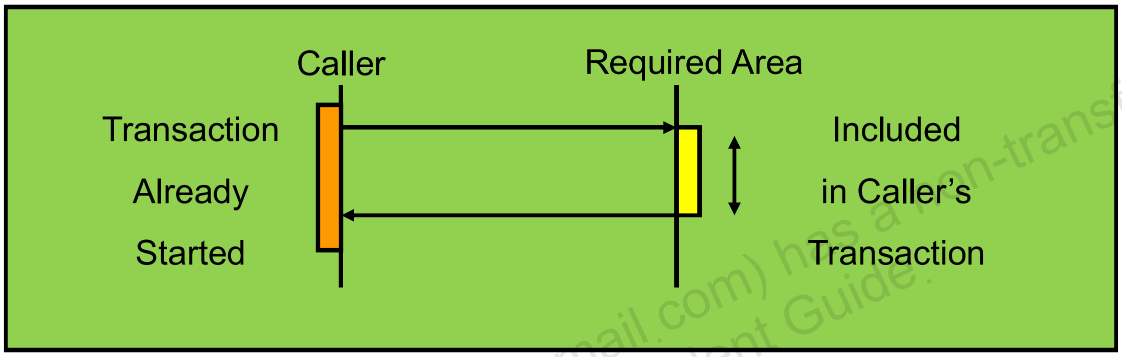
Implication

An exception or an explicit rollback in the required area rolls back only those changes that were made in that area. Changes made in the caller's scope are not rolled back because they are not executed within the transaction.

`REQUIRED` Transaction Mode: Transaction Already Started

Example 2: Transaction already started

Activity in the marked area is included in the transaction that has already started.



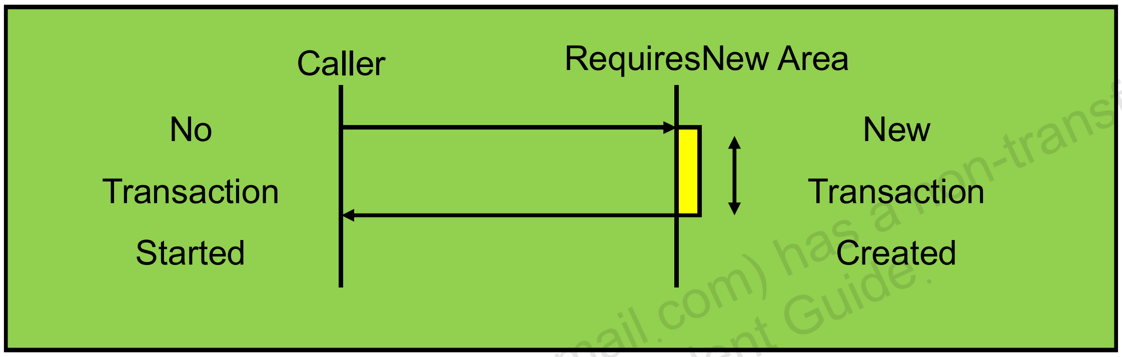
Implications

- An exception or an explicit rollback in the required area will roll back changes in the required area *and* in the caller's scope—all the way back to the beginning of the transaction.
- An exception or an explicit rollback in the caller's scope (after the required area has successfully completed) will roll back changes in *both* the caller's scope and the required area's scope.

`REQUIRES_NEW` Transaction Mode: No Transaction Started

Example 1: No transaction started

- If no transaction exists when the demarcated area is entered, a new one is created.
- The transaction ends when the demarcated area ends.



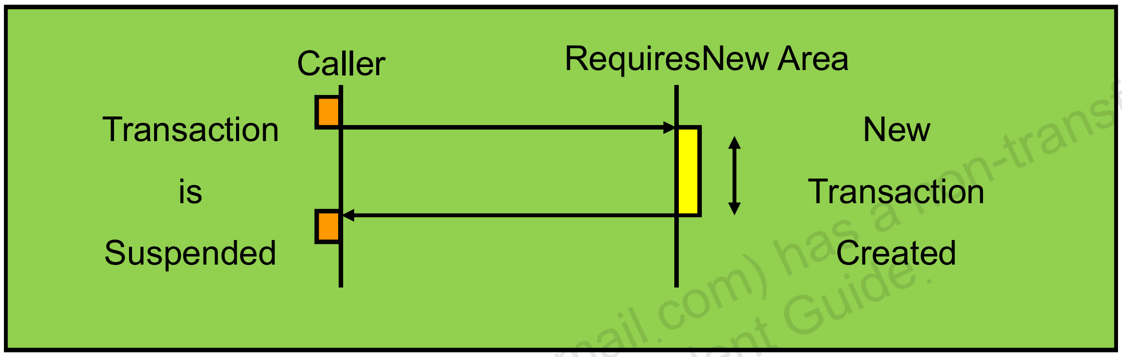
Implication

An exception or an explicit rollback in the RequiresNew area will roll back only those changes that were made in that area. Changes made in the caller's scope will not be rolled back because they are not executed within a transaction.

`REQUIRES_NEW` Transaction Mode: Transaction Already Started

Example 2: Transaction already started

- The first transaction is suspended.
- A new transaction is created.
- The first transaction resumes when the new transaction is complete.



Implications

- An exception or an explicit rollback in the RequiresNew area will roll back changes only in the required area.
- An exception or an explicit rollback in the caller's scope (after the RequiresNew area has successfully completed) will not roll back changes in the RequiresNew area's scope.

Note: Suspending a transaction can cause a deadlock situation to occur. This might happen if the current transaction has a lock in place and the new transaction tries to access the locked data.

Basic Example

Suppose that Transaction A begins and writes data to a profile. Then, within a RequiresNew demarcation area, the new transaction (Transaction B) also begins to write data to the same profile. Because Transaction A has a write lock on the database for this record, B is not able to obtain a read lock. A deadlock results.

Other Transaction Modes

Mode	Is the calling program in a transaction?	
	Yes	No
`REQUIRED`	Join	Start
`REQUIRES_NEW`	Suspend	Start
`MANDATORY`	Join	Exception
`NEVER`	Exception	No transaction
`SUPPORTS`	Join	No transaction
`NOT_SUPPORTED`	Suspend	No transaction

- **`REQUIRED`:** Use this mode when a section of code needs to be in a transaction and it is acceptable to join an existing transaction.
- **`REQUIRES_NEW`:** Use this mode when a section of code needs to be in its own transaction.
- **`MANDATORY`:** Use this mode when a section of code requires that it be part of a larger transaction.
- **`NEVER`:** Use this mode for long-running processes or for processes that require user input that should not end up in a transaction.
- **`SUPPORTS`:** This mode is mainly for completeness of the specification. The effect is the same if nothing is included.
- **`NOT_SUPPORTED`:** This mode is mainly for completeness of the specification.

Note: Caution should be taken when using the `NOT_SUPPORTED` and `REQUIRES_NEW` modes, because these modes can suspend the current transaction. If the new transaction attempts to use a resource that has been locked by the first transaction, a deadlock situation occurs.

Using `TransactionDemarcation`

To use the `TransactionDemarcation` class:

1. Create a `TransactionDemarcation` instance.
2. Call `begin()` at the start of your transaction and specify the demarcation mode.
3. If any errors occur that prevent the transaction from completing, call `TransactionManager.setRollbackOnly()`.
4. Call `end()` at the end of the transaction.

You should not call `commit()` or `rollback()` directly on the transaction when using transaction demarcation. Instead, you should call `setRollbackOnly`, and then `TransactionManager` automatically rolls back at the right time.

Why? If one process calls `commit()` on the transaction and a second process later tries to `end()` the demarcated area, an exception is generated. Consider an example in which the first process starts the transaction. A second process joins in the transaction in `REQUIRED` mode. Clearly, the second process should not roll back or commit the transaction.

Instead, by calling the `setRollbackOnly` method, you can be assured that the transaction will not be committed and that proper ownership of the transaction is preserved.

Using Transaction Control to Create New Songs: Part One

```
SongsManager sm = getSongsManager();
TransactionDemarcation td = new TransactionDemarcation();
    try {
      try {
          td.begin(sm.getTransactionManager(),
             td.REQUIRED);
          …create new song item in repository…
          artistid = sm.createArtistFromUser(userid);
          sm.addArtistToSong(songid,artistid);
      } //end second try
      catch (RepositoryException re) {
          …see next slide…
      } //end catch
      finally {
          td.end();
      } //end finally
    } //end first try
catch (TransactionDemarcationException tde) { … }
```

Consider a method that creates a new song item and then calls the previously discussed methods `createArtistFromUser` and `addArtistToSong`. You should make sure that if any of those steps fails, any changes made to the repository are rolled back. You do not want to have a song item in the repository without a valid artist. This can be handled (as in the example in the slide) by inserting a transaction demarcation "begin" before the song is created and an "end" demarcation after all modifications have been made.

The code in the slide also imports the `TransactionDemarcation` class by using import `atg.dtm.*;`.

The `TransactionDemarcation` class has static properties for each of the demarcation modes: `td.REQUIRED`, `td.REQUIRES_NEW`, `td.MANDATORY`, `td.NEVER`, `td.SUPPORTS`, and `td.NOT_SUPPORTED`.

The `begin` method can be called without the mode. In this case, `td.REQUIRED` is the default.

Using the `TransactionDemarcation` class provides you with a useful way to programmatically demarcate transactions within your ATG application.

Using Transaction Control to Create New Songs: Part Two

```
try {
    td.begin(getTransactionManager(),td.REQUIRED);
    artistid = sm.createArtistFromUser(userid);
    sm.addArtistToSong(songid,artistid);
}
catch (RepositoryException re) {
    if (isLoggingError())
      logError("Unable to add artist to song", re);
    try {
      sm.getTransactionManager().setRollbackOnly();
    }
    catch(SystemException e) {
      if (isLoggingError())
          logError("Rollback failed too", e);
    } //end second catch
} //end first catch
finally { td.end(); }
```

In the previous example, what if a problem occurs when attempting to create and modify the song? An exception would be caught, and in handling the exception, you would have to set the transaction to "rollback only."

`setRollbackOnly` is a method on the `javax.transaction.TransactionManager` interface. The method signature is:

```
public void setRollbackOnly() throws
java.lang.IllegalStateException, javax.transaction.SystemException
```

Default Transaction Behavior

Default behavior of the Repository API:

- If a JTA transaction is in progress, all repository changes are part of that transaction and are committed when the transaction completes.
- If there is no transaction in progress, each API call that modifies the repository occurs in its own transaction.
- `RepositoryFormHandler` includes all repository changes within a single transaction.

Both `ProfileFormHandler` and `RepositoryFormHandler` handle transactions intelligently. That is, all updates on items made by the form are enclosed within a transaction. If you subclass either form handler and override the “pre” or “post” methods, those methods are also included within the transaction.

Therefore, the code shown in this lesson is not necessary within the context of a form handler that subclasses from one of the supplied form handlers (as discussed in the lesson titled “Custom Form Handlers”). However, including the code will do no harm.

Debugging Transactions in ATG

- Setting the `loggingDebug` property of the `/atg/dynamo/transaction/TransactionManager` to `true` shows each transaction as it is created, as it enlists resources, and as it is committed or rolled back.
- You may also want to set the `loggingDebug` property on your repository component to `true`.

When you set the `loggingDebug` property to `true` on your repository component, you see the actual SQL generated. This is a verbose action, so make sure you leave it set to `false` in your production environment.

Quiz

The `TransactionDemarcation` class exposes JTA transaction boundaries to the ATG programmer.

a. True

b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Describe ATG's support for the Java Transaction API (JTA) specification
- Use the `TransactionDemarcation` class to provide simplified control of transactions instead of using JTA directly

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Programming Guide*
  - "Transaction Management"
- *ATG API Platform Reference*
  - `atg.dtm` package
- http://www.oracle.com/technetwork/java/javaee/jta/index.html
  - Includes the API Reference, tutorial, and so on

Practice 4 Overview: Managing Transactions

This practice covers adding transaction management to `deleteAlbumsbyArtist()` so that if one album deletion fails, all the others are rolled back.

5

Complex Repository Configuration

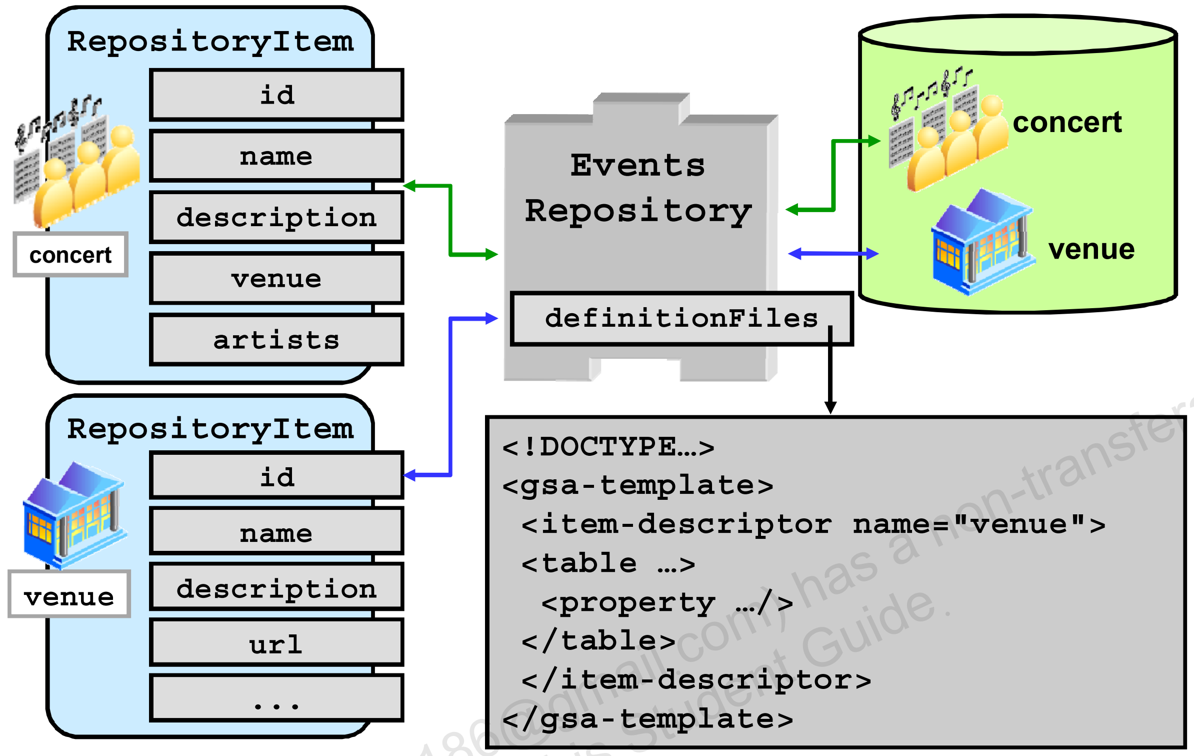
Objectives

After completing this lesson, you should be able to model:

- Different kinds of one-to-many relationships
- Many-to-many relationships in repositories
- A composite repository ID

In *Foundations of ATG Application Development*, you learned how to create one-to-many relationships by using a `Set`. In many cases, you need a more complex relationship. This lesson covers how to create a many-to-many relationship, how to create a composite primary key, and some of the other one-to-many relationship types.

Repository Review: EventsRepository



```
<!DOCTYPE…>
<gsa-template>
 <item-descriptor name="venue">
 <table …>
  <property …/>
 </table>
 </item-descriptor>
</gsa-template>
```

In *Foundations of ATG Application Development*, you created a new SQL repository to hold information about events. The repository consisted of a Nucleus component called `EventsRepository`, whose `definitionFiles` property pointed to the `events.xml` file. This XML file maps the database tables and columns to the repository's objects.

Some reminders about repositories include the following:

- A property can be simple (String, integer, and so on) or enumerated, or it can refer to other repository items.
- Properties can be single-valued or multi-valued.

An example of a multi-valued property is the `artists` property on the `concert` item type. A concert can have more than one artist, so this property refers to a set of `artist` items. The `artist` item is in the `SongsRepository`, so it is also an example of a property referring to another repository.

For a review of the XML definition file for this repository, refer to the `<`*`atgdir`*`>/DynamusicEAP/config/dynamusic/events.xml` file.

Dynamusic Repositories

- Dynamusic needs to make the repositories more flexible.
  - The site wants to add tours.
    - A tour contains multiple concerts.
    - Tours appear at multiple venues, and a venue can have more than one tour.
  - Albums can contain multiple artists.
- Dynamusic has a legacy database containing classic posters for concert promotions, but those posters have a composite ID.

In the *Foundations of ATG Development* course, you created `EventsRepository`, which implemented the concepts of *concerts* and *venues*, but did not implement the concept of a concert tour. A *concert tour* is a series of concerts at multiple venues.

`SongsRepository` was configured to allow an album to have one artist. The `artist` item, however, had no concept of its related albums. Dynamusic would like to allow an album to contain multiple artists and to have the artist object store a link to all of the albums that include the artist.

Road Map

- One-to-many relationships
- Many-to-many relationships
- Composite repository IDs

Review: Types of Multi-Valued Properties

- In *Foundations of ATG Application Development*, you learned how to use a Set for the multi-valued property type.
  - A `Set` holds unordered data.
- Other multi-valued types are supported, requiring an additional column for sequence or index.
  - `List`: An ordered collection instantiated as a Java List
  - `Map`: A collection with key-value pairs
  - `Array`: An ordered collection instantiated as a Java array

`Set` is the easiest data type to use, which is why you learned about it first. Your application, however, might require data in a different form. The decisions about what type to use depend on your application's requirements.

For example, images associated with an album in a set would be unordered. What if the images were always from the CD insert liner notes, and Dynamusic wanted to be able to display them by page number ("click on the number to see that page of the CD inserts…")? What data type would support that functionality?

On the other hand, what if Dynamusic had different types of images available and wanted to be able to display them by type, such as "front cover," "back cover," "promotion poster," and so on? What data type would support labeling each image according to type?

Configuring Maps, Lists, and Arrays

- An additional `<table>` tag attribute identifies the key/sequence column: `multi-column-name`.
- Additional attributes for the `<property>` tag are:
  - `data-type` values: `map`, `list`, or `array`
  - If primitive, specify `component-data-type` attribute.
  - If referring to another item type, specify `component-item-type`.

ORACLE

A `Map` is a Java Collection type that binds keys to values. A map cannot contain duplicate keys. Each key can map to at most one value.

To support this Collection type, the "multi" table should contain two columns: one for the key and the other for the values.

Note: The key value `multi-column-name` will hold one of several things: a key value for a dictionary or map, an index for an array, or a count for a list. The "multi" in `multi-column-name` is related to the fact that it has multiple uses (and not the fact that it deals with multiple columns).

One-to-Many Relationships Using a List

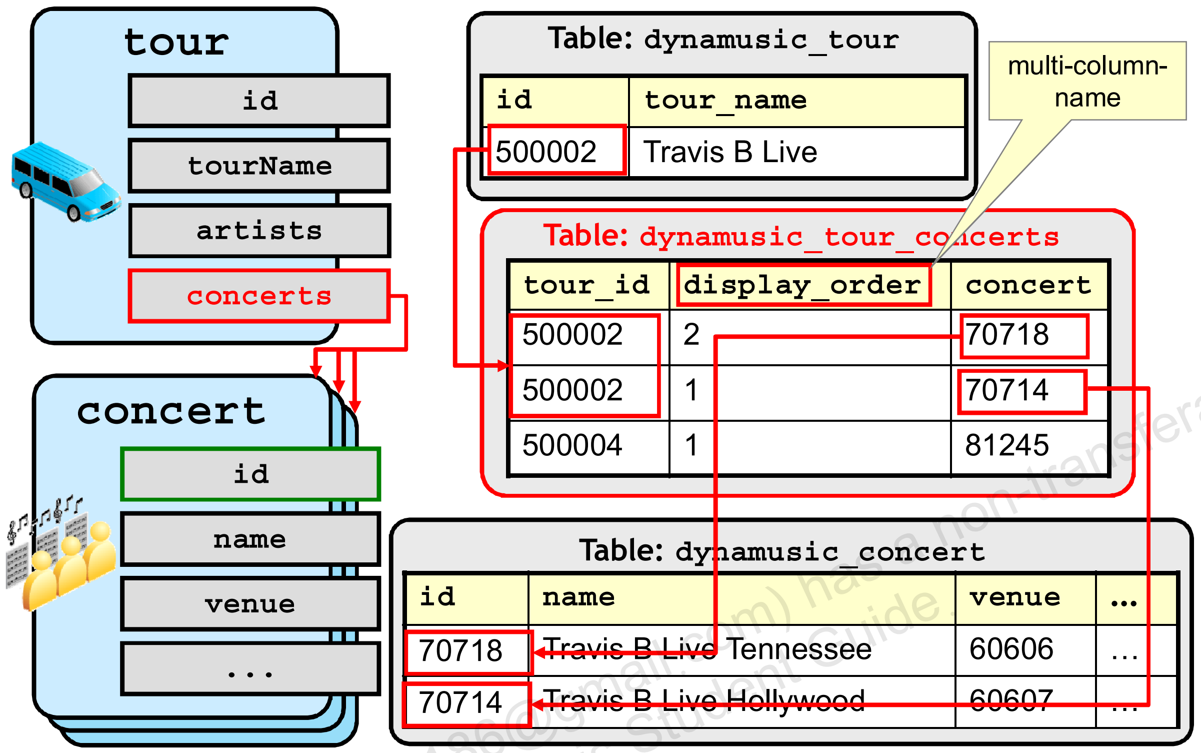


Table: dynamusic_tour

id	tour_name
500002	Travis B Live

Table: dynamusic_tour_concerts

tour_id	display_order	concert
500002	2	70718
500002	1	70714
500004	1	81245

Table: dynamusic_concert

id	name	venue	...
70718	Travis B Live Tennessee	60606	...
70714	Travis B Live Hollywood	60607	...

Dynamusic would like to display information about concert tours, which typically contain more than one concert. To set up the relationship between the tour object and its concerts, Dynamusic defines a one-to-many relationship as a list. Merchandisers would like the ability to specify the order in which concerts are displayed so that they can promote certain concerts. The index number of the list provides one way for the application to specify the display order on the tour page. To map this, Dynamusic sets up three tables:

- **`dynamusic_tour`:** Holds the tour ID and name for each tour
- **`dynamusic_concert`**: Stores the information about each individual concert
- **`dynamusic_tour_concerts`:** Connects the tour to concerts as a "multi" table with three columns: `tour_id`, `display_order`, and `concert` (to hold the concert ID). The `display_order` column is defined as the `multi-column-name` in the XML file.

In the example in the slide, Travis B's Hollywood concert will be displayed first, followed by his Tennessee concert.

Note: The `dynamusic_tour` table has only two columns. All other properties are handled through relationships, which are stored in different tables. For instance, the `artists` property is a one-to-many relationship between the `tour` item and the `artist` item. For more information about the full definition for the tour item, refer to the `<coursedir>/examples/lesson05/events.xml` file.

Defining a Tour in `events.xml`

```
...
<item-descriptor name="tour" display-property="tourName">
    <table name="dynamusic_tour" type="primary"
      id-column-name="id">
      <property name="id" column-name="id"
       data-type="string"/>
      <property name="tourName" column-name="tour_name"
       data-type="string" />
    </table>
...
    <table name="dynamusic_tour_concerts" type="multi"
      id-column-name="tour_id"
      multi-column-name="display_order">
      <property name="concerts" data-type="list"
      component-item-type="concert" column-name="concert" />
    </table>
</item-descriptor>
...
```

The `tour` item is in the `EventsRepository` and is defined in the `events.xml` file. In the example in the slide, the `dynamusic_tour_concerts` table is defined as a multi table (`type="multi"`) with the tour ID as the `id-column-name`. The `multi-column-name` is set to the `display_order` table. The `concerts` property is defined as a list `data-type` that holds `concert` items in the `concert` column. You do not need to include the repository attribute for the property, because the `concert` item type is defined in the same repository as the `tour` item type.

Road Map

- One-to-many relationships
- Many-to-many relationships
- Composite repository IDs

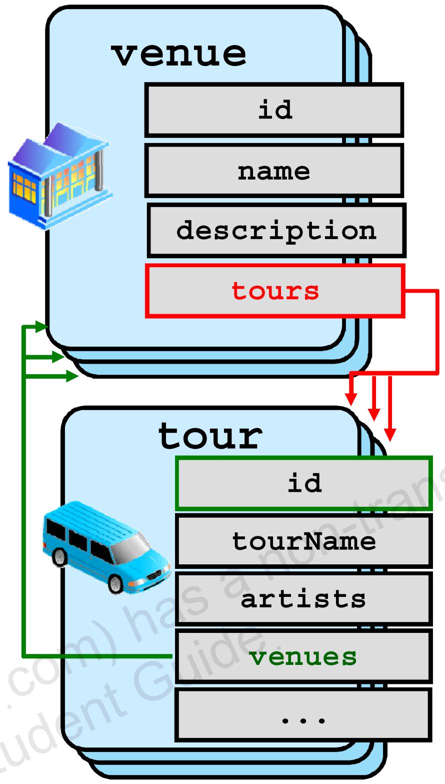
Multi-Valued Properties: Many-to-Many

Form a two-way relationship:

- Like two one-to-many relationships
- Must use a Set

Example:

- A tour appears at multiple venues.
- A venue can hold multiple tour events.



A many-to-many relationship is basically a two-way mapping of a one-to-many relationship. When defining a many-to-many relationship, you use a `Set`.

Multi-Item Properties: Many-to-Many

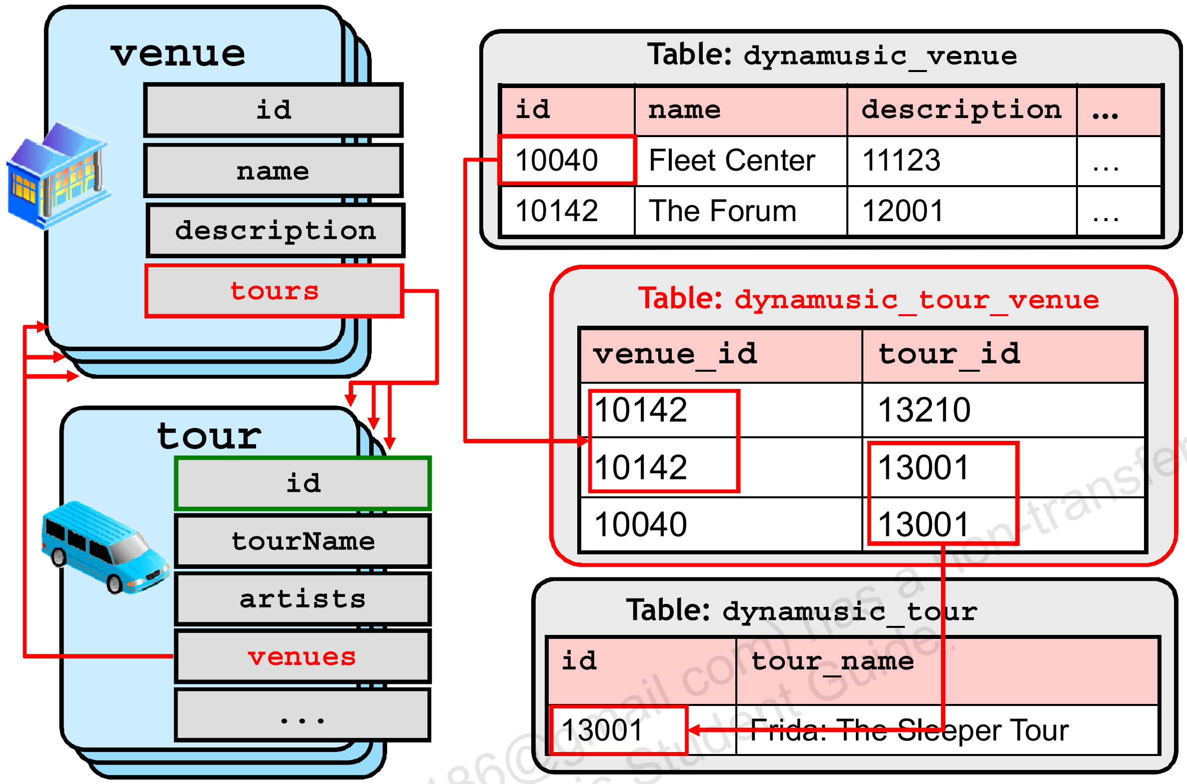


Table: dynamusic_venue

id	name	description	...
10040	Fleet Center	11123	…
10142	The Forum	12001	…

Table: dynamusic_tour_venue

venue_id	tour_id
10142	13210
10142	13001
10040	13001

Table: dynamusic_tour

id	tour_name
13001	Frida: The Sleeper Tour

Consider two types of items: venues and tours. A venue hosts many different tours, and a tour takes place in multiple venues. To map this, you need a many-to-many relationship.

In such a mapping, each `venue` item has a `tours` property that is a collection of references to `tour` items. In turn, each `tour` item has a `venues` property that is a collection of references to `venue` items.

The SQL for this relationship is very similar to the SQL for a one-to-many relationship. The following SQL code creates the `dynamusic_tour_venue` table in Oracle Database:

```
CREATE TABLE dynamusic_tour_venue (
 tour_id VARCHAR(32) not null references
  dynamusic_tour(id),
 venue_id VARCHAR(32) not null references
  dynamusic_venue(id),
 primary key(tour_id, venue_id)
);
```

Defining Tour Venues in `events.xml`

```
<item-descriptor name="tour" display-property="name">
<table name="dynamusic_tour" type="primary"
id-column-name="id">
<property name="name" data-type="string"/>
<property name="description" data-type="string"/>
</table>
<table name="dynamusic_tour_venue" type="multi"
id-column-name="tour_id">
<property name="venues" column-name="venue_id"
data-type="set" component-item-type="venue"/>
</table>
</item-descriptor>
```

In the `tour` item descriptor, the one-to-many relationship is defined as a set (as required in a many-to-many relationship).

Note: The item descriptor closely resembles an item descriptor for a one-to-many mapping. The only difference in the example in the slide is that the other item, instead of being a stand-alone table mapping, also maps a one-to-many relationship going the other way. This other mapping is shown in the next slide.

Defining a Venue's Tours in `events.xml`

```
...
<item-descriptor name="venue" display-property="name">
   <table name="dynamusic_venue" type="primary"
     id-column-name="id">

     <property name="name" data-type="string"/>
     <property name="description" data-type="string"/>
   </table>

   <table name="dynamusic_tour_venue" type="multi"
     id-column-name="venue_id">

     <property name="tours" column-name="tour_id"
         data-type="set" component-item-type="tour"/>
   </table>
</item-descriptor>
   ...
```

The example continues in this slide with the second one-to-many relationship between a venue and tours. The full code example can be found in `<coursedir>/examples/lesson05/events.xml`.

Note: This item descriptor closely resembles an item descriptor for a one-to-many mapping. The only difference in this example is that the other item, instead of being a stand-alone table mapping, also maps a one-to-many relationship going the other way.

Quiz

When creating a many-to-many relationship, you can use `set`, `map`, `list`, or `array` as your `data-type`.

a. True

b. False

Answer: b

Road Map

- One-to-many relationships
- Many-to-many relationships
- Composite repository IDs

ORACLE

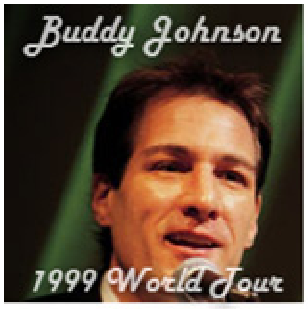
Review: Repository IDs

- Every repository item must have a unique ID.
- This ID is exposed through the `repositoryId` property of `RepositoryItem`.
- The `repositoryId` property is always modeled as a string, regardless of how the ID is stored in the database.
  - Developers can also choose to model the ID as a separate property, which can have any data type.
- The `repositoryId` property is populated from the `id-column-name` attribute of the primary table.

Dynamusic Classic Posters

Dynamusic has a legacy database containing classic posters for concert promotions.

- This database tracks the artist and the tour.
- Posters are identified by both the artist ID and the tour ID.



artistId=100002

tourId=100057

poster's repository id=100002:100057

By default, the ID separator is a colon (`:`). You can, however, specify your own separator by using the `id-separator` attribute in the `item-descriptor` tag. For more details, see the slide titled “Defining Posters in `songs.xml`.”

Composite Repository IDs

- A composite ID is a unique ID represented by more than one column in the database.
  - Composite IDs are often found in legacy databases.
- The `repositoryID` property concatenates the ID's elements, in the order specified by the item descriptor's `id-column-names` attribute.

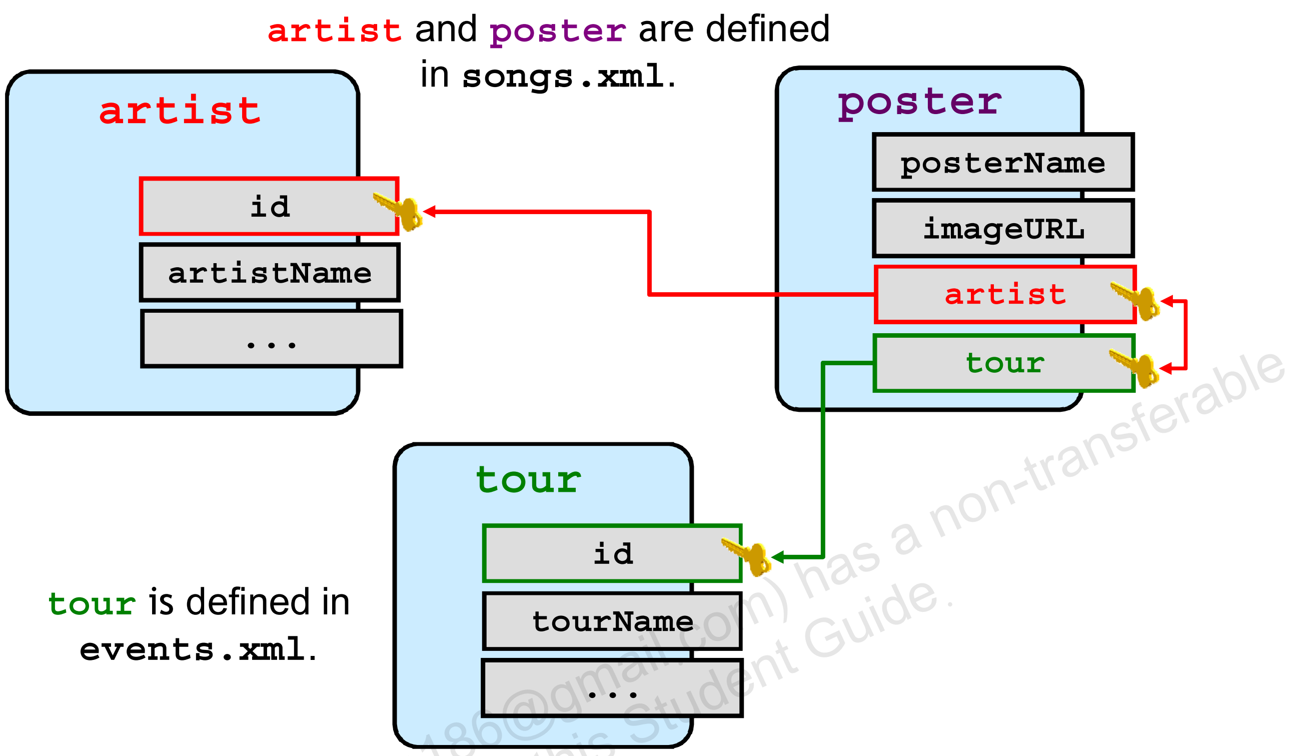
Table: `dynamusic_poster`

`artist_id`	`tour_id`	`poster_name`	`image`
100004	500004	The Brighton Bramblers 1964 Tour	images/posters/Bramblers64.gif
100006	500006	Elwin Redshoes 1979 Tour	images/posters/redshoes79.jpg

A repository ID can also be represented by more than one column in the database, each column of which can be either String, Integer, or Long (the same as a simple ID). This type of repository ID is also referred to as a *multicolumn ID* or *composite key ID*.

As the repository creator, you can determine what the separator is between the IDs. By default, the separator is a colon(`:`), but you can choose a different character.

Classic Concert Poster Item Types


artist and poster are defined in songs.xml.
artist
id
artistName
...
poster
posterName
imageURL
artist
tour
tour
id
tourName
...
tour is defined in events.xml.

Dynamusic has three tables that hold information for concert posters:

- **`dynamusic_artist`**: Same table that you previously used for the artist
- **`dynamusic_tour`**: Legacy table for the tour
- **`dynamusic_poster`**: Legacy table for the concert poster that links `artist` to the artist object and links `tour` to the tour object

Defining Posters in `songs.xml`

```
...
<item-descriptor name="poster"
    display-property="posterName">

    <table name="dynamusic_poster" type="primary"
      id-column-names="artist_id,tour_id">
      <property name="posterName" column-name="poster_name"
          data-type="string"/>
      <property name="imageURL" column-name="image"
          data-type="string"/>

      <property name="artist" column-name="artist_id"
          item-type="artist"/>
      <property name="tour" column-name="tour_id"
       repository="/dynamusic/EventsRepository"
          item-type="tour" />
    </table>
</item-descriptor>
...
```

The item descriptor that defines a composite repository key uses the `id-column-names` attribute instead of `id-column-name`. You can actually use `id-column-names` in any item descriptor, even if only one ID column is necessary. ATG uses a separator character to concatenate the ID's elements. By default, that separator is a colon (`:`). You can, however, specify your own separator by using the `id-separator` attribute in the `item-descriptor` tag.

Because the `tour` item type is defined in a different repository, the definition for that property includes the `repository` attribute.

Note: You should not use brackets (`[]`) or commas (`,`) for the separator character, because these characters are used by RQL and the SQL repository to specify lists of IDs.

Summary

In this lesson, you should have learned how to:

- Model different kinds of one-to-many relationships in a repository
- Define a many-to-many relationship between two repository items
- Create the repository definition to handle a composite database ID

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Repository Guide*
  - "Repository Queries"
    - "Repository Query Language"
  - "SQL Repository Data Models"
    - "id Property"
    - "One-to-Many Relationships: Multi-Valued Properties"
    - "Many-to-Many Relationships"
  - "SQL Repository Reference"

Practice 5 Overview: Configuring a Many-to-Many Relationship

This practice covers configuring a many-to-many relationship between artists and albums, including the following:

- Adding an `artists` property to the `album` item type
- Adding an `albums` property to the `artist` item type

In *Foundations of ATG Application Development*, you created `SongsRepository` and defined an `album` item with an `artist` property. This allowed you to assign only a single artist to an album.

In this practice, you add a new second property called `artists`, which allows you to assign more than one artist to an album. The `artist` property remains in the application for backward compatibility but is normally removed. You also configure the `artist` item in `SongsRepository` to have a new property called `albums`. This is a new relationship.

6

Advanced SQL Repository Configuration

Objectives

After completing this lesson, you should be able to:

- Add additional logic to manipulate properties
- Describe how SQL content repositories differ from regular repositories

Road Map

- Custom property logic
- SQL content repositories

Custom Property Logic

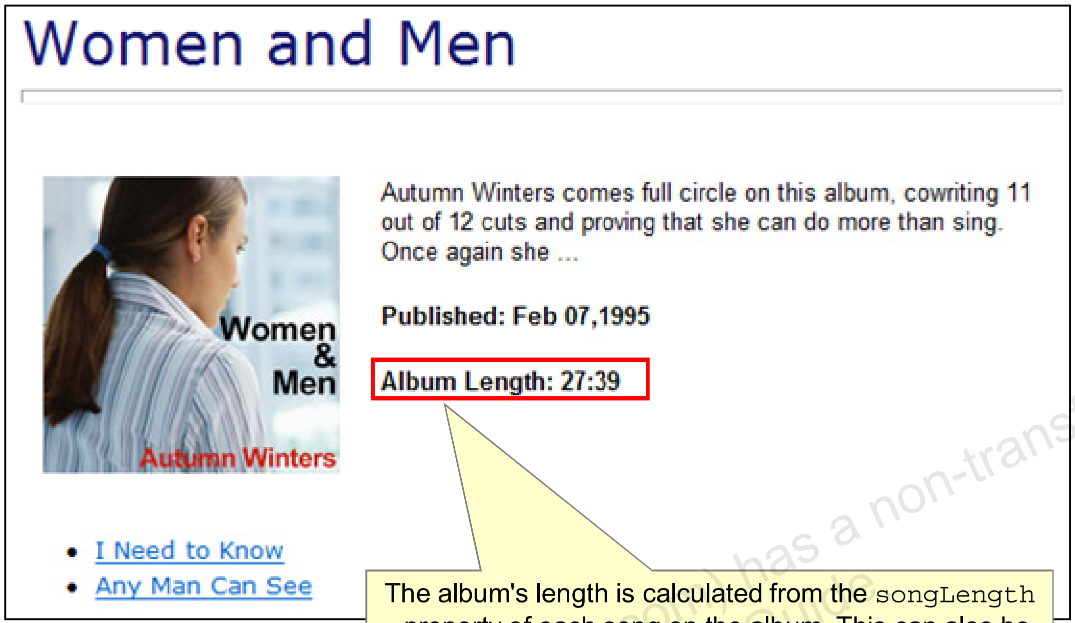
- By default, properties of repository items are instantiated as objects of `GSAPropertyDescriptor` or `RepositoryPropertyDescriptor`.
- Custom Java classes can provide special handling for repository item properties.
- Here are some examples:
  - Transient properties that are computed “on the fly” from values from the database
  - Values that must be modified or processed before storage or after retrieval from the database

Custom properties enable you to specify additional logic to manipulate properties.

When ATG SQL repositories are configured, each property in an item descriptor is usually represented by an instance of the `GSAPropertyDescriptor` class. This object has methods to configure all the property’s attributes, as well as to get and set that property’s values for a repository item.

Custom properties enable you to use your own class rather than the default.

Dynamusic Example: `albumLength`



Autumn Winters comes full circle on this album, cowriting 11 out of 12 cuts and proving that she can do more than sing. Once again she ...

Published: Feb 07,1995

Album Length: 27:39

- I Need to Know
- Any Man Can See

The album's length is calculated from the `songLength` property of each song on the album. This can also be used to model the playlist's length.

The screenshot in the slide shows the webpage for a particular album.

Creating a Custom `property-type`

To define your own `property-type`:

- Create a `PropertyDescriptor` class for your property type.
  - Write a Java class that extends either of the following:
    - `GSAPropertyDescriptor` (nontransient properties)
    - `RepositoryPropertyDescriptor` (transient properties)
- Set `property-type=`*`YourClass`* in the item descriptor.

The full names of the base classes:

- `atg.adapter.gsa.GSAPropertyDescriptor`
- `atg.repository.RepositoryPropertyDescriptor`

Use `GSAPropertyDescriptor` for properties that are stored in the database (that is, for properties whose descriptors appear inside a `<table>` tag).

Use `RepositoryPropertyDescriptor` for transient properties (that is, for properties whose property descriptors appear outside a `<table>` tag).

A full list of the methods of both classes is available in the *API Platform Reference*.

Note: If you will be using your custom property type for many item properties, you may want to "register" it so that you can refer to it by name rather than by the full classpath. In the *ATG Repository Guide*, you can find more information about registering custom property types in the "User-defined Property Types" section of the "SQL Repository Item Properties" chapter.

Property Descriptor Objects

- To change how property values are stored or retrieved, override the following methods:
  - `getPropertyValue()`
  - `setPropertyValue()`
- Two parameters are passed to `getPropertyValue()`:
  - `pItem`: A reference to the entire repository item (for example, the album)
  - `pValue`: The cached value of this property. To store a value in the cache, use the `setPropertyValueInCache()` method of the repository item.

When an XML definition file is parsed, an instance of a property descriptor is created for each property that is defined for an item descriptor. The property descriptor is a bean whose methods enable you to set attributes of a repository item's properties. The `RepositoryPropertyDescriptor` class has properties such as:

- `propertyValue`
- `defaultValue`
- `propertyType`
- `queryable`
- `cachable`
- `required`
- `readable`

The subclass `GSAPropertyDescriptor` adds properties such as:

- `columnName`
- `table`
- `dataType`

The full list of methods of both classes is in the *API Platform Reference*.

Sometimes you want to customize how a value is returned but not how it is saved. In that case, you override `getPropertyValue()` but not `setPropertyValue()`. For example, you may want a property that dynamically determines a “default value” if the property is currently null, but allows a value to be set normally.

Conversely, you may want to have a property that is set specially but returned normally. To do that, you override `setPropertyValue()` but not `getPropertyValue()`. For example, you may want a password value to be hashed before it is stored in the database.

The method signatures are:

- `public java.lang.Object getPropertyValue(RepositoryItemImpl pItem, java.lang.Object pValue)`
  The first parameter is the repository item whose value is being computed, and the second parameter is the value for this property retrieved from cache (null if the item was not in cache).
- `public void setPropertyValue(RepositoryItemImpl pItem, java.lang.Object pValue)`
  The first parameter is the repository item being changed; the second parameter is the value to which it is being set.

The `get/setPropertyValue` methods of the property descriptor are called by the `get/setPropertyValue` methods of the underlying repository item. In other words, whenever a repository item’s properties are retrieved or modified, the property descriptor’s `get/setPropertyValue` methods are called. Thus, these methods can do any required special processing when getting or setting a repository item’s property values.

Note that these methods have access to the whole repository item and not just to the individual property being accessed. This enables you to use the item’s other properties to calculate the requested property, or even modify the item’s other properties.

Overriding `getPropertyValue()` in `dynamusic.TimeCalc`: Part One

```
public Object getPropertyValue(RepositoryItemImpl pItem,
    Object pValue) {

    if (pValue != null) return pValue;

    int totalSeconds=0;
    Set albumSongs =
      (Set) pItem.getPropertyValue("songList");
    Iterator i = albumSongs.iterator();

    while(i.hasNext()) {
      RepositoryItem currentSong =(RepositoryItem)i.next();

      Integer lengthInSeconds = (Integer)
      currentSong.getPropertyValue("songLength");
      totalSeconds += lengthInSeconds.intValue();
    } //end while
    ...
```

In the code example in the slide, you see a class called `TimeCalc` that extends `atg.repository.RepositoryPropertyDescriptor` and that is used with the Songs Repository. This class calculates the total length of an album from the songs in its `songList` property. Because this is a read-only property, you do not need to override the `setPropertyValue` method. However, you want to override the `getQueryable` and `getWritable` methods so that both return false (not shown here but in the full code for the `TimeCalc` class.)

Note: You may notice that `songLength` is being cast into an Integer wrapper class. The `getPropertyValue` method returns an `Object`, but `songLength` is declared as an `int` (in `songs.xml`).

The full code for `TimeCalc.java` appears in `<coursedir>/examples/lesson06/TimeCalc.java`.

Overriding `getPropertyValue()` in `dynamusic.TimeCalc`: Part Two

```
   ...
   int minutes = totalSeconds / 60;
   int seconds = totalSeconds % 60;
   String length = minutes + ":" + seconds;
   pItem.setPropertyValueInCache(this, length);
   return length;
} //end of getPropertyValue()
```

The rest of the code calculates minutes and seconds to display length in the proper format.

Defining `albumLength` in `songs.xml`

```
<gsa-template>

   <item-descriptor name="album">
   ...
     <property name="albumLength"
         property-type="dynamusic.TimeCalc"
     writable="false"
         data-type="string">
     </property>

   </item-descriptor>

</gsa-template>
```

This sample code in the slide shows how to add the new `albumLength` property to the album item descriptor in the Songs Repository. The new property is called `albumLength`, and it uses the `dynamusic.TimeCalc` class shown in the previous slide.

Note that the `albumLength` property is not within a `<table>` tag. Thus, it is a transient property because you are computing the value whenever it is requested rather than storing it in the database.

In the example in the slide, the `albumLength` property includes the `data-type` attribute. This is optional. However, if you want to see the property in the ACC and Business Control Center (BCC), you must either set the data type or return the `propertyType`:

```
public Class getPropertyType() {
 return String.class;
}
```

Property Attributes

User-defined properties can use custom XML attributes to control behavior:

- Those attributes are passed to the `setValue()` method of the property descriptor.
- `setValue()` is called once for each specified attribute.

The `setValue` and `getValue` methods of the `atg.repository.RepositoryPropertyDescriptor` class are:

- `public java.lang.Object getValue(String pAttributeName)`
- `public void setValue(String pAttributeName, Object pValue)`

Dynamusic Example

- Dynamusic wants to reuse the code in `dynamusic.TimeCalc` to iterate through the lists of songs in both albums and playlists without duplicating the calculation.
- The example in the following slides shows the following:
  – The XML configuration for the `album` and `playlist` item descriptors to include the custom attribute
  – The custom attribute in the Java code

In the previous example of the `dynamusic.TimeCalc` class, the code iterated through the album's `songList` property. This works well for that specific item but is not very flexible.

The code example in this section adds a custom attribute called `SongLocation` to the new custom property. You can then define which property of the item to iterate through, enabling the property to be used in different item types.

The next few slides show how to define the custom attribute in XML and how the attribute is coded in the Java class.

Using `dynamusic.TimeCalc` with a Custom Attribute in `songs.xml`

```
<item-descriptor name="album">

   <property name="albumLength"
     property-type="dynamusic.TimeCalc"
     writable="false">

      <attribute name="songLocation" value="songList"/>

   </property>

...

</item-descriptor>
```

In the slide, `dynamusic.TimeCalc` is used to configure two different properties: `album.albumLength` and `playlist.playlistLength`.

This example shows how the new custom attribute called `songLocation` is configured in the `album` item descriptor. The property that contains the list of an album's songs is called `songList`.

Using `dynamusic.TimeCalc` with a Custom Attribute in `userProfile.xml`

```
<item-descriptor name="playlist">

   <property name="playlistLength"
     property-type="dynamusic.TimeCalc"
     writable="false">

      <attribute name="songLocation" value="songs"/>

   </property>

...

</item-descriptor>
```

In the slide, `dynamusic.TimeCalc` is used to configure two different properties: `album.albumLength` and `playlist.playlistLength`.

This example shows how the new custom attribute called `songLocation` is configured in the `playlist` item descriptor. The property that contains the list of a playlist's songs is called `songs`.

Using `songLocation` in `dynamusic.TimeCalc`

```
...
String mSongLocation = "songs";

public void setValue(String pAttributeName, Object
    pValue) {
    super.setValue(pAttributeName, pValue);
    if (pAttributeName.equalsIgnoreCase("songLocation")){
      mSongLocation = pValue.toString();
    } //end if
} //end setValue()

public Object getPropertyValue(RepositoryItemImpl pItem,
    Object pValue) {
    ...
    Set albumSongs = (Set)
      pItem.getPropertyValue(mSongLocation);
    ...
} //end getValue()
```

The code example in the slide declares the additional member value called `mSongLocation` and overrides the `setValue` method to catch the `songLocation` attribute. This value is then used when retrieving the list of songs from the item.

Attribute Value Types

- The data type of attributes can be specified by using `data-type`.
  - The default is `string`.
  - Possible data types are `int`, `short`, `byte`, `long`, `float`, `double`, `string`, `date`, `boolean`, and `timestamp`.
- Attribute values can be set to components or properties of components by using `bean` instead of `value`.

```
<attribute name="songLocation" value="songList"
data-type="string" />
```

```
<attribute name="format"
bean="/atg/userprofiling/Profile.preferredFormat" />
```

ORACLE

The slide shows two examples. The first example shows how to use `data-type` for the attribute. The second example shows how to set the attribute value to a component by using `bean` instead of `value`.

Quiz

When you create a custom property type, which of the following steps are valid for every customization? Select all that apply.

a. Create a property descriptor class that extends `GSAPropertyDescriptor` or `RepositoryPropertyDescriptor`.

b. Set the `property-type` attribute to your class in the repository definition file.

c. Override both the `getPropertyValue` and `setPropertyValue` methods.

d. Set `writable=false` for the custom property definition.

e. All of the above

Answer: a, b

Answer c: You need to override the methods only if you want to change how property values are stored or retrieved. For example, if you want a password value to be hashed before being stored in the database but have it returned normally, you override the `setPropertyValue` method.

Answer d: You set the `writable` attribute for the property to `false` only if you want the property to be read-only.

Road Map

- Custom property logic
- SQL content repositories

SQL Content Repositories

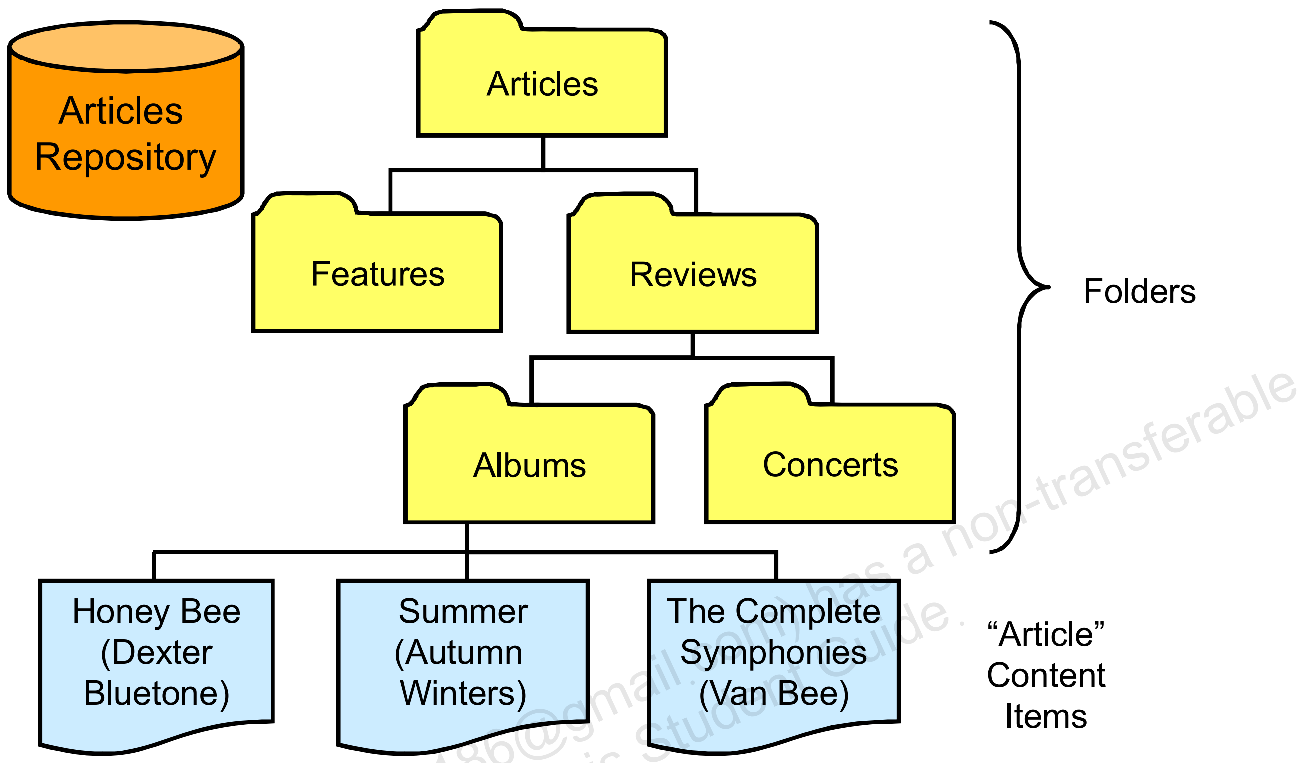
- In their original implementation, repositories were file-based, using folders in the file system as a way of organizing items. SQL repositories were added later.
- SQL content repositories model the original file system.
- Any SQL repository can contain content. A SQL content repository is a repository that is organized into a pseudo-folder structure.
- They are also called *hierarchical repositories*.

SQL content repositories are SQL repositories that mirror a file system. This concept of a SQL content repository is built into the architecture of ATG. It is helpful to know the history of repositories in ATG to understand why this is relevant.

- **ATG 3 and earlier versions:** No repositories
- **ATG 4:** Repositories were introduced but were built on top of files. This was designed for file-based content to provide targeting capabilities. The content consisted of HTML or XML files with metadata describing what was in the file, such as author, title, and so on.
- **ATG 5:** Repositories were extended to also be able to pull data from the database. When this functionality was introduced, it was necessary (for backward compatibility) to be able to mimic the concept of file-based repositories in the database structure. You mostly do not need this, but some places in the ATG infrastructure still call for this type of structure.

Note: For a more in-depth look at SQL content repositories, refer to Appendix A (titled "Content Repositories") in the Student Guide for this course, and to the chapter titled "SQL Content Repositories" in the *ATG Repository Guide.*

Example: Folder Hierarchy

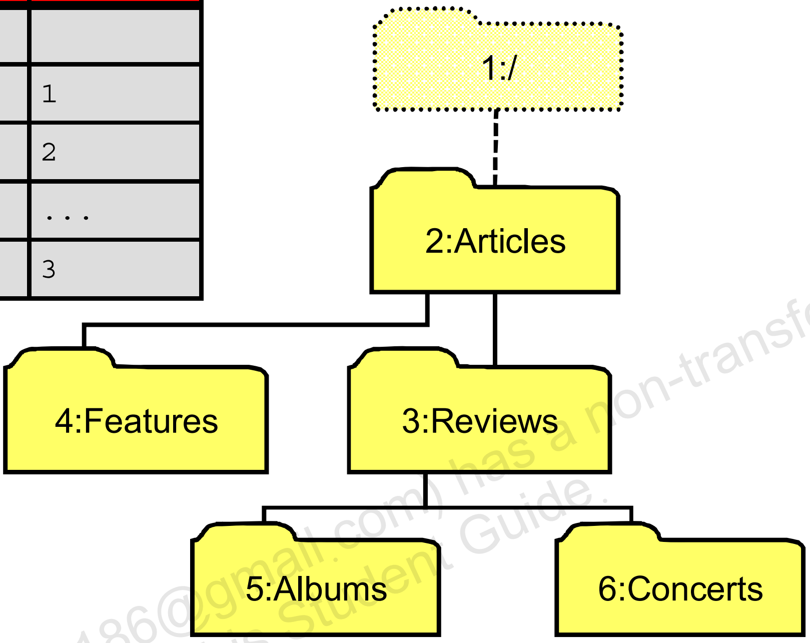


In the example in the slide, you have a repository consisting of articles about music. The folder hierarchy starts with an Articles folder with two subfolders: Features and Reviews. The Reviews folder has two subfolders as well: Albums and Concerts. There are three albums in the Albums folder that represent the article content items.

The full path to a content item includes the folder hierarchy. For example, the full "path" for the "Honey Bee" content item (article) is `/Articles/Reviews/Albums/Honey Bee`.

Example: Folder Data Structure

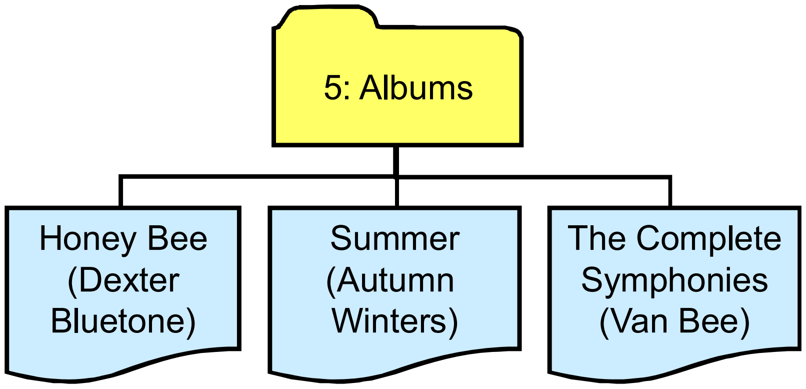
id	folderName	parent Folder
1	/	
2	Articles	1
3	Reviews	2
...	...	...
6	Concerts	3



The example in the slide shows the folder structure as it appears in the database:

- There is a “root” folder “`/`” at the top with an ID of `1`.
- The `Articles` folder has an ID of `2` and its parent folder is `/` as represented by the entry of `1` (the parent folder’s ID) in the `parentFolder` column.
- The `Features` folder (ID `4`) and `Reviews` folder (ID `3`) both have the `Articles` folder (ID `2`) as the parent folder.
- And so on throughout the structure

Example: Content Data Structure


5: Albums
Honey Bee (Dexter Bluetone)
Summer (Autumn Winters)
The Complete Symphonies (Van Bee)

id	title	folder	articleName	author
100	Honey Bee (Dexter Bluetone)	5	honeyBee.html	A. Critic
101	Summer (Autumn Winters)	5	summer.html	I.M. Fan
102	The Complete Symphonies (Van Bee)	5	symVanBee.html	A. Critic

The example in the slide shows the data structure for the three album reviews. The columns include the ID for the article item, the title, the folder (ID `5`, which is `Albums`), the name of the article as an HTML page, and the author.

Uses of Hierarchical Content Repositories

- Advantages of a SQL content repository:
  - Ease of organization for content maintainers
  - Hierarchical targeting
- Features:
  - Content Repository API
  - Repository Loader to import existing file-based content
- Hierarchical repositories are used extensively in ATG Content Administration and ATG Commerce.

- **Ease of organization:** When dealing with hundreds, thousands, or even more content items, it can be difficult for developers, business users, and administrators to find and manage them. For example, if you have a product catalog with several image-file content items for each product, it can be helpful to organize those images according to product type.
- **Hierarchical targeting:** Storing content hierarchically enables you to treat the "location" of a content item as a targetable attribute (for example, "show user all items in the Book Reviews" folder).
- **Content Repository API:** Using content and folder items enables you to use a convenient API to get a content item's path, a folder's contents, and so on.
- **Repository Loader:** The Repository Loader enables you to load content from a file system into a SQL repository. (This is covered in the *Establishing the ATG Content Administration Environment* course.)

Summary

In this lesson, you should have learned how to:

- Create custom property types that allow the programmatic manipulation of repository item properties
- Describe how hierarchical content repositories mirror a file system repository

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Repository Guide*
  - “SQL Repository Item Properties”
    - “User-Defined Property Types”
  - “SQL Content Repositories”
- Appendix A: Content Repositories (in this Student Guide)

Practice 6 Overview: Creating a Custom Property

This practice covers creating a custom property type that calculates a user's age from the user's birthday.

7

Derived Properties

Objectives

After completing this lesson, you should be able to implement derived properties.

Deriving Property Values

A *derived property* is a transient property (not associated with a column in a table) that:

- Derives its value from another property
- Can search through a series of properties to find a non-null value
- Specifies the derivation mechanism in the XML Repository Definition File by using a `<derivation>` tag

Derived properties provide a way for you to point toward different property values within the same object to derive a default value when an initial value is not available for that particular property.

You can look at property values in the same item descriptor that the current values reside in, or at subproperty values within the same object.

Any value that is accessible to that object is available for derivation, and no code needs to be written to enable this flexibility. This provides you with the maximum amount of flexibility in obtaining values for properties with a minimal amount of extra coding effort.

`<derivation>` Tag

- The `<derivation>` tag enables you to specify one or more properties from which to derive the current property's value.
  - This tag uses `<expression>` tags to list the properties to look through for a value.
  - Both the derived property and the expressions must be of the same type.
- By default, the repository returns the value of the first property in the list that is not null.
- Anything more complex requires a custom property (as covered in an earlier lesson).

The default method for deriving a property value is `firstNonNull`, as described in the slide. You learn about other derivation methods later in the lesson.

Dynamusic and Derived Properties

- Registered Dynamusic users have a quota on the number of MP3 songs they can download, which varies for different types of members (Dynamusic member or fan club member).
- The value for the quota may come from one of the following:
  - The initial quota set when the Dynamusic member registered
  - The quota set manually by Dynamusic
  - The default quota for the fan club that the user is associated with as a fan club member
- Users and programmers of the application should not need to query each of these quota sources when they need to determine a user's download limit.

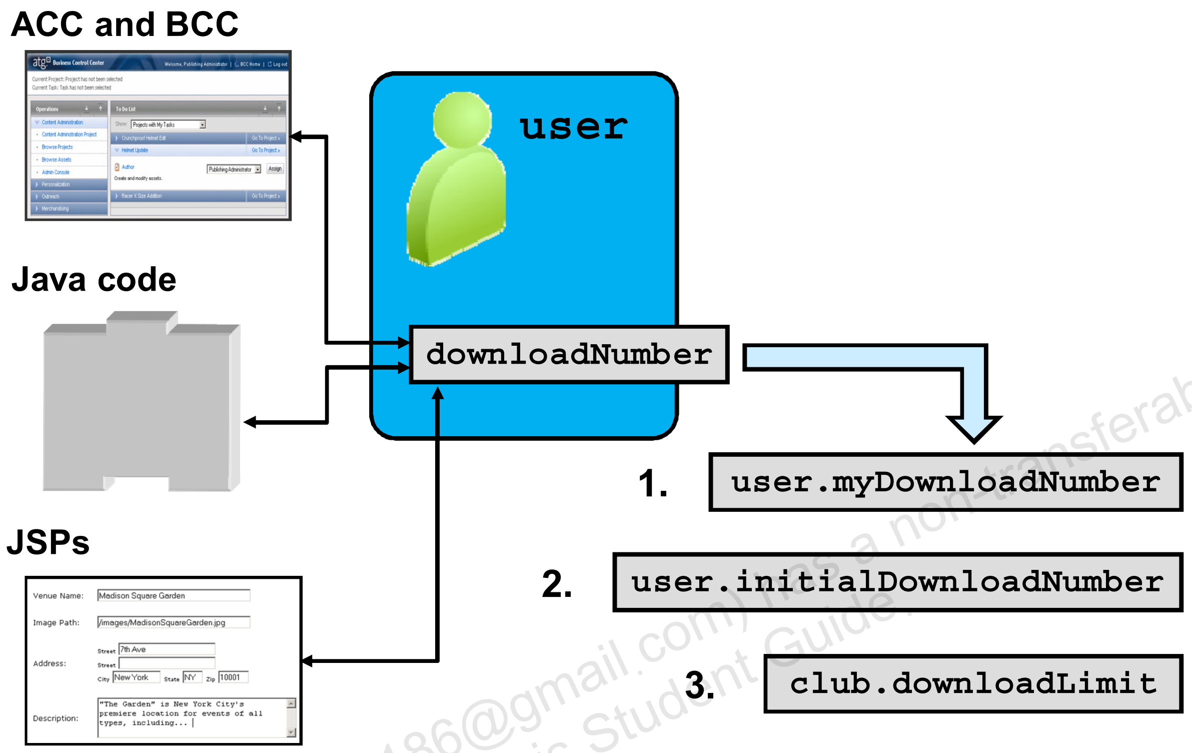
The examples in this lesson make some assumptions that you need to be aware of. They assume that there are two types of registered users with Dynamusic:

- Users who register directly with the site (Dynamusic member)
- Users who register for the site through a fan club

Users who register through Dynamusic are given an initial quota through a profile property. Users who register for a fan club and become members of Dynamusic through their association with that fan club do not have the initial quota property set. Instead, their quota is derived from the quota for the fan club.

The examples also assume an underlying architecture that differentiates between the two types of users. For example, a Dynamusic user may have a `member` property set to true, and a scenario then fills the initial quota based on that setting.

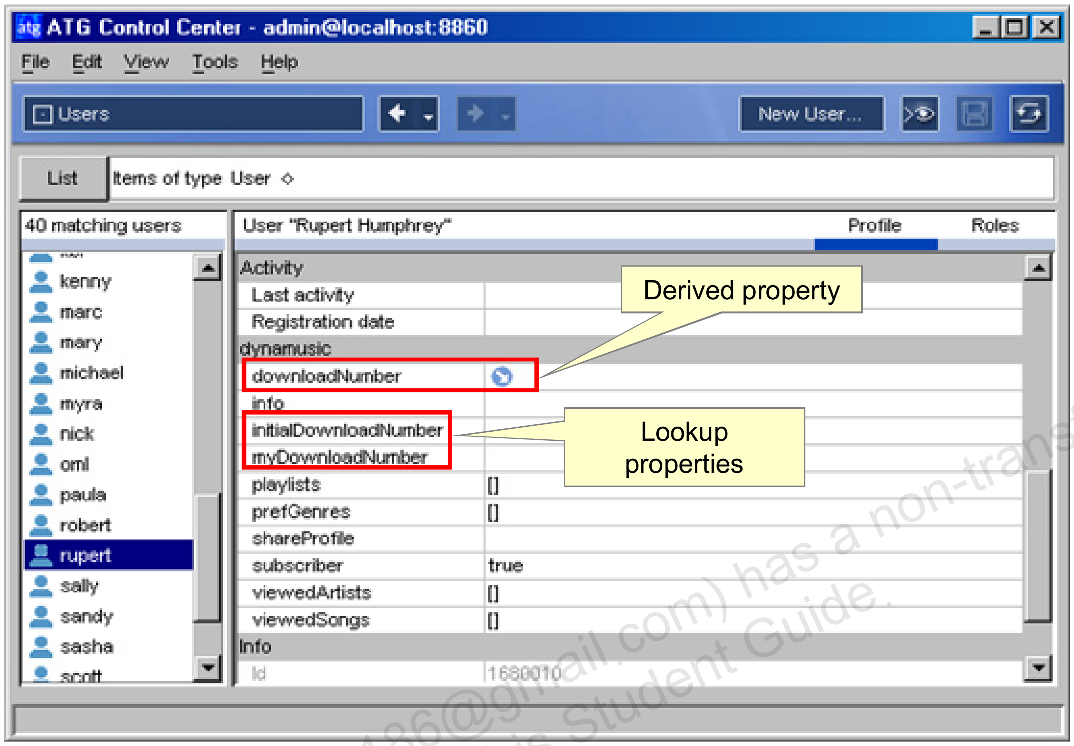
Dynamusic and Derived Properties: Example



The purpose of derived properties is to shield users and programmers from having to look in multiple spots for a value. In the example in the slide, the `downloadNumber` property is the one that is accessed by, for example, the Java code, JSPs, the ACC and the ATG Business Control Center (BCC). This property looks through one of three other properties for a value.

First, it looks at the user's `myDownloadNumber` to see if there is a value. If not, it proceeds to the user's `initialDownloadNumber`. If that is also not set, it looks at the `downloadLimit` for the user's club.

Dynamusic Derived Property Extensions



The following XML was added to the `userProfile.xml` file to support Dynamusic's derived properties:

```
<item-descriptor name="user" >
    <table name="dynamusic_user" type="auxiliary"
         id-column-name="user_id">
         <property name="myDownloadNumber" data-type="int"
              category="dynamusic"
              column-name="my_download_number"/>
         <property name="initialDownloadNumber" data-type="int"
              category="dynamusic"
              column-name="initial_download_number"/>
    </table>
</item-descriptor>
```

A Simple Property Derivation in `userProfile.xml`

```
<item-descriptor name="user">
   <table name="dynamusic_user" type="auxiliary" id-
     column-name="user_id">
   ...
     <property name="initialDownloadNumber" data-
 1    type="int"/>
     <property name="myDownloadNumber" data-type="int"/>
   </table>

   <property name="downloadNumber" data-type="string"
      writable="false">        2
     <derivation>
    3    <expression>myDownloadNumber</expression>
         <expression>initialDownloadNumber</expression>
     </derivation>
   </property>
</item-descriptor>
```

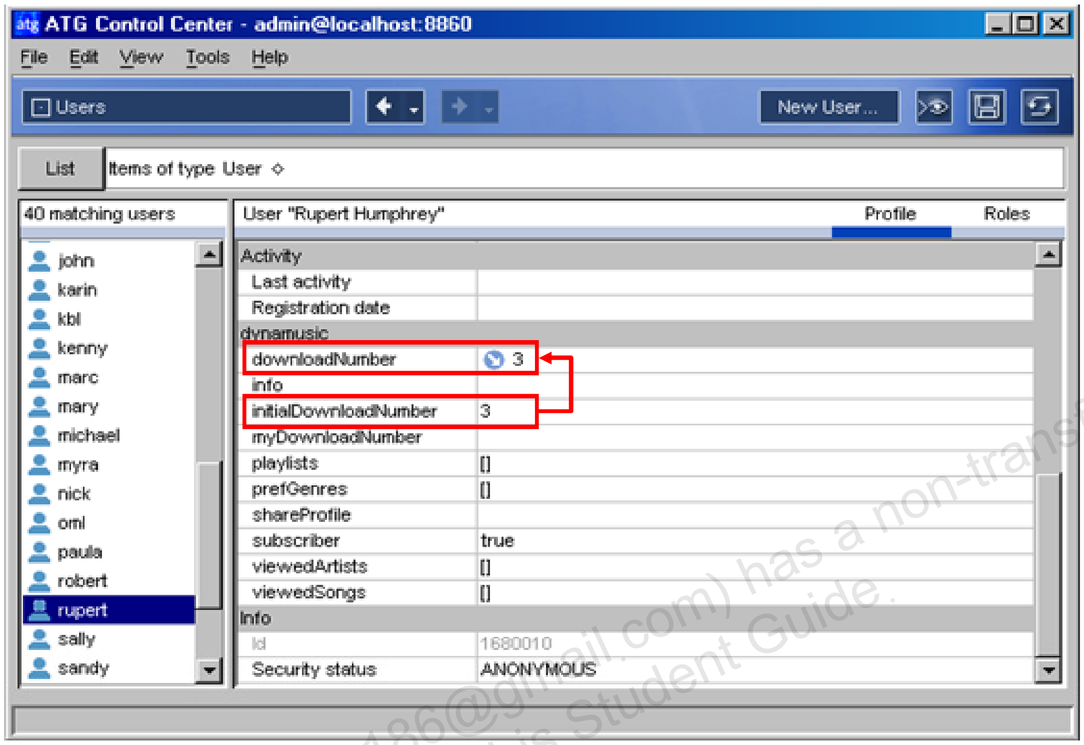
ORACLE

In the example in the slide, one item property is derived from another.

1. The `myDownloadNumber` and `initialDownloadNumber` properties are nonderived properties that are set explicitly.
2. The `downloadNumber` property is transient and is nonwritable because its value is derived (rather than set explicitly).
3. When it is requested, the value of `myDownloadNumber` is returned if it is non-null; otherwise, `initialDownloadNumber` is returned.

Note: In the example, the `column-name` attribute is missing for each property. These attributes were omitted for space reasons. They can be found in the code example in `<coursedir>/examples/lesson07/userProfile.xml`.

Result: Simple Property Derivation



Activity	
Last activity	
Registration date	
dynamusic	
downloadNumber	3
info	
initialDownloadNumber	3
myDownloadNumber	
playlists	[]
prefGenres	[]
shareProfile	
subscriber	true
viewedArtists	[]
viewedSongs	[]
Info	
Id	1680010
Security status	ANONYMOUS

In the example in the slide from the ACC, Rupert's `initialDownloadNumber` is set to `3`. The `downloadNumber` property derives its value from `initialDownloadNumber`. The blue circle with the arrow symbol shows that the value of `downloadNumber` is derived.

Deriving from a Related Item in `userProfile.xml`

```
<item-descriptor name="fanClub">

   <table name="dynamusic_fanclub" type="primary"
     id-column-name="id">

     <property name="downloadLimit" data-type="int"
      column-name="d_l"/>

   </table>

</item-descriptor>
...
```

In the example in the slide, there is now a `downloadLimit` defined for each fan club on the site (as defined in the `fanClub` item descriptor). If the user belongs to a fan club that has this value set, the `downloadNumber` can be derived from that club's `downloadLimit`.

Dynamusic gives memberships to members of fan clubs to increase the Dynamusic customer base; such members can be associated with their clubs. Of course, you might want to use the User Directory to define this association instead (this topic is discussed later in the lesson).

In the simple example in this slide and the next slide, you are deriving the property only from nonderived properties, but you can also derive properties from other derived properties. The derivation tree can be as complex as necessary.

Deriving from a Related Item in `userProfile.xml`

```
<item-descriptor name="user" >
   <table name="dynamusic_user" type="auxiliary" id-
     column-name="user_id">
     <property name="initialDownloadNumber"
         data-type="int"/>
     <property name="myDownloadNumber" data-type="int"/>
     <property name="club" item-type="fanClub"/>
   </table>
   <property name="downloadNumber" data-type="int"
     writable="false">
     <derivation>
         <expression>myDownloadNumber</expression>
         <expression>initialDownloadNumber</expression>
         <expression>club.downloadLimit</expression>
     </derivation>
   </property>
</item-descriptor>
```

When the `downloadNumber` property is queried, the repository first checks `myDownloadNumber` and `initialDownloadNumber`. If both are null, it returns `downloadLimit` that is associated with the member's club.

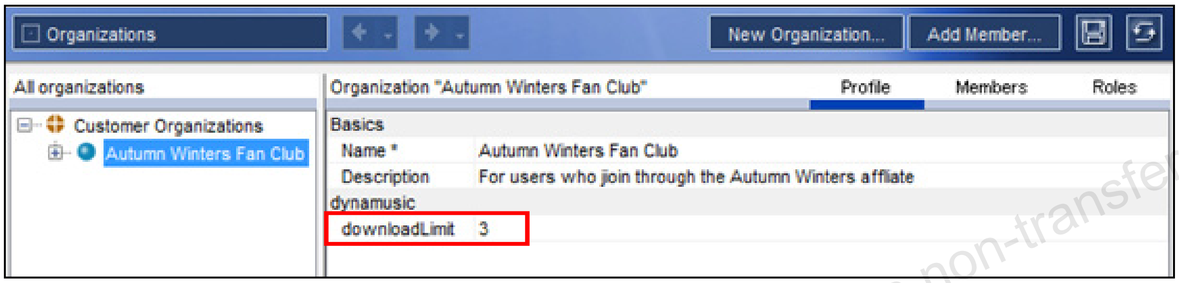
Property Derivation and Organizations

- The Dynamo User Directory enables you to group users according to the organization to which they belong:
  - Useful for B2B and intranet applications
- Individual user properties are often derived from organization properties.
- Here is a Dynamusic organization example:
  - Users may join through a fan club organization.

The Dynamo User Directory and the features that it includes, such as organizations and roles, are covered in detail in the *ATG Personalization and Scenario Development* course. The organization feature is mentioned here only as it relates to derived properties. For more information, see the “Working with the Dynamo User Directory” chapter in the *ATG Personalization Programming Guide*.

Deriving from Organizations

To demonstrate derivation from an organization, the organization has been extended to include a new property, `downloadLimit`.



Basics

Name *	Autumn Winters Fan Club
Description	For users who join through the Autumn Winters affliate

dynamusic

downloadLimit	3

You can also use derivation with the User Directory. Typically, users derive a value from the organization to which they belong. Deriving from an organization is very similar to deriving from a related item. In this case, the related item is the organization item. Dynamusic has extended the organization item in `userProfile.xml` to include a new property called `downloadLimit`. The XML to do this is as follows:

```
<item-descriptor name="organization">
    <table name="dynamusic_fanclub" type="auxiliary"
         id-column-name="organization_id">
         <property name="downloadLimit" data-type="int"
              column-name="download_limit" category="dynamusic"/>
    </table>
</item-descriptor>
```

Deriving from the Organization in `userProfile.xml`

```
<item-descriptor name="user">
    <table name="dynamusic_user" type="auxiliary"
      id-column-name="user_id">
      ...
      <property name="initialDownloadNumber"
          data-type="int"/>
      <property name="myDownloadNumber" data-type="int"/>
    </table>

    <property name="downloadNumber" data-type="int"
      writable="false">
      <derivation>
          <expression>myDownloadNumber</expression>
          <expression>initialDownloadNumber</expression>
          <expression>parentOrganization.downloadLimit
             </expression>
      </derivation>
    </property>
</item-descriptor>
```

If no download limit has been set for this user (in either `myDownloadNumber` or `initialDownloadNumber`), the code looks at the `downloadLimit` property of the organization to which this user belongs.

Overriding Property Derivation

- What if you need to set the value of a derived property?
  - An exception to the derivation rules
- Overriding property derivation allows a property value to be explicitly set rather than derived.
  - This bypasses property derivation logic if the override value is set.

When do you use a derived property override?

If a user subscribes to Dynamusic, you may want to raise the download quota of that user. You may also want to give your customer support representatives the ability to raise the download quota of any given user, perhaps to rectify a software glitch or to soothe an angry user.

If the value is overridden, the derivation properties are ignored and the entered value takes precedence.

Overriding Property Derivation in `userProfile.xml`

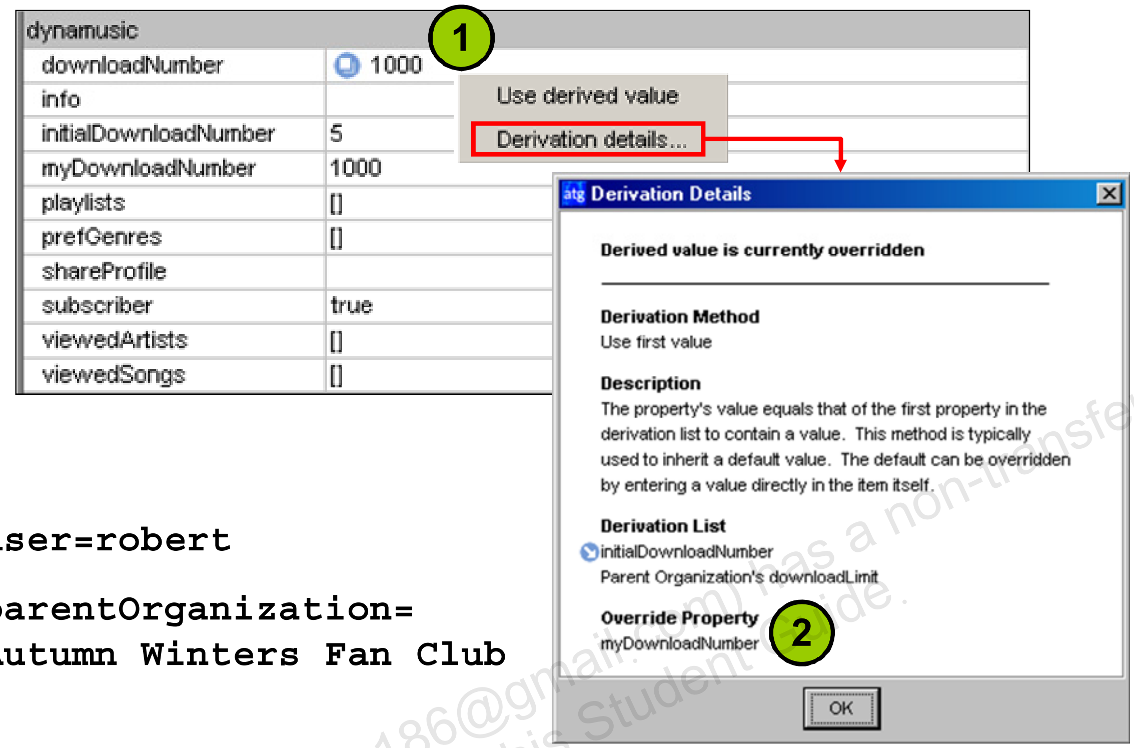
```
<item-descriptor name="user">
    <table name="dynamusic_user" type="auxiliary"
      id-column-name="user_id">
      ...
      <property name="myDownloadNumber" data-type="int"/>
    </table>

    <property name="downloadNumber" data-type="int"
      writable="true">
      <derivation override-property="myDownloadNumber">
           <expression>initialDownloadNumber</expression>
           <expression>parentOrganization.downloadLimit
              </expression>
      </derivation>
    </property>
</item-descriptor>
```

This example adds an "override" property that allows the user's download number to be set explicitly.

Because there is an override property, this property is now considered "writable"; if `downloadNumber` is set, the value will be stored in `myDownloadNumber`.

Because this property's override property has been specified, the repository first looks to `myDownloadNumber` when the property value is requested. If that value is non-null, it is returned. If it is null, the value of `initialDownloadNumber` is requested and, if non-null, is returned. Otherwise, the value of the user's parent organization's `downloadLimit` is returned.

Result: Overriding Property Derivation in `userProfile.xml`



dynamusic	
downloadNumber	1000
info	
initialDownloadNumber	5
myDownloadNumber	1000
playlists	[]
prefGenres	[]
shareProfile	
subscriber	true
viewedArtists	[]
viewedSongs	[]

Derived value is currently overridden

Derivation Method
Use first value

Description
The property's value equals that of the first property in the derivation list to contain a value. This method is typically used to inherit a default value. The default can be overridden by entering a value directly in the item itself.

Derivation List
initialDownloadNumber
Parent Organization's downloadLimit

Override Property
myDownloadNumber

```
user=robert

parentOrganization=
Autumn Winters Fan Club
```

1. The screenshot in the slide from the ACC shows the user Robert Zimmerman, the president of the Autumn Winters Fan Club. You may recall that the `downloadLimit` property for that fan club was set to 3. In Robert's case, he has his `initialDownloadNumber` set to 5. However, because he is a subscriber to Dynamusic, his `myDownloadNumber` property has also been set to a value of 1000. The circle with a square next to the `downloadNumber` value indicates that this property's value is overridden. Because of the derivation rules, `downloadNumber` derives its value from `myDownloadNumber` (the override property).
2. Clicking the symbol next to the derived property enables you to bring up a window showing you the method and derivation source of the current value. If you click the symbol, you can also select "Use derived value" to remove the override.

Derivation Methods

ATG provides six different derivation methods:

- `firstNonNull` (default; used most of the time)
- `firstWithAttribute`
- `firstWithLocalefirstWithAttribute`
- `alias`
- `union`
- `collectiveUnion`

```
<derivation method="derivation method">
<expression>…</expression>
<expression>…</expression>
</derivation>
```

By default, the SQL repository derives a property by traversing the expressions in order, starting with the property itself. The first non-null value found is used as the property value. This is the `firstNonNull` method. You do not have to specify this method.

The `firstWithAttribute` method requires you to specify an attribute named `derivationAttribute`. The code iterates through the expressions in order and uses the first property with an attribute that matches the value of the `derivationAttribute`. You specify a `defaultKey` property as well. If the value of the real key is null, the value of the `defaultKey` is used.

The `firstWithLocale` method is a subclass of `firstWithAttribute`. It performs the following actions:

1. Gets the user's current locale as the key from the Nucleus component defined by a `keyService` attribute
2. Compares this locale to each expression's `locale` value
3. Returns the first property whose attribute matches

The locale is searched in a locale-specific way. For example, if `locale=fr_FR_EURO`, it first looks for a property where the locale attribute is `fr_FR_EURO`, then looks for `fr_FR`, and finally looks for `fr`.

The `alias` derivation method lets you define an alternate name for a repository item property and use either name to access the property. This can be useful in a situation where different application modules use the same property but want to use different names for the property. A single alternate name can be defined in an `<expression>` element within a `<derivation>` element.

The `union` derivation method enables the combination of several properties of a repository item into a single property. The class takes two or more set or list type properties and then combines the values of those properties in the current repository item to create the new derived property.

The `collectiveUnion` derivation method enables a property to be derived from a union of subproperties. The expression element is a property in the item descriptor that represents a collection of values. The derived property returns a union of the properties indicated by the value of the `collectionProperty` attribute.

firstWithAttribute: Example

```
<item-desciptor name="myItem">
 <property name="name">
  <derivation method="firstWithAttribute">
   <expression>englishName</expression>
   <expression>icelandicName</expression>
  </derivation>
  <attribute name="derivationAttribute"
             value="language"/>
  <attribute name="defaultKey" value="en"/>
 </property>
 <property name="englishName">
  <attribute name="language" value="en"/>
 </property>
 <property name="icelandicName">
  <attribute name="language" value="is"/>
 </property>
</item-descriptor>
```

The example in the slide shows how to define a derived property using the `firstWithAttribute` method. You specify the correct derivation method to use for the property in the `derivation` tag with the `method` attribute. You define your expressions. For `firstWithAttribute`, you must specify the attribute that is being derived. In the example in the slide, the `derivationAttribute` is set to language. The `defaultKey` is `en`, which means that `en` is used as the language if both expressions return null. Note that the properties in the expressions are also set with the attribute name language and the value for the correct language.

Derived Properties in ATG

- Derived properties are used in several ways in out-of-the-box ATG repositories.
- Here is an ATG Business Commerce example:
  - The default values for a user’s shipping and billing address are derived from the shipping and billing address of the user’s organization.
  - Addresses can be set explicitly to override the derived organizational address.

Derived properties are available in the ATG Dynamo Application Framework. Many people may want to use this functionality in connection with organizations and users, where the user derives certain values from the organization. This functionality is used extensively in ATG Business Commerce.

For more information about derived properties, refer to the section titled “Derived Properties” in the “SQL Repository Data Models” chapter of the *ATG Repository Guide*.

Quiz

Which of the following statements describe a derived property? Select all that apply.

a. The property always has `writable="false"`.

b. You can override the property derivation by using the `override-property` attribute.

c. The most common derivation method is `firstWithAttribute`.

d. You can derive from subproperties as well as properties in the same item.

Answer: b, d

Answer a: When you set an override property, the derived property is considered “writable” and has `writable` set to `true`.

Answer c: The most common derivation method (and the default) is `firstNonNull`.

Summary

In this lesson, you should have learned how to:

- Create a derived property from a list of properties
- Allow users to explicitly override a derived property's value

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Repository Guide*
  - "SQL Repository Data Models"
    - "Derived Properties"

Practice 7 Overview: Creating a Derived Property

This practice covers the following topics:

- Creating a derived property to set an album's genre from its artist's `artistGenre`
- Creating a new enumerated property for the album item type called `albumGenre`, and then changing the genre's definition to make `albumGenre` the override property

8

Repository Caching

ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe how SQL repository caching works
- Configure caching in a SQL repository
- View caching statistics

Road Map

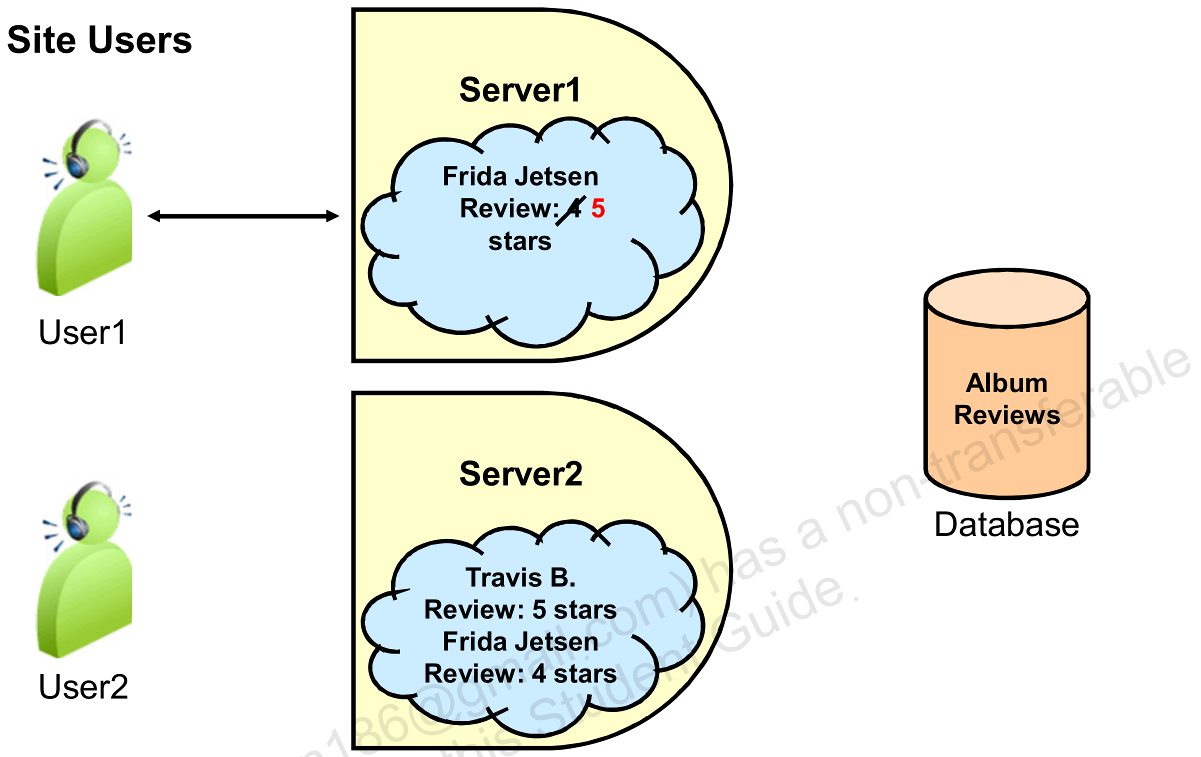
- Caching architecture
- Cache modes
- External caching

ORACLE

Caching Architecture

- Each ATG server in a cluster maintains its own caches.
- Caches are completely independent from each other and reflect the queries and items that are retrieved on that server.
- Caches may or may not be synchronized when changes are made, depending on the cache mode configured.

Caching Architecture: *Externally* Managed Content

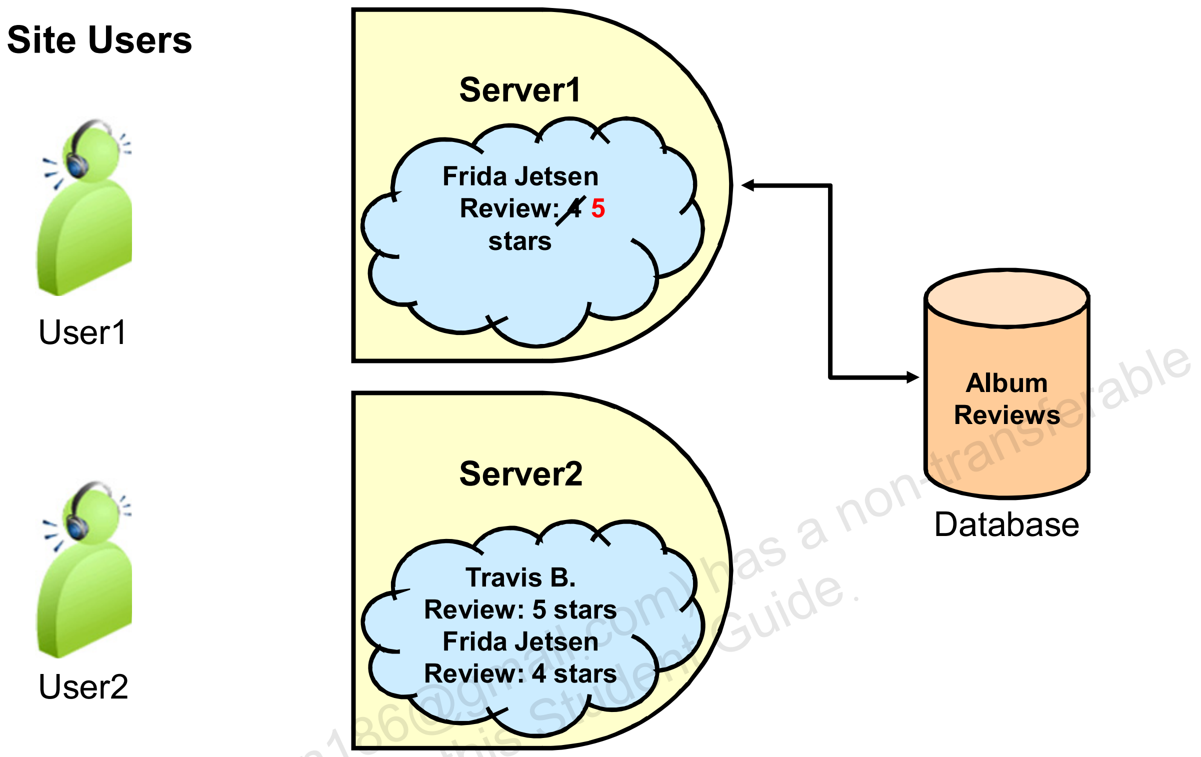


Each server in a cluster maintains its own cache of content requested by users on that server.

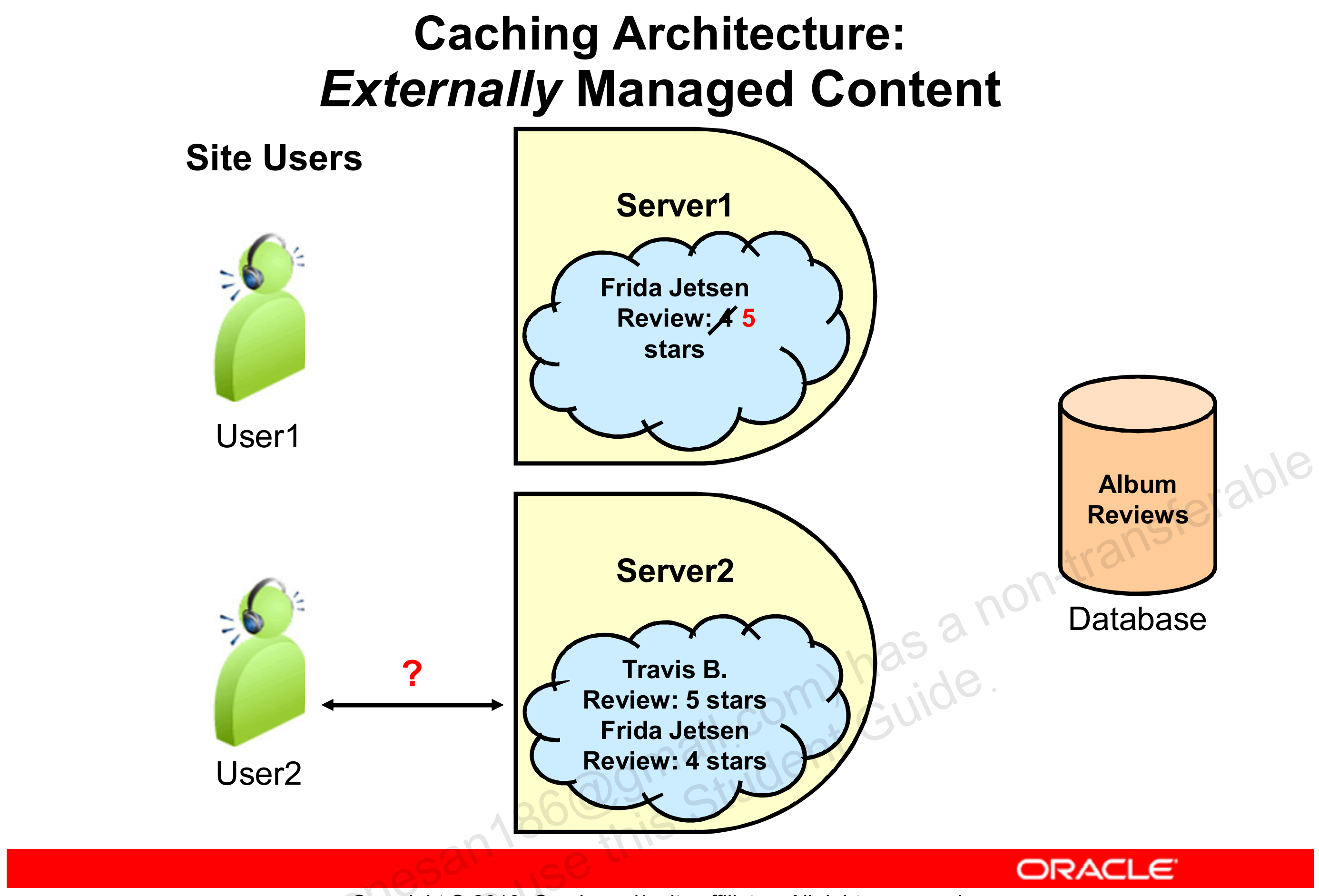
In the case of externally managed content, such as users' reviews of albums, the content is changed through a user session on one of these servers. This change is written to the common database but may or may not be reflected in the other server caches, depending on the cache mode.

In the example in the slide, album reviews are submitted and maintained by Dynamusic's website users. User1, who submitted the Frida Jetsen review for 4 stars, has decided to change it to 5 stars. User1's session is on Server1, so the data in Server1's cache is updated. This review has also been cached on Server2.

Caching Architecture: *Externally* Managed Content



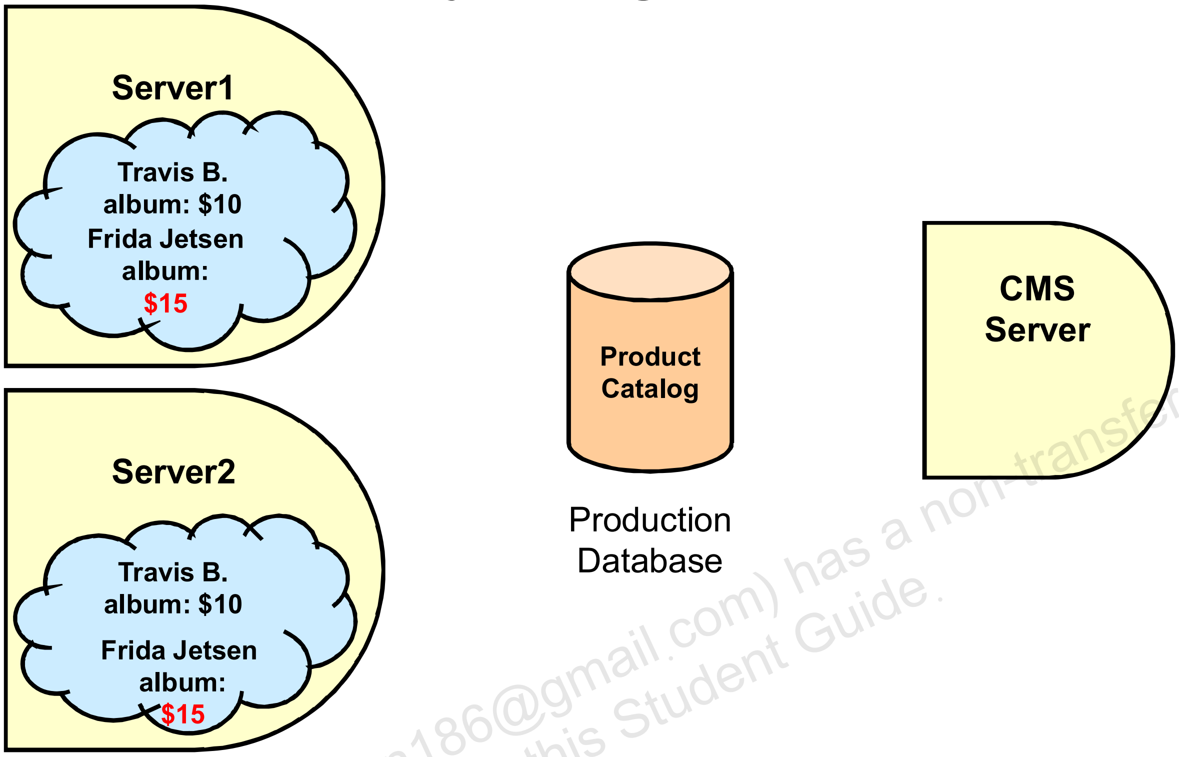
The example continues from the previous slide. The changed review is saved to the underlying database.



Caching Architecture: *Externally* Managed Content

The example continues from the previous slide. Whether and when the data is updated in Server2's cache—and therefore which version User2 sees—depends on the cache mode that is selected. The available cache modes are covered later in this lesson.

Caching Architecture: *Internally* Managed Content

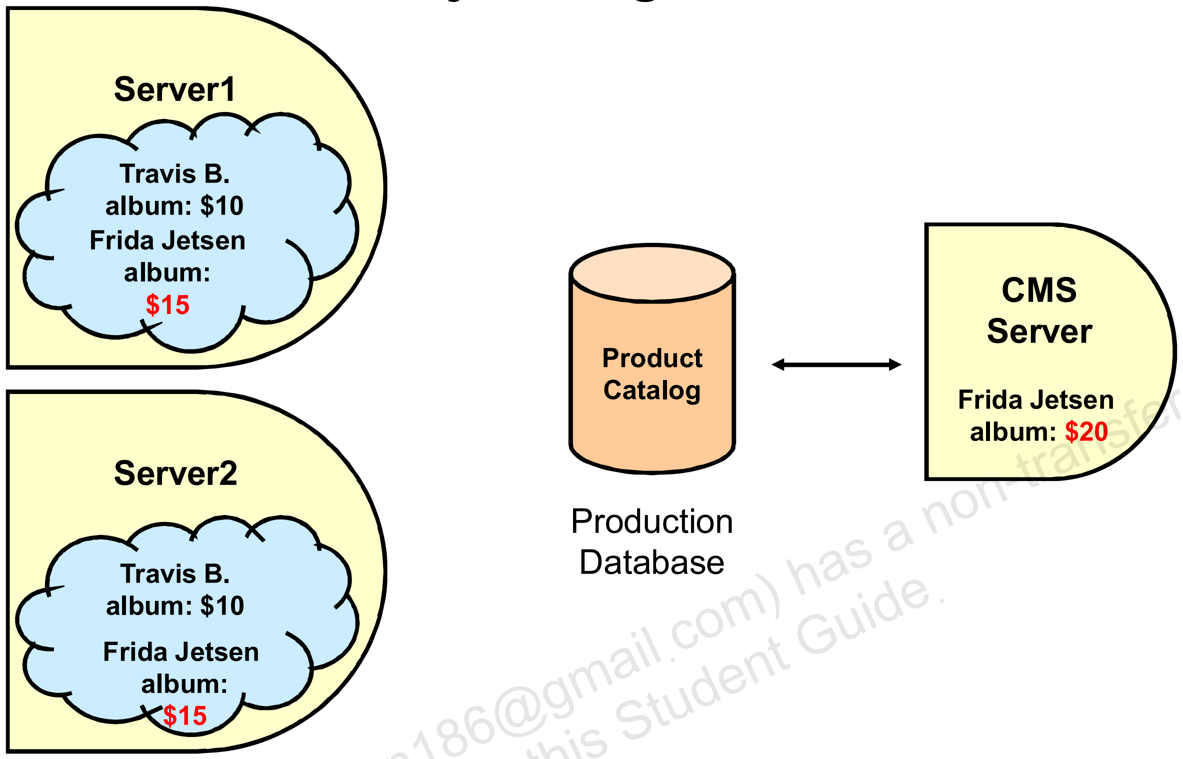


The example in the slide shows how caching is handled with internally managed content. Ideally, internally managed content, such as the product catalog, should be changed by using a content management system (CMS). ATG's CMS product is called ATG Content Administration (CA). CA is covered in more detail in *Managing Your ATG Commerce Solution* and *Establishing the ATG Content Administration Environment*.

Changes made from the CMS will likely be sent directly to the database used by the production cluster. The production servers should then be notified that a change has occurred. ATG Content Administration does this automatically.

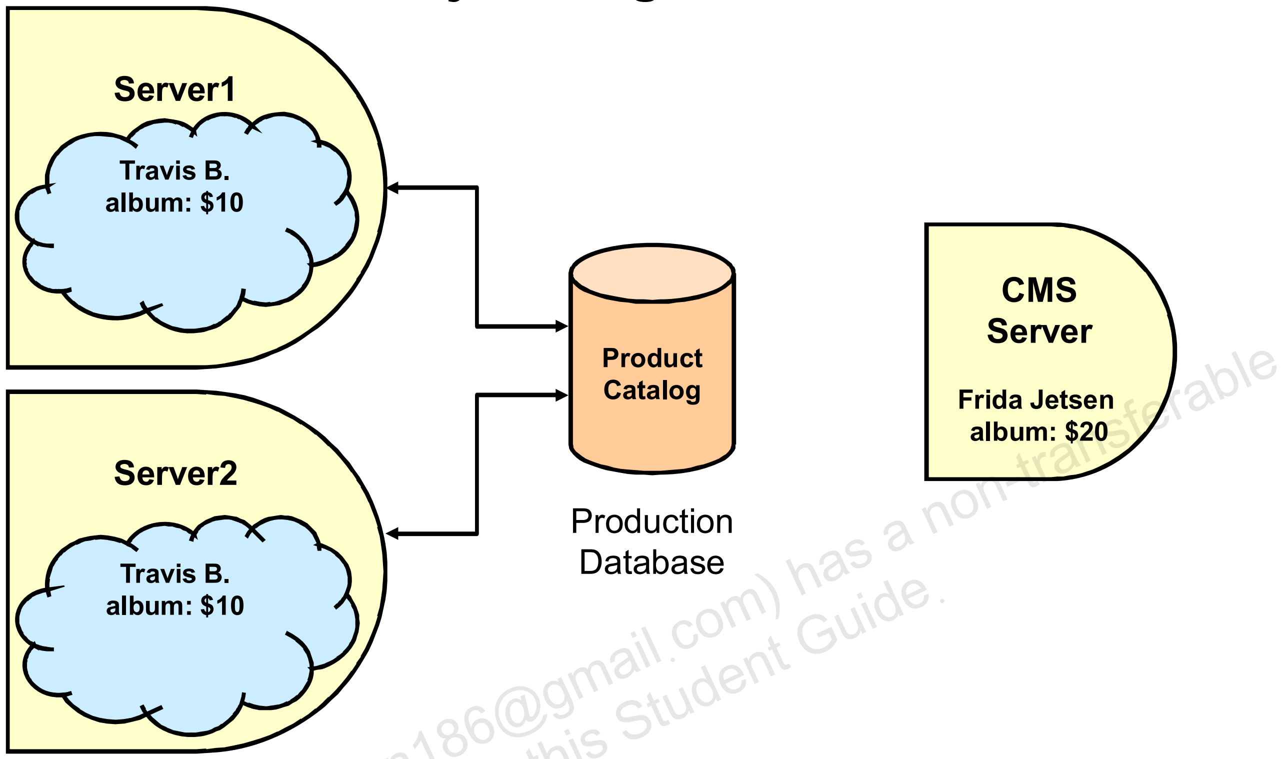
In the example in the slide, the price for the Frida Jetsen album in the cache for both Server1 and Server2 is $15. The next two slides show what happens when the price is changed.

Caching Architecture: *Internally* Managed Content



In the example in the slide, a merchandiser has updated the price of a Frida Jetsen album from $15 to $20 by using the CMS server. That change has been sent to the production database. The old price of $15 is cached by Server1 and Server2.

Caching Architecture: *Internally* Managed Content



When the production servers receive the notification that the cached data is out of date, they delete the affected item and query caches. The next time that information is needed, it is read from the database. In the example in the slide, the Frida Jetson album's price is no longer in the cache on either server.

Repository Queries

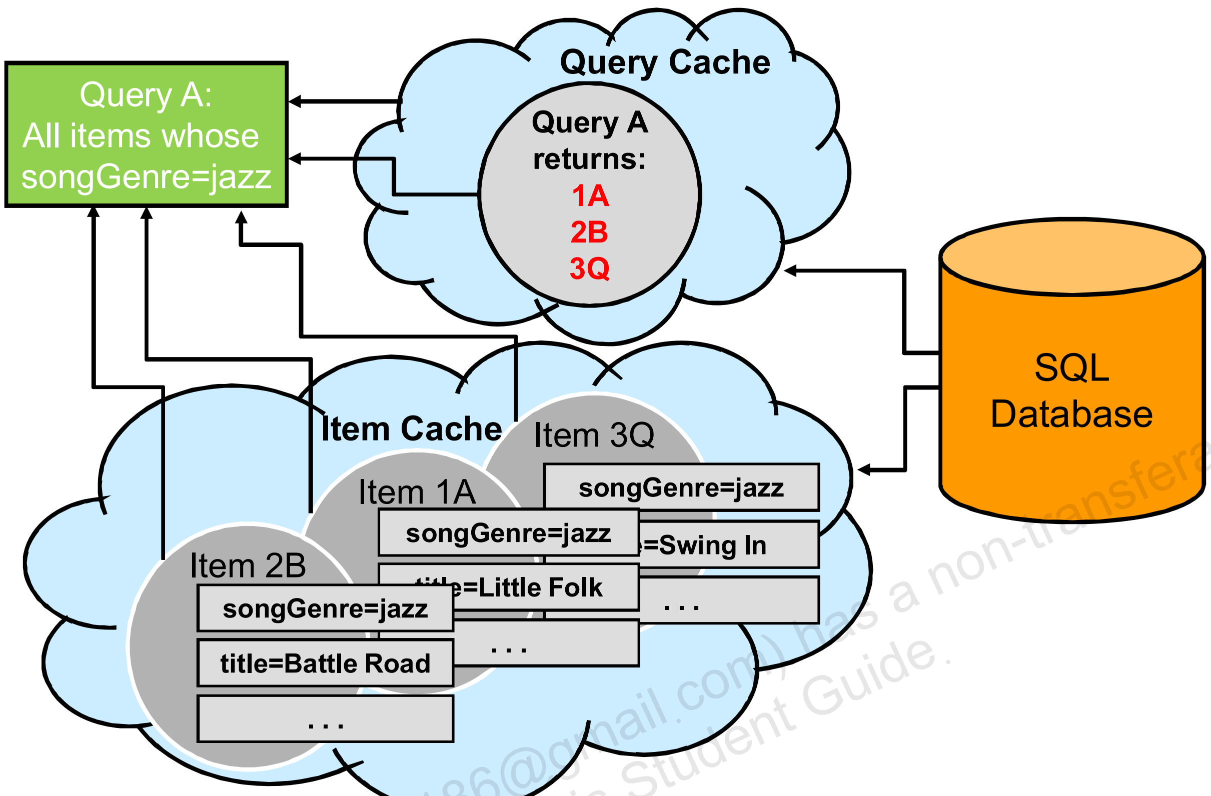
Repository queries are performed in two phases.

1. The first phase retrieves IDs of items that fulfill the query.
   – Example: What are the IDs of the songs with a genre of “jazz?”
   – The list of item IDs for a particular query may be cached in the query cache.
2. The second phase queries for the properties of selected items.
   – Item properties may be cached in the item cache.

Suppose that an item has a property that is itself a repository item or a list of repository items (such as an album's `artists` property). By default, when that item is cached, the properties of the related items are also cached. Alternatively, the child items can be set to be cached by ID by using the `cacheReferencesById` property:

```
<item name=album>
    ...
    <property name="artists" ...>
         <attribute name="cacheReferencesById" value="true"/>
    </property>
</item>
```

Query and Item Caches



The item cache stores individual items indexed by item ID. The query cache stores the IDs of the set of items returned by a particular query. Individual items are retrieved from the item cache, if possible.

In the example in the slide, query A looks for all items whose `songGenre` property is `jazz`. The item cache contains three items with `songGenre` equal to `jazz`. The query cache returns those three items.

Query Cache Invalidation

Whenever a repository item is changed, its *item* cache entry is invalidated. But how is the *query* cache invalidated? The SQL repository keeps track of the properties that are used in each query. When any of these properties are modified, the affected queries are invalidated. Also, if an item is added or removed, all queries are invalidated.

Item Cache Configuration

The item cache options are:

- `cache-mode`
- `item-cache-size` (default = 1000)
- `item-expire-timeout` (default = −1)
- `item-cache-timeout` (default = −1)

The possible values for `cache-mode` are `disabled`, `simple`, `locked`, and `distributed`.

`item-cache-size`: Number of items that can be stored in cache memory. Note that the cache size can be set only by number of items and not by amount of memory. When the cache is full, the least recently used item is removed.

`item-expire-timeout`: Amount of time (in milliseconds) that an item can stay in the cache, unused, before it is refreshed. After the time expires, ATG queries the database for fresh information about that item. The default is −1, which means that the item stays in the cache indefinitely without being refreshed.

`item-cache-timeout`: Amount of time (in milliseconds) that an item can stay in the cache, unused, before it is removed. After the time expires, ATG deletes that item from the cache. It is read from the database the next time it is needed by the application. The default is −1, which means that the item stays in the cache indefinitely.

Query Cache Configuration

- The query cache options are:
  - `query-cache-size` (default = 0)
  - `query-expire-timeout` (default = −1)
- The following is an example of item and query cache configuration for the user item:

```
<item-descriptor name="user"
      cache-mode="simple"
      item-cache-timeout="550000"
      query-expire-timeout="550000" >
```

`query-cache-size`: Number of queries that can be stored in cache memory. Note that the default is 0, which means that unless you set this to a higher number, query caching is disabled by default. Enabling query caching is most appropriate for data that does not change much, such as a product catalog. For data that changes frequently, such as profile data, the cache would be invalidated so often that it would often not be worth the caching overhead. It also does not make much sense to cache queries that are rarely, if ever, repeated.

`query-expire-timeout`: Amount of time (in milliseconds) that a query can stay in the cache, unused, before it is refreshed. After the time expires, ATG queries the database for fresh results for that query. The default is −1, which means that the query stays in the cache indefinitely without being refreshed.

The code example in the slide shows how to set caching on the item level. For the user item, it sets the cache mode to simple caching. Items and queries are both set to be deleted from the cache if they remain unused for 550,000 milliseconds. You learn more about the different cache modes in the next section.

Road Map

- Caching architecture
- Cache modes
- External caching

ORACLE

Cache Modes

- There are five cache modes for repository items:
  - Disabled
  - Simple (default)
  - Locked
  - Distributed (TCP or JMS)
  - Distributed hybrid
- The cache mode is set at the item level (for example, `album`).

```
<item-descriptor name="item-name" cache-mode="cache-mode-name">
```

- Disabled caching can be set at the property level (for example, `album.coverURL`).

Subsequent slides describe the different cache modes.

Cache Mode: Disabled

- The cache mode name is `disabled`.
- Items are not cached. Each query accesses the database.
- Suggested use: When data is changed frequently from outside the ATG application
- Alternative: Simple caching with timeouts

Caching should be disabled when it is critical for the data to be accurate, and when data might change in the database directly (not through the Repository API). For instance, if you have an online banking application and the same data is accessed by other applications in addition to ATG, you would want to turn off caching for displaying the user's account balances.

This mode should be used with great caution because it results in database access for every page that accesses an item of this type. This has a potentially severe impact on performance.

The other caching modes can be set only on a per-item-type basis, but "no caching" mode may be set on a per-property basis. If a request is made for a "no cache" property of a cached item, the database is queried.

Here is an example from `userProfile.xml`:

```
<property category="Login" name="password"
    data-type="string" required="true"
    column-name="password" cache-mode="disabled" >
```

Cache Mode: Simple

- The cache mode name is `simple`.
- No attempt is made to keep cache items in sync between servers.
- If one server modifies an item, other servers will have inaccurate data in the cache until items are refreshed or removed.
- Suggested use: When data is predominantly read-only

This mode is recommended for items that are read frequently but modified rarely, such as product catalog items or press release archives. It is also recommended for an environment with a single ATG server.

Note that simple caching is the default. Sites running multiple ATG servers will want to change the cache mode on important item types.

Choosing a Cache Mode

In most cases, the appropriate cache mode is either simple or locked. Distributed cache mode is used more rarely.

Cache Mode: Locked

- The cache mode name is `locked`.
- Only one server can modify an item at a time.
  - The server that wants to modify an item must obtain write lock.
- When write lock is granted, the item is cleared from other servers' caches and may not be read again until the lock is released.
- One ATG server acts as a lock manager.
  - If all lock managers fail, caching becomes disabled.
- Suggested use: When data is changed frequently
- Disadvantage: Can lead to deadlocks

Note: If you use locked caching for an item, you must disable query caching for that item because the lock manager does not manage locks on cached queries. This is done by setting the `query-cache-size` attribute on the item to `0`. This is the default setting for an item, so it needs to be changed only if you configured the query cache previously.

Locked caching is most useful for items that are modified frequently.

Locked caching is based on write locks and read locks. If no servers have a write lock for an item, any number of servers may have a read lock on that item. When a server requests a write lock, all other servers are instructed to release their read locks. Once an item is write locked, no other server may get a read lock or write lock until the first server releases its write lock. In other words, after a server has a write lock on an item, all access to that item is blocked until the write is completed.

A server requests a read lock the first time it tries to access an item. After the server has a read lock on the item, it holds that read lock until the lock manager notifies the server to release its read lock. At that time, it drops the item from its cache.

A write lock is requested whenever a server calls `getItemForUpdate()` or the first time `setPropertyValue()` is called. It is released at the end of the transaction. Note that, like any locking system, locked caching has the possibility of deadlock. The cache system detects a caching deadlock and throws an exception.

One server is designated as the lock manager; other servers can be designated as backup lock managers. If all the lock manager servers fail, caching is disabled for that item type. In that case, all requests query the database directly.

If your site is running multiple ATG servers, and you are using ATG Scenario Personalization, the best practice is to change the cache mode to locked on the following item descriptors in `/atg/userprofiling/userProfile.xml`:

- `individualScenario`
- `collectiveScenario`
- `scenarioInfo`

Cache Mode: Distributed

- The cache mode name is `distributed` or `distributedJMS`.
- This mode allows any server to read or modify any item without obtaining a lock.
- When a server modifies an item, it broadcasts a cache invalidation event to all servers (“peer-to-peer”).
- Cache events are transmitted by TCP or JMS.
- Suggested use: When data is read from multiple servers regularly and changed infrequently
- Disadvantage: Not scalable (TCP) or usable across clusters (JMS)

This mode is seldom used. In most cases, locked caching is preferable to distributed caching. Examples of when distributed caching might be useful include discussion boards and a “who is online” function that is used by many servers. Distribution messages are fired to all of the other servers when data is changed.

To ensure that no servers have stale data, the cache invalidation event message is sent immediately before the item update transaction is completed.

Distributed caching by TCP uses a `GSAEventServer` component to send invalidation events to other servers via their IP addresses and a port that is randomly selected by default. These events can be configured to be synchronous or asynchronous. For more information, refer to the section titled “Distributed TCP Caching” in the “SQL Repository Caching” chapter of the *ATG Repository Guide.*

Distributed caching by JMS slows performance considerably compared to distributed caching by TCP; it cannot be used across ATG clusters. However, with TCP some servers miss invalidation events if the TCP connection times out. JMS guarantees that all events are received by all servers. The exact method of transferring JMS messages is determined by your JMS implementation. To use distributed caching by JMS, you must enable the Cache Invalidator Service.

Cache Mode: Distributed Hybrid

- The cache mode name is `distributedHybrid`.
- It is a “hybrid” between locked and distributed.
- A central server lists what is cached on all servers.
- Servers send cache events to a central server, which forwards them to servers that need them.
- Suggested use: In larger environments and across clusters

Distributed hybrid caching is recommended when the application requires real-time knowledge of item updates across servers, caches a large number of items across servers, and updates items infrequently. It is also recommended when the environment includes a large number of servers (over 50) or requires cache synchronization between clusters.

Optimistic Locking

The `GSARepository` enables optimistic locking. This can be used in combination with any of the locking modes.

To make use of optimistic locking, add a new property, stored as an INTEGER in the primary table, to the item descriptor that will be using optimistic locking. This property acts as a version ID (referred to here as the *version ID* property). Next, add an attribute called `version-property` to the item descriptor tag. Its value is the name of the version ID property.

When several ATG servers read in the same `RepositoryItem` for update (using this item descriptor), the version ID property is read as part of the `RepositoryItem`. Keep in mind that using `getItemForUpdate()` starts a transaction. Later, when one ATG server tries to write this `RepositoryItem`, if the value for the version ID property in the `RepositoryItem` is the same as what is found in the corresponding database column, the version ID property value is incremented by ATG and the `RepositoryItem` is written. (The database column now has a number that is 1 greater than before.) If the same `RepositoryItem` on another ATG server with the original value for this property attempts to write to the database, the `GSARepository` notices the difference between its value and the new value in the database column, rolls back the transaction, and throws a `ConcurrentUpdateException`.

Cache Invalidation Methods

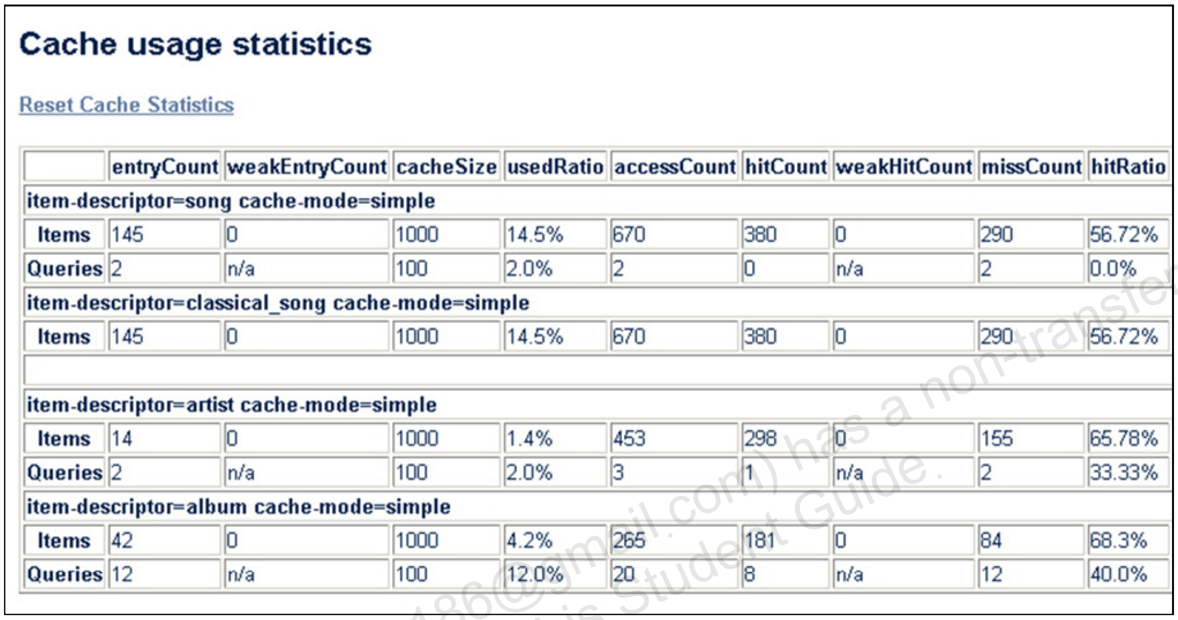
- To flush all item and query caches in a repository, use `invalidateCaches()` in `atg.repository.RepositoryImpl`.
- To flush caches for a specific item type, use one of the following parameters in `atg.repository.ItemDescriptorImpl`:
  - `invalidateItemCache()`
  - `invalidateCaches()`
  - `removeItemFromCache(itemId)`

Usually, cache invalidation happens automatically when repository items are changed through the Repository API. Sometimes, it is necessary to force cache invalidation manually (such as when the contents of the database are changed directly) without going through the Repository API.

Note: By default, these methods invalidate the caches only on the server on which the code is running. To invalidate the caches on the other servers in the cluster, add a Boolean parameter set to `true` (for example, `removeItemFromCache(itemId, true)`). For cache invalidation to be communicated across the cluster when using simple or locked caching, the `GSAInvalidatorService` must be enabled (see "Cache Flushing" in the *ATG Repository Guide* for details). Distributed caching sends the invalidation message by using TCP or JMS, as usual.

Viewing Cache Usage

The Dynamo Admin Component Browser enables you to view caching statistics for any repository.



Cache usage statistics

Reset Cache Statistics

	entryCount	weakEntryCount	cacheSize	usedRatio	accessCount	hitCount	weakHitCount	missCount	hitRatio
item-descriptor=song cache-mode=simple									
Items	145	0	1000	14.5%	670	380	0	290	56.72%
Queries	2	n/a	100	2.0%	2	0	n/a	2	0.0%
item-descriptor=classical_song cache-mode=simple									
Items	145	0	1000	14.5%	670	380	0	290	56.72%
item-descriptor=artist cache-mode=simple									
Items	14	0	1000	1.4%	453	298	0	155	65.78%
Queries	2	n/a	100	2.0%	3	1	n/a	2	33.33%
item-descriptor=album cache-mode=simple									
Items	42	0	1000	4.2%	265	181	0	84	68.3%
Queries	12	n/a	100	12.0%	20	8	n/a	12	40.0%

To view the cache statistics for a given repository, go to the Dynamo Admin URL for your server, select Component Browser, and then navigate the component folders to find the Repository component for the repository that you are interested in.

Using these cache statistics helps you optimize your cache sizes. Ideally, your cache should be big enough to hold all of the frequently hit items. A 90%−95% hit ratio for the most commonly hit item types is a good goal for excellent performance.

The example in the slide shows the cache statistics for `SongsRepository` components. The screenshot shows only some of the statistics available for your cache. To see the full listing of categories, refer to the repository component in the Component Browser.

Preloading Caches

- Items can be preloaded for improved performance by using the `<query-items>` XML tag.

```
<query-items item-descriptor="item">RQL query</query-items>
```

- To “precache” all items of a given item type:

```
<query-items item-descriptor="item">ALL</query-items>
```

- To use the cache from the last time ATG ran, set `restoreCacheOnRestart` to `true` on the repository component.
  - The cache is stored in a text file as a list of the cached items’ repository IDs.

The code examples in the slide show how to preload caches with the `query-items` tag. You can execute these queries manually by using the XML command window when viewing the repository through the Dynamo Administration Component Browser, or you can put them into a repository’s XML definition file to run them each time the repository starts.

Note: In addition to the `<query-items>` tag, there is also a `<load-items>` tag, which enables you to preload a smaller set of items based on specific query criteria.

Administrators can use the `<dump-caches>` tag to record the current contents of selected caches in the form of `<load-items>` tags or as a list of repository IDs.

Note that XML queries are an easy way to preload the cache, but not the most efficient way. If you have an extremely large repository (over around 100,000 items) and want to preload all the items in the repository, you should write your own code to do this by using the Repository API. Another option is to consider identifying the most commonly accessed 1000 (or so) items and preload just those.

Using XML queries in the repository definition template file for preloading also has another performance implication—the XML queries are run when the repository starts. Because ATG does not start until all its repositories are running, this can slow down the startup of your ATG servers. If you have a large number of items to precache, you should probably write an ATG service to perform the cache loading, which can run in the background after ATG starts.

Quiz

Which cache mode requires an additional server in the cluster?

a. Locked
b. Distributed
c. Distributed hybrid
d. Simple
e. All of the above

Answer: c

With distributed hybrid caching, a central server lists what is cached on all servers.

Road Map

- Caching architecture
- Cache modes
- External caching

This section of the course provides a brief overview of how to configure repositories to use external caching. For more details, see the chapter titled "External SQL Repository Caching" in the *ATG Repository Guide,* or click the link to "Viewlet 4a: External Caching" in the knowledge article titled "Oracle ATG Web Commerce 10.1 Release Training [ID 1490709.1]" on My Oracle Support (support.oracle.com).

Cache Types

There are three caches that ATG servers may use, depending on the cache configuration.

1. Weak cache:
   – Contains dereferenced `GSAItem` objects awaiting garbage collection in the ATG server's JVM
   – Is used as a lightweight local cache
2. Native ATG cache is stored in the ATG server's JVM.
3. External cache has a location determined by the Oracle Coherence configuration.

External Caching with Oracle Coherence

- You can configure your ATG application to use Oracle Coherence for external caching.
  - A key advantage is potentially better server and database performance.
- External caching is:
  - Best used for repositories that have very large caches of read-only data, such as a product catalog with over 100,000 cached items
  - Configured by the system administrator (servers) and developer (repositories)

Oracle Coherence is an in-memory data grid solution that enables organizations to predictably scale applications by providing fast access to frequently used data. Data grid software is middleware that reliably manages data objects in memory across many servers. By automatically and dynamically partitioning data, Oracle Coherence ensures continuous data availability and transactional integrity, even in the event of a server failure. It provides organizations with a robust scale-out data abstraction layer that brokers the supply and demand of data between applications and data sources. Features include caching, analytics, transactional data management, and event handling.

Depending on your configuration, external caching can:

- Reduce the portion of the server's Java Virtual Machine (JVM) that is used for caching
- Enable a larger cache

Your system administrator is responsible for configuring:

- ATG servers and (optionally) stand-alone Coherence servers to join a Coherence cluster
- The Coherence cache

The developer typically configures the desired repositories and item descriptors to use external caching.

Configuring Repositories for External Caching

1. Configure properties on the repository component.
2. Set the cache locality on the repository or the item.
3. Set the cache mode to `simple` or `distributedExternal` on the repository item.

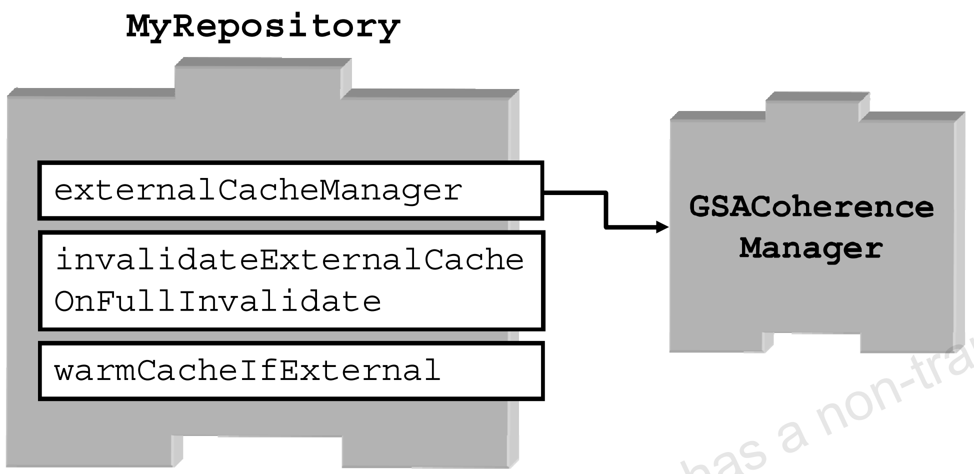
ORACLE

When you use external caching, two cache modes are valid options:

- **`simple`:** This cache mode is best for items that are not updated by production servers. There is no local ATG cache invalidation by production servers.
- **`distributedExternal`:** ATG servers listen for cache invalidation events from Coherence and invalidate items in their native and weak caches.

You have already learned how to set the cache mode for a repository item by using the `cache-mode` attribute.

Configuring Repositories for External Caching: Repository Component

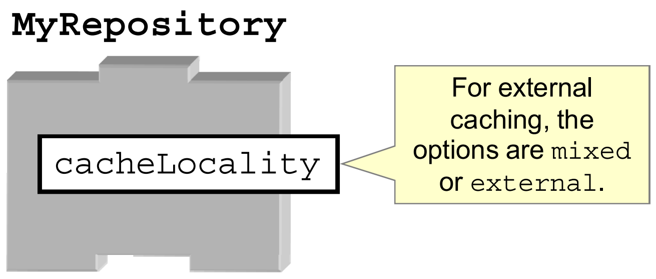


To use external caching, you need to set properties on the repository component:

- **`externalCacheManager`:** Is set to `/atg/adapter/gsa/externalcache/GSACoherenceManager`
- **`invalidateExternalCacheOnFullInvalidate`:** Boolean property that has a default value of `true`. Controls whether invalidation messages are sent to Coherence on:
  - Full cache invalidations
  - Selective cache invalidations
  - Calls to the repository's `invalidateCaches` method
  - Invalidations after a deployment from Content Administration
- **`warmCacheIfExternal`:** Boolean property that also defaults to `true` (Warming will pre-load data into cache after a Content Administration deployment.)

Configuring Repositories for External Caching: Cache Locality

- To set the cache locality on the repository:



- To set the cache locality on an item:

```
<item-descriptor name="sku"
cache-locality="external"
cache-mode="distributedExternal"
... />
```

The cache locality specifies where the cache is located. There are three options, but only two are available for externally cached items or repositories: `mixed` and `external`.

- **`local`:** The entire cache is kept in the ATG server's JVM native cache. This is the default setting.
- **`mixed`:** There is a smaller ATG native cache and a larger external cache.
- **`external`:** The only local cache is a weak cache, with the rest of the cache external.

The item descriptor setting overrules the repository setting.

When you set the cache locality to `mixed` or `external`, you limit the possible cache modes to simple or distributed external. Other modes are changed to simple when you restart the ATG server.

Caching Decisions and Roles

- Your architect or technical lead typically determines your caching strategy, including
  - Which cache mode to use
  - What data is cached
- The developer configures the appropriate cache mode in the repository definition files.
- The system administrator may:
  - Build the environment to support caching (such as setting up the server lock manager)
  - Tune the caching based on performance
  - Configure ATG with Oracle Coherence (if you are using it)

System administrator tasks, such as setting up lock managers and configuring ATG with Coherence, are covered in more detail in the *Configuring and Deploying an Oracle ATG Web Commerce Site* course.

Summary

In this lesson, you should have learned how to:

- Describe the basics of the ATG caching architecture
- Choose an ATG caching option
- Preload repository items
- Configure repositories for external caching

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Repository Guide*
  - "SQL Repository Caching"
  - "External SQL Repository Caching"

Practice 8 Overview: Configuring Repository Caching

This practice covers the following topics:

- Preloading the Songs Repository cache
- Configuring query caching
- Manually invalidating the cache

9

Extending the ATG Platform

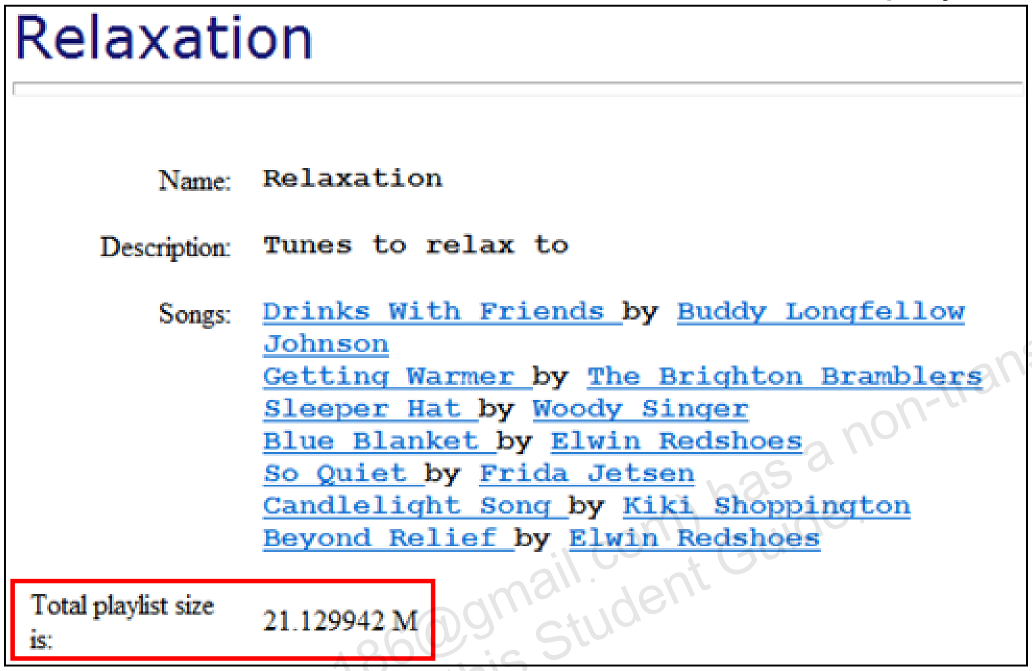
Custom Droplets

Objectives

After completing this lesson, you should be able to create and configure custom servlet beans (droplets).

Dynamusic Servlet Bean: Example

Dynamusic needs to calculate the total file size of a playlist.



Relaxation

Name:	Relaxation
Description:	Tunes to relax to
Songs:	Drinks With Friends by Buddy Longfellow Johnson Getting Warmer by The Brighton Bramblers Sleeper Hat by Woody Singer Blue Blanket by Elwin Redshoes So Quiet by Frida Jetsen Candlelight Song by Kiki Shoppington Beyond Relief by Elwin Redshoes
Total playlist size is:	21.129942 M

Road Map

- Review: ATG servlet beans
- Custom servlet beans
- Best practices

ORACLE

Review: ATG Servlet Beans

You have seen examples of the use of servlet beans in Dynamusic, including:

- Data access
  - `ArtistLookup`
- Programming functionality
  - `Switch`, `ForEach`
- Targeting servlet beans
  - `TargeterForEach`
- Event sender
  - `ViewItemEventSender`

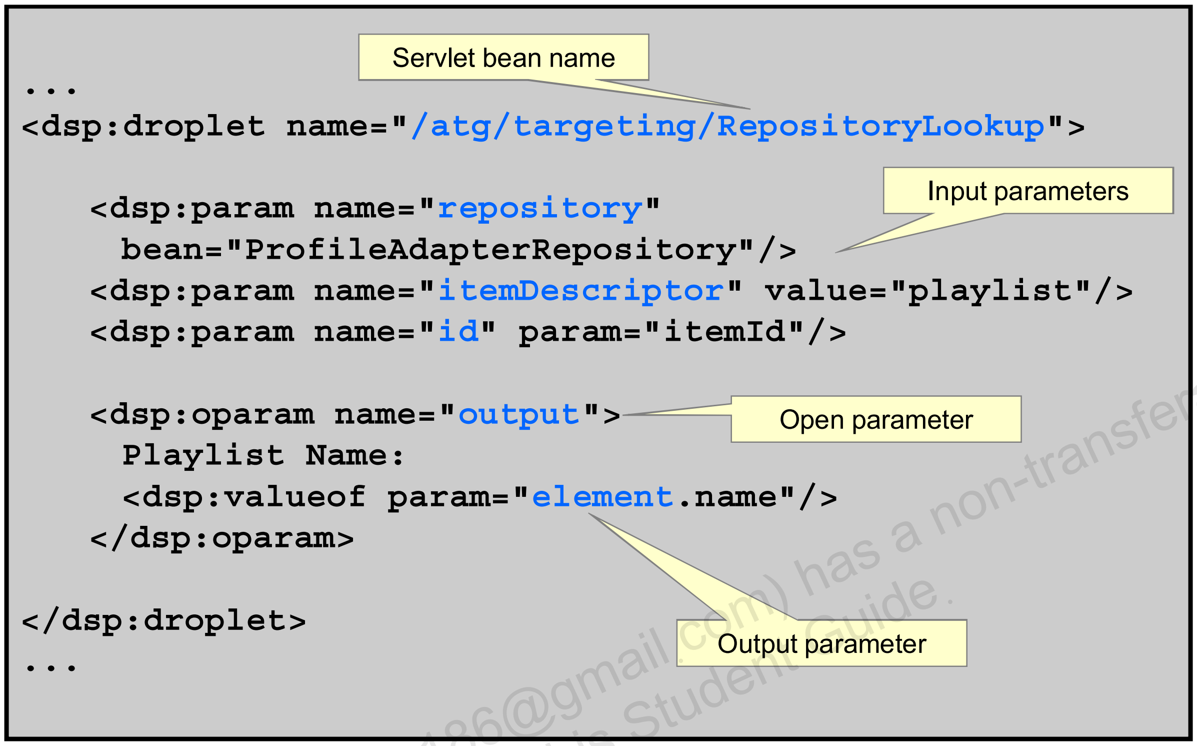
The slide shows a few examples of servlet beans that you used in the *Foundations of ATG Application Development* course.

Review: ATG Servlet Bean Parameters

Three types of parameters are used by servlet beans:

- Input parameters: Information that is given to the servlet bean by the page developer
  - Example: The array for a `ForEach` droplet)
- Output parameters: Objects that are calculated inside the code of the servlet bean
  - Example: Each object from the array
- Open parameters (oparams): Dynamic output that is rendered by the servlet bean (if and when needed)
  - Example: “empty” when the collection is empty

Review: ATG Servlet Beans in JSPs


```
...
<dsp:droplet name="/atg/targeting/RepositoryLookup">

   <dsp:param name="repository"
     bean="ProfileAdapterRepository"/>
   <dsp:param name="itemDescriptor" value="playlist"/>
   <dsp:param name="id" param="itemId"/>

   <dsp:oparam name="output">
     Playlist Name:
     <dsp:valueof param="element.name"/>
   </dsp:oparam>

</dsp:droplet>
...
```

Servlet bean name
Input parameters
Open parameter
Output parameter

The example in the slide is from `playlistDetails.jsp` and shows the `/atg/targeting/RepositoryLookup` servlet bean, which takes three input parameters:

- **`repository`**: This is the name of the repository that the servlet bean is looking up, `ProfileAdapterRepository`.
- **`itemDescriptor`**: Within the repository defined by the repository parameter, this is the particular item descriptor called `playlist`.
- **`id`**: This is the ID of the particular item. In the example, it is passed in from a request parameter called `itemId`.

The open parameter for this servlet bean is called `output`. Within the open parameter, information about the `playlist` item is displayed through the `element` output parameter.

Road Map

- Review: ATG servlet beans
- Custom servlet beans
- Best practices

ORACLE

Custom Servlet Beans

- You can create your own servlet beans (also known as *droplets*).
- Servlet beans are both servlets and beans.
  - To create a custom servlet bean, you must write a servlet that follows the JavaBeans specification.

Review: Servlets

- Servlets are Java programs (classes) that:
  - Run on a server
  - Respond to client requests
  - Typically generate dynamic content (HTML) or forward to a JSP to return a response
- Servlets implement the `javax.servlet.Servlet` interface.

Review: `service()`

The `javax.servlet.Servlet` interface contains a `service()` method that performs the work of the servlet.

```
public void service (ServletRequest request, ServletResponse response)
throws ServletException, IOException
```

ORACLE

The `service()` method takes two parameters:

- **A request object:** Contains the client request
- **A response object:** Contains the servlet response

The request object contains input parameters and data such as HTTP headers.

The response object contains possible output, such as data that will be written to the browser, cookies, and redirection information.

The arguments to `javax.servlet.Servlet.service()` are `ServletRequest` and `ServletResponse`. `HttpServletRequest` and `HttpServletResponse` extend `ServletRequest` and `ServletResponse` and are located in the `javax.servlet.http` package.

`IOException` is in the `java.io` package.

`ServletException` is in the `javax.servlet` package.

Creating and Using Custom Servlet Beans

The steps for creating and using a custom servlet bean are similar to the steps that are used for any Nucleus component.

1. Write a Java class:
   a. You must implement the Servlet interface.
   b. You must follow the JavaBeans specification.
2. Create a Nucleus component from your class.
3. Embed the servlet bean in a page by using the `droplet` tag.

Writing a Custom Servlet Bean Class

- To create a custom servlet bean, you normally write a class that extends `atg.servlet.DynamoServlet`.
- `DynamoServlet`
  - Implements the `javax.servlet.Servlet` interface, supplying default implementations for all methods except `service()`
  - Is a subclass of `GenericService`
- To extend `DynamoServlet`, you just need to implement the `service` method.

`DynamoServlet` implements the servlet methods so that you can concentrate just on writing the service method to implement the servlet functionality for your application.

`DynamoServlet` extends `HttpServletService`, which extends `ServletService`, which implements the `Servlet` interface.

`ServletService`, in turn, extends `TimedOperationService`, which extends `atg.nucleus.GenericService`, which allows a servlet bean to have access to many ATG services, such as logging.

The `DynamoServlet` API is documented in the *ATG API Platform Reference*.

DynamoServlet: `service()` Method

- The `service()` method services requests, calculates output parameters, and renders open parameters.
- ATG provides convenient subclasses for the `service()` parameters.

```
public void service (DynamoHttpServletRequest request,
DynamoHttpServletResponse response)
 throws javax.servlet.ServletException, java.io.IOException
```

`DynamoHttpServletRequest` and `DynamoHttpServletResponse` are ATG extensions to `HttpServletRequest` and `HttpServletResponse`.

These classes include methods that enable you to pass objects as parameters. These methods are discussed later in this lesson.

Creating a Custom Servlet Bean in `PlaylistSizeDroplet.java`

```
package dynamusic;
import javax.servlet.*;
import java.io.*;
import java.util.*;
import atg.servlet.*;
import atg.repository.*;

public class PlaylistSizeDroplet extends DynamoServlet {
    public PlaylistSizeDroplet() {}

    public void service (DynamoHttpServletRequest request,
    DynamoHttpServletResponse response) throws
    ServletException,IOException
    {
      Code to perform function of servlet
      ...
    } //end service()
} //end PlaylistSizeDroplet
```

For Dynamusic to calculate the songs' total size in bytes (and convert the size to MB), you need to create a new servlet bean that extends `DynamoServlet` and implements the `service` method.

Embedding a Custom Servlet Bean in a Page

- You embed a custom servlet bean in a page in the same way that you embed any servlet bean:

```
<dsp:droplet name="PlaylistSizeDroplet">
    ...
</dsp:droplet>
```

- The entire `droplet` tag is replaced with the response output generated by the `service` method of the servlet bean.

Note again that the servlet should not generate output directly. Rather, the servlet renders open parameters as appropriate. The page should format the output of the servlet within open parameters.

Servlet Bean Functionality

- In general, the `service()` method of a Servlet Bean performs three functions:
  - Pick up the values of the input parameters.
  - Calculate and set the values of the output parameters.
  - Render the open parameters.
- Servlet bean code can be complex, involving branching or looping. Some of the output parameters may not be set, or some of the open parameters may not be rendered.

Servlet beans do not necessarily render all of the open parameters or calculate all of the output parameters every time they run. For instance, the pseudocode of the `service` method of the `ForEach` droplet might look like this:

```
pick up "array" input parameter
if (array is empty) {
 render "empty" open parameter
}
else {
 render "outputStart" open parameter
 loop through array {
  set "element" output parameter to next object in array
  render "output" open parameter
 }
 render "outputEnd" open parameter
}
```

Accessing Input Parameters

- Use `request.getParameter()` to access `String` parameters.
- Use `request.getParameterValues()` to access arrays of `String`s.
- Use `request.getObjectParameter()` to access objects.
  - `getObjectParameter()` is an ATG extension.

`request.getParameter()` and `request.getParameterValues()` are methods of `javax.servlet.ServletRequest`.

`request.getObjectParameter()` is an ATG extension.

The standard servlet implementation supports only String parameters. It is the ATG extension to the servlet model that allows passing parameters of arbitrary type.

Setting an Input Parameter in `playlistDetails.jsp`

```
<dsp:droplet name="/atg/targeting/RepositoryLookup">
    ...

    <dsp:oparam name="output">

      <dsp:droplet bean="PlaylistSizeDroplet">

          <dsp:param name="songs" param="element.songs">
          ...
      </dsp:droplet>

    ...
</dsp:droplet>
```

The files for the code examples shown in this lesson are located in `<`*`coursedir`*`>/examples/lesson09`.

In the example in the slide, `PlaylistSizeDroplet` is called in the output of a `RepositoryLookup` droplet that retrieved the `playlist` object from the repository. `PlaylistSizeDroplet` expects an object (a set of songs) to be passed in. Because `PlaylistSizeDroplet` is called within the context of another droplet, you are passing in `element.songs` (in effect, you are passing in `playlist.songs`).

Accessing an Input Parameter in `PlaylistSizeDroplet.java`

```
public void service(DynamoHttpServletRequest request,
    DynamoHttpServletResponse response)
    throws ServletException, IOException {

    Set songs = (Set)request.getObjectParameter("songs");

    ...calculate output parameters...
    ...render open parameters...
}
```

The example in the slide is located in *<coursedir>*`/examples/lesson09`.

`PlaylistSizeDroplet` calls the `getObjectParameter` method to access the set of songs passed in. For a string input parameter, you simply use `String stringName = request.getParameter("StringName");`.

If the parameter requested is not defined, these methods return null. The best practice is to program for this situation.

Setting Output Parameters

- A servlet bean can add or change parameters to the request object through the `request.setParameter` method.
- These output parameters can then be used by the dynamic output on the JSP.

Setting an Output Parameter in `PlaylistSizeDroplet.java`

```
public void service(DynamoHttpServletRequest request,
    DynamoHttpServletResponse response)
    throws ServletException, IOException {

    ...access input parameters...

    request.setParameter("playlistSize", sizeInMegs);

    ...render open parameters...
}
```

The example in the slide is located in *<coursedir>*`/examples/lesson09`. It creates an output parameter called `playlistSize`.

`sizeInMegs` is the result of the calculation that occurs with the set of songs supplied by the input parameter.

Using an Output Parameter in `playlistDetails.jsp`

```
<dsp:droplet name="PlaylistSizeDroplet">
    <dsp:oparam name="output">
      <dsp:valueof param="playlistSize"/>
    </dsp:oparam>
</dsp:droplet>
```

The example in the slide is located in *<coursedir>*`/examples/lesson09`. It shows the `playlistSize` output parameter in the context of the `droplet` tag in the JSP.

Rendering Open Parameters

To render the value of an open parameter, invoke the `serviceParameter` method of the `request` object.

Servicing an Open Parameter in `PlaylistSizeDroplet.java`

```
public void service(DynamoHttpServletRequest request,
    DynamoHttpServletResponse response)
    throws ServletException, IOException {

    ...access input parameters...

    ...calculate output parameters...

    request.serviceParameter("output",request,response);
}
```

The example in the slide is located in `<coursedir>/examples/lesson09`.

Calling `serviceParameter()` renders the specified parameter. In the example in the slide, the call to `serviceParameter()` causes the servlet bean to render the content specified by the page in the `output` open parameter.

Using an Open Parameter in `playlistDetails.jsp`

```
<dsp:droplet name="PlaylistSizeDroplet">
   <dsp:oparam name="output">
     ...
   </dsp:oparam>
</dsp:droplet>
```

The example in the slide is located in *<coursedir>*`/examples/lesson09`. It shows how the `output` open parameter appears in the JSP.

Here is the full code in `PlaylistSizeDroplet.java`:

```
package dynamusic;
import javax.servlet.*;
import java.io.*;
import java.util.*;
import atg.servlet.*;
import atg.repository.*;

public class PlaylistSizeDroplet extends DynamoServlet {
  public PlaylistSizeDroplet() {}
  public void service (DynamoHttpServletRequest request,
            DynamoHttpServletResponse response)
     throws ServletException,IOException {

     int totalSize = 0;

     //Get the "songs" input parameter from the Profile.playlist item
    Set songs = (Set)request.getObjectParameter("songs");

     Iterator i = songs.iterator();

     // Loop through the playlist and pull out each song
     while(i.hasNext()){
      RepositoryItem song = (RepositoryItem)i.next();
     // getPropertyValue returns an Object - cast it to Integer
      Integer songSize =
          (Integer)song.getPropertyValue("songSize");
// Pull out the primitive int value
      int size = songSize.intValue();

// Add up the size of all songs
      totalSize += size;
     } // while

// Convert it to MB
   int megs = totalSize / 1000000;
   int fraction = totalSize % 1000000;
   String sizeInMegs = megs + "." + fraction + " M";

// Set the playlistSize output parameter
   request.setParameter("playlistSize", sizeInMegs);

// Service the output oparam
   request.serviceParameter("output",request,response);
 }
}
```

Creating the `PlaylistSizeDroplet` Component

You create a Nucleus component from a servlet bean in the same way that you create a component from any other Java class.

Road Map

- Review: ATG servlet beans
- Custom servlet beans
- Best practices

Separating Java and HTML

- When you create servlet beans:
  - The servlet beans should contain only Java
  - The pages should contain HTML
- This separation of Java and HTML is very important.
  - Ensures clean design
  - Enables HTML designers and Java coders to focus on their areas of expertise
  - Facilitates portability of servlet beans

ORACLE

Example: Bad Servlet Bean

The following servlet bean displays a list of numbers, including the HTML to format them as an unordered list:

```
public void service (DynamoHttpServletRequest req,
 DynamoHttpServletResponse res)
 throws ServletException, IOException
 {
   PrintWriter out = res.getWriter();
   out.println ("<UL>");

   for (int i=0; i< 10; i++) {
     out.println ("<LI> This is number " + i);
   }
   out.println ("</UL>");
 }
```

ORACLE

This servlet bean contains HTML formatting information. The formatting information, however, may change frequently in the course of a project. Putting HTML code in a servlet bean requires the servlet bean to be changed and recompiled for each HTML code change. As a result, the distinction between Java coders and HTML page designers becomes less clear as Java coders become involved in page design.

Example: Good Servlet Bean

The following servlet bean lets the page specify the HTML formatting:

```
public void service (DynamoHttpServletRequest req,
  DynamoHttpServletResponse res) throws ServletException, IOException {
  ...
  for (int i=0; i < numTimes; i++) {
    req.setParameter("number", new Integer(i));
    req.serviceParameter ("output", req, res);
  }
}
```

```
<dsp:droplet name="/essentials/GoodNumberDroplet">
  <dsp:param name="numTimes" value="10" />
  <dsp:oparam name="output">
    <li> This is number <dsp:valueof param="number"/>
  </dsp:oparam>
</dsp:droplet>
```

In the example in the slide, open parameters are used to separate Java and HTML. This servlet bean contains no HTML formatting. Instead, it passes the formatting as a parameter named `output`. It uses the `number` parameter to set the value for each line.

Quiz

Which of the following statements describe custom servlet beans? Select all that apply.

a. Your Java class must implement the `Servlet` interface and follow the JavaBeans specification.
b. When you extend `DynamoServlet`, you need to implement the `service` and `getParameter` methods.
c. Your servlet bean should contain Java and no HTML.
d. The page designer must include every open parameter in the JSP.

Answer: a, c

Answer b: You need to implement the `service` method only when you extend `DynamoServlet`.

Answer c: You provide possible open parameters for the page designer to use. The page designer then determines which parameters are important to use in the JSP. For example, the `ForEach` droplet provides the open parameters `outputStart` and `outputEnd` that the page designer may not need to use in the page.

Summary

In this lesson, you should have learned how to create custom servlet beans to generate dynamic HTML through Java objects.

For More Information

Documentation that contains more information about the topics covered in this lesson:

- *ATG Programming Guide*
  - "Creating and Using ATG Servlet Beans"
- General information about servlets:
  - *Java Servlet Programming,* by Jason Hunter, William Crawford, and Paula Ferguson (O'Reilly)
  - *Core Servlets and JavaServer Pages,* by Marty Hall (Sun Microsystems)
  - Sun tutorial http://java.sun.com/javaee/5/docs/tutorial/doc/bnafd.html

Practice 9 Overview: Creating a Custom Servlet Bean

This practice covers creating a custom servlet bean that iterates through the possible values of a repository item's enumerated property and returns the formatted HTML as defined by an open parameter.

A

Content Repositories

Objectives

After completing this appendix, you should be able to:

- Describe
  - SQL-based content repositories
  - The difference between SQL content repositories and regular SQL repositories
- Configure a content repository
- Add and edit content

Road Map

- Content repository overview
- SQL-based repositories
- Hierarchical repositories

Content in a Web Application

Content, in the context of a web application, is:

- Information available to be displayed
- Historically, a collection of web pages (or page fragments) stored in a file system

Targeted Content Delivery

- A major feature of ATG Scenario Personalization is the ability to deliver *targeted* content to users.
- Targeted content delivery involves selecting content items based on certain criteria. These criteria may depend on:
  - Information collected about a user
  - Other factors, such as the state of a Nucleus component, the time of day, or the particular page displaying the content

Here is an example of targeting from the Quincy Funds Reference Application, a fictional financial services site that offers mutual funds.

Information about mutual funds sold by the hypothetical Quincy Funds company is stored in a content repository. Each mutual fund is associated with an "aggressiveness index" indicating whether it is growth-oriented, income-oriented, and so on. Users at the site may choose their investment strategy: income, short-term growth, long-term growth, or retirement. A targeter can select appropriate funds to display to users based on the user's investment strategy and the fund's aggressiveness index.

Content Repositories

- Content to be delivered is organized as *content items* within a *content repository.*
- A content repository can:
  - Store the content items
  - Qualify each content item with *attributes* (also known as *metadata* or *properties*)
  - Respond to a query for items having particular values in one or more attributes
    - Content items with attribute values satisfying the query are returned.

Road Map

- Content repository overview
- SQL-based repositories
- Hierarchical repositories

SQL-Based Repositories: Basic Concepts

- Any SQL repository can be used to target content.
- The term *SQL content repository* refers to a SQL repository that models a file system hierarchy.
- SQL content repositories are covered in detail later in this appendix.

Despite the name, you do not have to use a SQL content repository to store your content in a SQL repository. Any SQL repository can be used to hold and deliver targeted content. The only difference between an ordinary SQL repository and a SQL content repository is that the SQL content repository uses “folder” content items to mimic a file system−like hierarchy.

The first part of this appendix discussed storing content in a general-purpose SQL repository. In the next section, you learn about the special configuration of a SQL content repository that provides additional content-oriented functionality beyond a general-purpose SQL repository.

Internal Content Versus External Content

A SQL-based repository can store only content metadata, or it can store the metadata *and* the content itself.

- If content is stored in the database, the item must have a property to hold the actual content data (stored “internally”).
- If content is stored in the file system, the item must have a property for the file name and path of the content (stored “externally”).

Internal Content: Example

```
<item-descriptor name="article">
   <table name="article" type="primary"
     id-column-name="id">

     <property name="title" data-type="string"/>
     <property name="author" data-type="string"/>
     <property name="date" data-type="string"/>

     <property name="articleText" data-type="string"/>

   </table>
</item-descriptor>
```

In the example in the slide, the text of a news article is stored in the database in a column called `articleText`. A potential problem with this approach is that the `articleText` column must be large enough to hold the entire article, but some databases have limits on column size.

The following sample JSP code displays the article text on a webpage:

```
Title: <dsp:valueof param="article.title" />
Author: <dsp:valueof param="article.author" />
Body: <dsp:valueof param="article.articleText" />
```

External Content: Example

```
<item-descriptor name="article">
   <table name="article" type="primary"
     id-column-name="id">

     <property name="title" data-type="string"/>
     <property name="author" data-type="string"/>
     <property name="date" data-type="string"/>

     <property name="articlePath" data-type="string"/>

   </table>
</item-descriptor>
```

The example in the slide shows an article item descriptor very similar to the example in the previous slide, except that instead of the actual text of the article, the descriptor has a property containing the file path of the file containing the text of the article.

The following sample JSP code displays the article text on a webpage:

```
Title: <dsp:valueof param="article.title" />
Author: <dsp:valueof param="article.author" />
Body:
<dsp:getvalueof id="articlePath" param="article.articlePath"
        idtype="java.lang.String">
   <dsp:droplet src="<%= articlePath %>"></dsp:droplet>
</dsp:getvalueof>
```

Note: The use of a SQL content repository, covered in the next section, lends itself well to an external-type repository if your content is arranged in a file system hierarchy. That section uses an external example in the SQL content repository section.

Road Map

- Content repository overview
- SQL-based repositories
- Hierarchical repositories

SQL Content Repositories

- SQL repository
  - Any SQL repository can hold content
- SQL content repository
  - A specialized SQL repository that organizes content into a hierarchy of “folders”
  - A “folder” structure that is modeled after a file system structure

The term *SQL content repository* is somewhat misleading because it implies that content cannot be stored in a “regular” SQL repository. This is not true. Any SQL repository can hold content and serve content for targeting (using the targeting droplets). The difference is that in a SQL content repository, each content item is associated with a “folder,” and the “folders” are arranged in a hierarchy. This models a file-based repository.

It is important to note that in this model, content items are considered synonymous with files (whether or not there is an actual file in the file system associated with the content item). Each content item (file) has a “path” or “location” consisting of its parent folder, that folder’s parent folder, and so on, just as a file in a file system has a “path” through directories in the file system.

Defining a SQL Content Repository

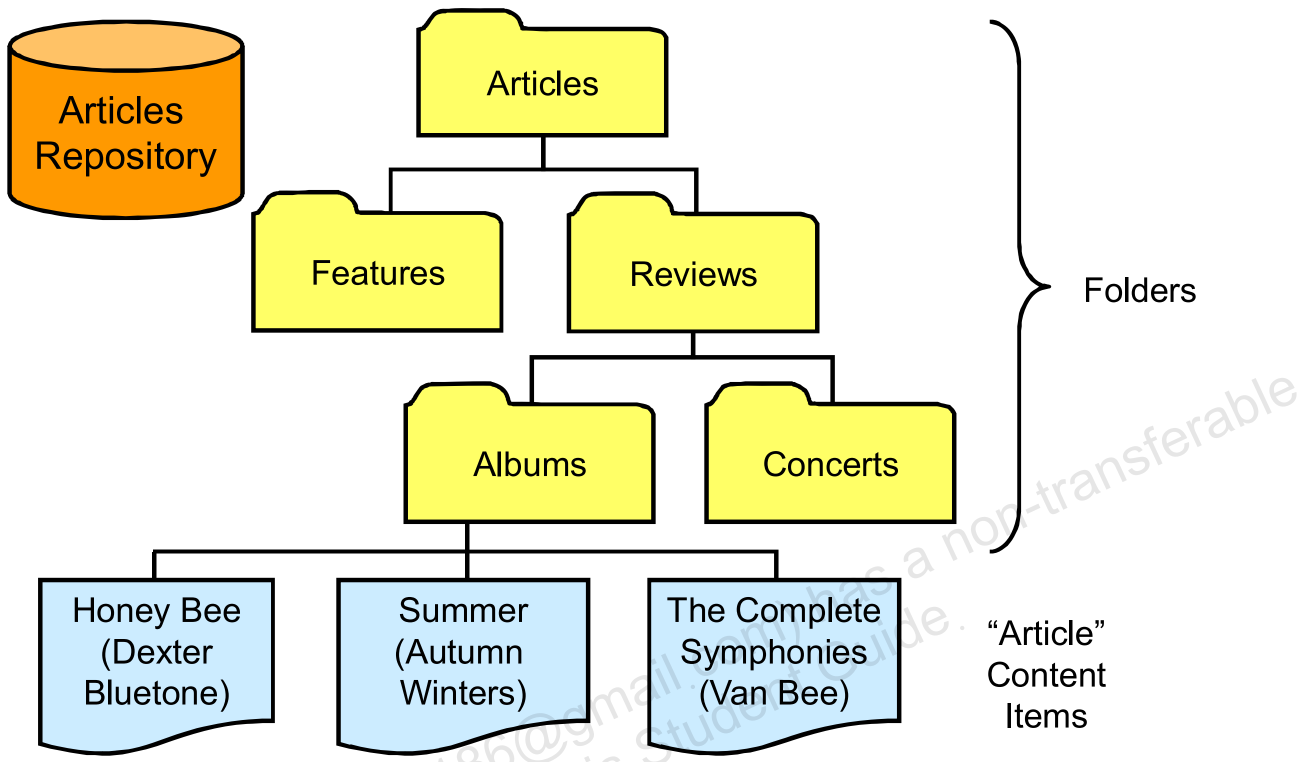
Special SQL content repository configuration requirements are:

- One *folder* item type
- One or more *content* item types

A SQL content repository must implement the `atg.repository.content.ContentRepository` interface. Because the `GSARepository` class implements the `ContentRepository` interface, any SQL repository can be a SQL content repository as long as it meets the requirements listed in the slide.

Using a SQL content repository also enables you to use the Content Repository API. This API has content-oriented methods to return content streams, calculate item paths, and so on. The Content Repository API is discussed later in this appendix.

Folder Hierarchy: Example



In the example in the slide, you have a repository consisting of articles about music. The folder hierarchy starts with an Articles folder with two subfolders called Features and Reviews. The Reviews folder has two subfolders as well: Albums and Concerts. There are three albums in the Albums folder that represent the article content items. The full path to the content item includes the folder hierarchy. For example, the full "path" for the Honey Bee content item (article) is `/Articles/Reviews/Albums/Honey Bee`.

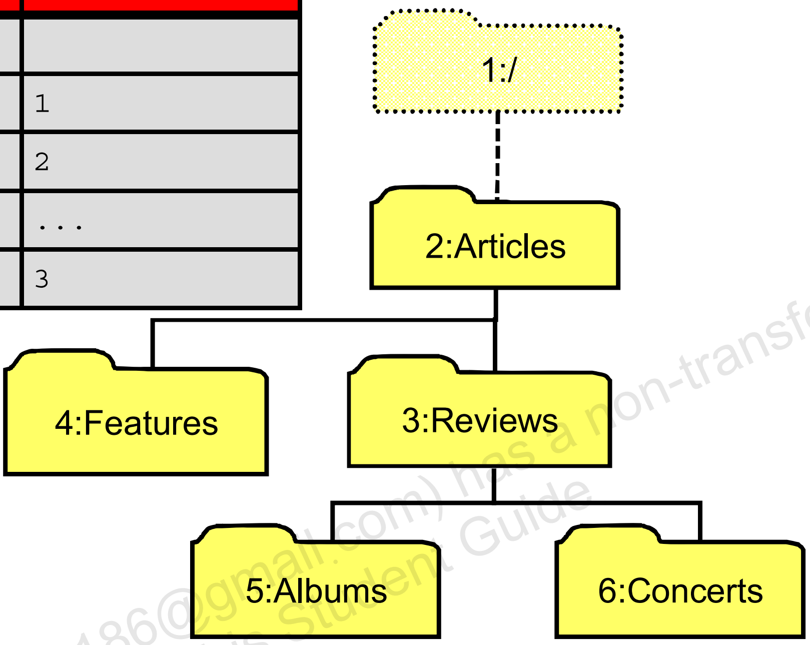
Modeling a Folder Hierarchy

- A SQL content repository models a folder hierarchy by defining a *folder* repository item type.
- Folder and content item types must have a property to retrieve an item's parent folder.
- Folders are stored in a folder table in the database.
- Each folder has a reference to its parent folder's ID.
- The path to a folder can be computed by traversing the folder "tree" represented by the folder table.

For a SQL repository to be a SQL *content repository*, exactly *one* folder item type must be defined, as well as one or more content item types. One folder in the table must be the "root" folder; that is, it must be the only folder with no parent folder and should have the name "/".

Folder Data Structure: Example

id	folderName	Parent Folder
1	/	
2	Articles	1
3	Reviews	2
...	...	...
6	Concerts	3



The example in the slide shows the folder structure as it appears in the database, starting with the “root” folder / at the top with an ID of 1. The Articles folder has an ID of 2 and its parent folder is / as represented by the entry of 1 (the parent folder’s ID) in the `parentFolder` column. The Features folder (ID 4) and Reviews Folder (ID 3) both have the Articles folder (ID 2) as the parent folder, and so on through the structure.

Folder Item Configuration

- Required attributes:
  - `folder=true`
  - `folder-id-property`
  - `content-name-property`
- At least three properties
  - The folder's ID
  - The ID of the parent folder
  - The folder's name

A SQL content repository must have exactly one item type with the `folder=true` attribute set. Note that the name of this item type can be anything; it need not necessarily be `folder`, because it is the `folder=true` attribute that marks it as a folder item.

The `folder-id-property` attribute should be set to the name of the property that contains the ID of the parent folder.

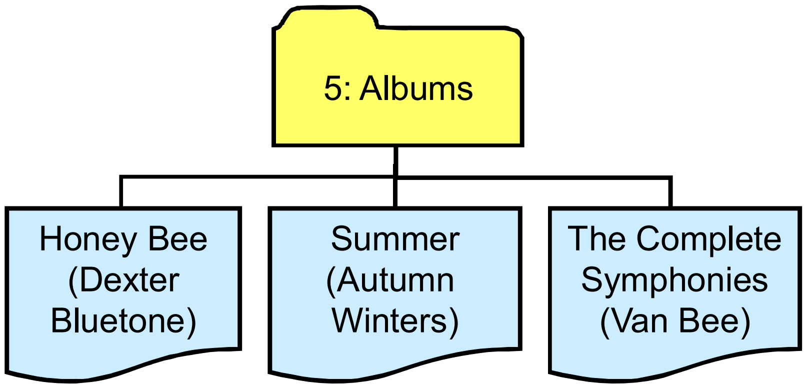
The `content-name-property` attribute should be set to the property containing the folder's name.

Folder Item Descriptor: Example

```
<item-descriptor name="folder"
    folder="true"
    content-name-property="folderName"
    folder-id-property="parentFolder">

    <table name="folder" type="primary"
      id-column-name="id">

      <property name="parentFolder" item-type="folder"/>
      <property name="folderName" data-type="string"/>

    </table>
</item-descriptor>
```

The example in the slide shows the XML item descriptor for the folder item type in the article repository. It has the `folder=true` attribute and defines the `content-name-property` as `folderName` and the `folder-id-property` as `parentFolder`. Those properties are then defined in the `table` tag.

Content Data Structure: Example


5: Albums
Honey Bee (Dexter Bluetone)
Summer (Autumn Winters)
The Complete Symphonies (Van Bee)

id	title	folder	articleName	author
100	Honey Bee (Dexter Bluetone)	5	honeyBee.html	A. Critic
101	Summer (Autumn Winters)	5	summer.html	I.M. Fan
102	The Complete Symphonies (Van Bee)	5	symVanBee.html	A. Critic

The example in the slide shows the data structure for the actual content items: the three album reviews. The columns include the ID for the article item, the title of the article, the parent folder (ID 5, which is Albums), the name of the article as an HTML page, and the author.

Content Item Configuration

- Attributes:
  - `content=true`
  - `folder-id-property`
  - `content-name-property`
  - `content-property`
- At least three properties:
  - Parent folder
  - Content name
  - Content

A SQL content repository can define any number of content item types to represent different kinds of content. Each content type must have the `content=true` attribute.

The `folder-id-property` attribute is set to the name of the property that contains a reference to the folder item containing this content item (the “parent folder”).

The `content-property` attribute is set to the name of the property that contains the content associated with this content item:

- The actual content itself in an internal-type content repository
- A reference to the file containing the content in an external-type content repository

The `content` property (the one referenced by the `content-property` attribute) can be of `data-type` String or byte (for content stored in the database), or it can be of `property-type atg.repository.FilePropertyDescriptor` (for content stored in a file).

The parent folder property should be an `item-type` pointing to the folder type.

Content Item Descriptor: Example

```
<item-descriptor name="article"
    content="true"
    folder-id-property="parentFolder"
    content-name-property="articleName"
    content-property="articleText">
    <table name="article" type="primary"
      id-column-name="id">
      <property name="parentFolder" item-type="folder"/>
      <property name="articleName" data-type="string"/>
      <property name="author" data-type="string"/>
    </table>

    <property name="articleText" property-type=
      "atg.repository.FilePropertyDescriptor">
      <attribute name="pathPrefix" value=
       "../My-Module/j2ee-apps/j2ee-app/web-app/Content"/>
    </property>
</item-descriptor>
```

This sample code in the slide shows the XML definition for the article item type definition for the articles content repository.

In the example, the actual content (text of the article) is stored in a file rather than in the database. The `articleText` property holds a reference to the file rather than to the actual article text. This approach might make even more sense if the hypothetical content were image files, movie clips, audio files, or other large media.

The `articleText` property is not part of a `<table>` descriptor. This means that the property is “transient” and is not stored in a column in a database table, as the other properties are.

The type of the property is `FilePropertyDescriptor`. This is a class that determines the location of a file by using the item’s path in the folder hierarchy, a path prefix, and the name of the item (corresponding to the file name).

Storing the Full Path of a Content Item

How do you find the path of a folder or content item?

- `folder-id-property` and `content-name-property`
- ID property
  - Set `use-id-for-path=true`.
- Content path property
  - Set `content-path-property`.

- **`folder-id-property`:** Each content item links to its containing folder. The full path to a folder can be calculated by traversing the "tree" represented in the table. The full path of the item would be the path of its parent folder combined with the content item's `content-name`. This is the default case.
- **ID property:** Because the full path name of an item is unique, it can be used as an ID, rather than the usual ID generated by the ID Generator Service. Using the item path for an ID means that you do not have to traverse the folder table to compute the path. It is already computed and stored in the content table. For example, if item Article 1 were in folder X, which in turn was in folder Y, the ID column value would be `/Y/X/Article1`. To do this, set the `use-id-for-path` attribute to `true`.

- **Content path properties:** You can also use a property other than the ID of the content table to store the path of the item. In the previous example, the content path property contains /`Y/X` and the content name property contains `Article1`. To do this, set the `content-path-property` item attribute to the property that contains the folder path of the item. This allows the path to be retrieved directly rather than computing it by traversing the folder tree.

Although storing the full path in a content item is convenient and saves time in computing the folder path, there are two potential problems with this approach:

- The absolute path may not fit into a single database column (particularly if the content files are deep within a folder hierarchy).
- The folder hierarchy in the folder table must be kept in sync with the paths of the content items. That is, if you have a content item with path-property /`Articles/Reviews/Album`, you must have an Albums item in the folder table, whose parent is a Reviews item in the folders table, and so on.

Note that the content repository Loader requires the use of the `content-path-property` attribute.

Accessing Content

- Content items in a SQL content repository are returned as `ContentRepositoryItem` objects.
- The `ContentRepositoryItem` interface provides a method to access content:
  - `getContentByKey()`
- The interface also includes these methods:
  - `getContentKeys()`
  - `getContentLastModified()`
  - `getContentLength()`

The full path of the content item interface is `atg.repository.content.ContentRepositoryItem`.

```
public java.io.InputStream getContentByKey(java.lang.String pKey)
throws RepositoryException
```

This method returns a stream containing the actual content associated with a SQL content repository item. The `pKey` argument should always be null.

Note: This argument is primarily for integration with content management systems, which support different versions of content. Passing an `html` key might return the HTML version of some content, whereas `text` might return a plain, unformatted version of the same item. A list of all possible keys is returned by `getContentKeys()`.

The advantage of this method is that it returns an input stream to the content, regardless of whether that content is stored in the database or externally in a file. This provides a particular advantage for very large content (such as MPEG files). Because `getContentByKey()` returns a stream, it does not require the entire content to be loaded before being served. An application can buffer the data to optimize its delivery.

Accessing Content Folders

`ContentRepositoryItem` implements `FolderItem`, so it also provides methods for accessing folder information for content items:

- `getItemPath()`
- `getAncestorFolderPaths()`
- `getAncestorFolderIds()`

- **`public java.lang.String getItemPath()`**: Automatically constructs the full path by using the Folder table, if necessary
- **`public java.lang.String[] getAncestorFolderPaths()`**: Returns an array of paths for all folders that are parents of this item. If the item is the root folder, it returns an empty array. For example, if you have a piece of content with the `itemPath` property value `/foo/bar/somepage.html`, this method returns `{/, /foo, /foo/bar}`. With these paths, you can fetch the content via the `ContentRepository.getFolderByPath` methods.
- **`public java.lang.String[] getAncestorFolderIds()`**: Returns an array of IDs for all folders that are parents of this item. If the item is the root folder, it returns an empty array.

Folder Information

- Folder items are returned as objects of type `ContentRepositoryFolder`, which also extends the `FolderItem` interface.
- `ContentRepositoryFolder` adds the following methods to provide access to a folder's content and subfolders:
  - `getChildContentIds()`
  - `getChildContentPaths()`
  - `getChildFolderIds()`
  - `getChildFolderPaths()`

The full class name is `atg.repository.content.ContentRepositoryFolder`.

- **`public java.lang.String[] getChildFolderPaths():`** Returns an array of folder names that access all the folders that are children of this folder
- **`public java.lang.String[] getChildFolderIds():`** Returns an array of IDs that access all the folders that are children of this folder
- **`public java.lang.String[] getChildContentPaths():`** Returns an array of document names that can be used to access the pieces of content in the folder
- **`public java.lang.String[] getChildContentIds():`** Returns an array of document IDs that can be used to access the pieces of content in the folder

Summary

In this appendix, you should have learned how to configure SQL content repositories.