# Contents

## Chapter 1: Model-driven development
About models ................................................................. 2
About the Modeler .......................................................... 5
Creating models ............................................................ 9

## Chapter 2: Model XML elements
annotation ................................................................. 11
annotation validators .................................................... 12
case ................................................................. 14
constraint ............................................................. 16
default ............................................................... 18
default entity .......................................................... 20
filter ................................................................. 24
function ............................................................... 31
id ................................................................. 32
inline ............................................................... 35
method .............................................................. 36
model .............................................................. 38
property ............................................................ 39
selector ............................................................ 47
service ............................................................ 47
style ............................................................... 49
variant ............................................................ 59
version ............................................................. 60

## Chapter 3: Model annotations
Annotations for server-side generation .............................. 62
Annotations for client-side generation ................................ 83
Annotations for LiveCycle ................................................ 90

## Chapter 4: Expression syntax
Data types ............................................................... 91
Interpretation of expressions and types by code generators .......... 96
Expression scope ........................................................ 97
Expression components .................................................. 97

## Chapter 5: Built-in functions
Functions by category ...................................................... 112
abs ................................................................. 114
ceiling .............................................................. 114
charAt .............................................................. 115
chr ................................................................. 115
collectionAvg .......................................................... 116
collectionMax .......................................................... 116
collectionMin  .............................................................................. 117
collectionSum  ............................................................................... 118
compare  ......................................................................................... 118
compareNoCase  .............................................................................. 119
createDate  ....................................................................................... 119
createDateUTC  ............................................................................... 120
dateAdd  ........................................................................................... 121
dateCompare  .................................................................................... 121
dateDiff  ............................................................................................ 122
datePart  ............................................................................................ 123
datePartUTC  ..................................................................................... 124
dayOfWeek  ......................................................................................... 124
dayOfYear  .......................................................................................... 125
daysInMonth  ..................................................................................... 125
daysInYear  ........................................................................................ 126
endsWith  .......................................................................................... 126
find  .................................................................................................... 127
findNoCase  ........................................................................................ 127
findLast  .............................................................................................. 128
findLastNoCase  ................................................................................... 128
findOneOf  .......................................................................................... 129
firstDayOfMonth  ............................................................................... 129
fix  ...................................................................................................... 130
floor  .................................................................................................. 130
getDay  .............................................................................................. 131
gDayUTC  ............................................................................................ 131
gHour  .................................................................................................. 132
gHourUTC  ........................................................................................... 132
gMinute  ............................................................................................ 133
gMinuteUTC  ....................................................................................... 133
gMonth  .............................................................................................. 134
gMonthUTC  ........................................................................................ 134
gQuarter  ............................................................................................. 135
gSecond  ............................................................................................. 135
gSecondUTC  ...................................................................................... 136
gYear  .................................................................................................. 136
gYearUTC  ............................................................................................ 137
insert  .................................................................................................. 137
isEmpty  ............................................................................................... 138
isLeapYear  ......................................................................................... 138
left  ....................................................................................................... 139
leftTrim  ............................................................................................... 139
len  ....................................................................................................... 140
lower  .................................................................................................... 140
max  ...................................................................................................... 140
min  ...................................................................................................... 141
Chapter 6: Client code generation

ActionScript objects ................................................................. 157
Service wrappers ................................................................. 164
Implicit service generation ..................................................... 187
Event registration for generated service wrappers ....................... 191
Using unsupported data types ................................................. 191
Model-driven Form generation ................................................. 192
Chapter 1: Model-driven development

Adobe® application modeling technology is a suite of technologies that enables model-driven development of Adobe® Flex® client applications. Application modeling technology consists of three things:

- An XML-based modeling language
- Tools for working with models and associated code generation
- A runtime environment

The application modeling language lets developers define entities, associations between entities, and additional behaviors.

The main purpose of application modeling technology is to make it easier and faster to build real-world data-centric applications. Application modeling technology provides a set of Java APIs that generate client-side ActionScript and server-side Java data-access code directly from a model.

New for use with Adobe® Flash® Builder™ and Adobe® LiveCycle® Workbench ES2 is a text and graphical modeling editor called the Modeler. The Modeler lets you create and manipulate models, and deploy models to the LiveCycle Data Services server. When you deploy a model to the server, application modeling technology generates Data Management Service destinations without any custom Java code. During the development process, the ActionScript code generator creates the ActionScript code that binds your Flex client application with your server. This generated ActionScript code enables easier Flex development. For more information, see “About the Modeler” on page 5.

LiveCycle Data Services also provides the application modeling technology runtime that automates common functionality such as data persistence, data management, and server validation of model constructs. This functionality improves developer productivity and simplifies configuration and integration with server-side business logic and systems. For more information, see “Model-driven applications” in Using LiveCycle Data Services.

Application modeling technology is useful for many different use cases and programming platforms. Here are some of the things you can do with it:

- Generate a model from an existing SQL database, edit it in the Modeler, and then generate client and server data-access code from that model. This is end-to-end model-driven development for both the client and server implementations.
- Manually create a model in the Modeler, and then generate SQL database tables as well as client and server data-access code. This is end-to-end model-driven development for both the client and server implementations.
- Generate a model from an existing remote data service implementation, and then generate client code from that model for calling the remote service. You have no control over the service implementation on the server in this workflow.

Model-based development with LiveCycle Data Services

When you use LiveCycle Data Services in combination with the Modeler, you can perform end-to-end model-driven development to create both client and server objects for your application. To jump-start development with existing data assets, the Modeler and LiveCycle Data Services server let you generate a model directly from an existing database. You can also build a model from scratch and generate database tables and Data Management Service destinations when you deploy the model to the server. You save the same model in a Flash Builder project to generate client-side ActionScript to use in a Flex client application. For more information, see “Model-driven applications” in Using LiveCycle Data Services.
In addition to end-to-end model-driven development, you can use model-driven development with existing LiveCycle Data Services assets, such as remote object destinations and legacy Data Management Service destinations. In these scenarios, you automatically generate a model by introspecting an existing service destination that resides on the server. With this type of model-driven development, you only generate client code. You have no control over the service destination on the server.

The role of Flash Builder in model-driven development

There are two distinct ways to use Flash Builder for model-driven development. You use Flash Builder for both end-to-end model-driven development and when working with existing service implementations.

For end-to-end model-driven development, you use Flash Builder in conjunction with the Modeler. When you save a model in the Modeler, ActionScript client code is automatically generated in your Flash Builder project.

The second way to use Flash Builder for model-driven development involves running a service import wizard on an existing service implementation. When you run the service import wizard, Flash Builder generates a model. Model generation kicks off ActionScript client code generation. This feature is available on platforms such as PHP, Adobe ColdFusion, Adobe BlazeDS, Adobe LiveCycle Data Services, and Adobe LiveCycle Foundation. This type of workflow generates only client code and gives you no control over the service implementation on the target server.

Note: LiveCycle Data Service users can take advantage of the ActionScript code generation features in Flash Builder and the Modeler to generate ActionScript code based on any valid model and build an end-to-end data-access application.

About models

A model is an abstract structure that describes how data is represented and accessed. A model is broadly similar in scope and granularity to a relational database schema. It defines a set of structured data collections. A model defines entities, which are abstract interrelated data objects. Each entity represents a custom data type. A model also defines abstract services responsible for delivering entities to clients. Services declare collections of functions, which are external callable endpoints that can be called within value expressions.

The application modeling language is an XML-based language that lets you construct models that define data entities, the relationships between those entities, and abstract services responsible for delivering the entities to data clients. A model is stored in an XML file with a .fml extension.

The modeling language works with a set of Java-based APIs that process and validate the content of models and can also generate concrete application code based on the model. For example, the modeling APIs provide built-in support for generating the following types of objects:

- ActionScript-based objects that represent model entities and services for calling data services from an Adobe Flex client application.
- Java-based objects that represent model entities, the relationships between those entities, and services on a LiveCycle Data Services server. The generated services can make queries to database tables that correspond directly to the entity elements in the model.

You can generate ActionScript-based client code and Java-based server code from the same model that maps to a SQL database. As a result, you can quickly create a functional Flex client that calls a LiveCycle Data Services destination and manipulates data in a SQL database.

In addition to entities and services, definitions found in a model include styles, which bundle user-interface-related information with data types, and annotations, which enable consumers to add domain-specific information.
Entities

Entities represent custom data types. They are made up of data properties that correspond to persistent storage and derived properties that are not backed by persistent storage and enable behavior customization in expressions on data properties and function calls. You can use variants to make parts of the structure conditional on the state of other parts of the structure.

A persistent entity is explicitly marked as persistent in the persistent attribute of an entity element in a model. It represents a freestanding entity. It is a particular collection of values that conform to the entity’s structural and behavioral specification, is stored, for example, in a database, and evolves over time. Persistent entities must contain at least one id property, and can also contain at most one version property.

A non-persistent entity represents the set of all possible values conforming to the entity definition. It is similar to a class definition.

Data properties

Data properties are named, strongly typed value cells.

id properties

id properties are special data properties that together uniquely identify an instance of the entity.

Derived properties

Derived properties define named expressions. They can reference other data or derived properties, built-in functions, and functions of services declared in the same model.

Constraints

Constraints are boolean-valued derived properties. The value of a constraint defines the subset of valid entity instances.

Variants

Variants are collections of conditional substructures. A variant contains cases that are selected by a value expression over the rest of the entity’s properties.

Inlines

An inline adds the child definitions of a specified target entity to the contents of the entity that contains the inline. The child definitions of the target entity are present as if declared at the point of the inline element.

filters

A filter describes a call to a persistence layer that returns a collection of entity instances. A filter translates to a fill method on a LiveCycle Data Services destination. You only use filters with the Model Assembler feature in LiveCycle Data Services. Each persistent entity can specify any number of filters. Non-persistent entities cannot contain filters.

methods

A method defines the prototype of a block of application logic that can take parameters and be referenced by derived properties, constraints, and other methods. The implementation of a method can be provided externally for each domain in which the model is used.

More Help topics

“entity” on page 20
Services
Services are made up of abstract function definitions. Specific consumers of a model could have knowledge of how services and functions are treated in their domain. For example, a consumer that uses application modeling technology to generate ActionScript code could interpret a model service to correspond to a WebService object and model functions to correspond to operations on that WebService object.

More Help topics
“service” on page 47

Functions (in services)
A function gives the prototype of an externally defined function within a service. Functions can be called from entity expressions, giving model definitions access to external data and functionality using type mapping mediated through application modeling technology.

More Help topics
“function” on page 31

Styles
A style is a collection of user-interface-related attributions that can be associated with a property element in a model. Styles let model designers reuse user interfaces aspects across multiple properties.

More Help topics
“style” on page 49

Annotations
Annotations enable consumers to add domain-specific information to a model. Annotations are a convenient way to associate data with model definition elements.

Annotations can be attached to almost everything in a model, using simple name-value pairs. Users of the model, such as tools, can annotate the model with domain-specific data.

More Help topics
“annotation” on page 11

Expressions and built-in functions
The application modeling language provides an expression language that you use to produce runtime values for derived properties, constraints, and variants by performing computations on inputs from properties in a model.

The application modeling language includes a set of built-in functions for use in expressions. These standard functions are treated as a collection of static utility functions and are not operations on objects in the manner of ActionScript and Java. The primary use case for these functions is to produce derived properties, with inputs that come from properties in a model.
About the Modeler

The Modeler is a tool for creating and editing models. The Modeler provides an XML source editor called the Source view and a graphical editor called the Design view. You can toggle back and forth between the two modes to best suit your own development style. Each mode works with a set of panels for specific viewing or editing capabilities.

The Modeler is built on the open-source Eclipse development platform. It consists of a set of Eclipse plug-ins that you install into Eclipse or the stand-alone Flash Builder tool, which itself is built on Eclipse.

In Eclipse or the stand-alone Flash Builder, the Modeler is contained in an Eclipse perspective called the Model perspective. You can use the Modeler with Flash Builder by switching between the Flash perspective of Flash Builder, and the Model perspective. The Modeler also appears as a tabbed pane in LiveCycle Workbench for use in LiveCycle-specific development.

Modeler Source view

The Modeler Source view contains a text editor for working directly with XML structure in a model file. The editor highlights the syntax of the model structure with color coding. It also validates the model and displays a red circle with an x through to the left of each line in the model that contains invalid content. The Modeler also displays all errors in a standard Eclipse Problems view.

When typing in Source view, hold down the Ctrl key and press the spacebar to view and select from the possible child elements or attributes that an element can contain.

When you are in the Source view, the Outline view displays a tree view of the model structure. You can click branches of the tree to expand and collapse them. The outline view stays in sync with your selections in the editor.

The Source view toolbar contains the following buttons:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collapse All</td>
<td>Collapses all XML elements in model.</td>
</tr>
<tr>
<td>Collapse Annotations</td>
<td>Collapses all annotation elements in model.</td>
</tr>
<tr>
<td>Deploy Model to LCDS Server</td>
<td>Opens the Deploy Data Model To LCDS dialog from which you deploy a model to the server.</td>
</tr>
<tr>
<td>Expand All</td>
<td>Expands all XML elements in model.</td>
</tr>
<tr>
<td>Expression Builder</td>
<td>Opens the Expression Builder, an editor for writing modeling language expressions.</td>
</tr>
<tr>
<td>Generate Code</td>
<td>Generates ActionScript, Java, and model-driven Form code depending on the code generation settings for the project or workspace.</td>
</tr>
<tr>
<td>Services Import Wizard</td>
<td>Opens the Import Service dialog from which you import existing service implementations.</td>
</tr>
</tbody>
</table>
**Modeler Design view**

The Modeler Design view provides a graphical environment for working with models. You can add elements to a model by dragging objects on to objects that can contain children, or by right-clicking objects that can contain children.

You can deploy models to a LiveCycle Data Services server by clicking the Deploy Model button on the toolbar.

The Design view shows associations between entities as lines between entity objects on the canvas. When you click on an association line, the Properties view displays the properties of the relationship between the associated entities and you can change any of the editable properties. For information about associations, see “entity associations” on page 44.

The Design view toolbar contains the following buttons:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deploy Model to LCDS Server</td>
<td>Opens the Deploy Data Model To LCDS dialog from which you deploy a model to the server.</td>
</tr>
<tr>
<td>Generate Code</td>
<td>Generates ActionScript, Java, and model-driven Form code depending on the code generation settings for the project or workspace.</td>
</tr>
<tr>
<td>Services Import Wizard</td>
<td>Opens the Import Service dialog from which you import existing service implementations.</td>
</tr>
<tr>
<td>Show Grid</td>
<td>Displays a grid for positioning objects on the canvas. By default, objects snap to the grid.</td>
</tr>
<tr>
<td>Show Rulers</td>
<td>Displays vertical and horizontal rulers for positioning objects on the canvas.</td>
</tr>
<tr>
<td>Zoom</td>
<td>Zooms the canvas in and out.</td>
</tr>
</tbody>
</table>

**Properties view**

When you are in Design view, the Properties view content is based on the currently selected object on the canvas. For example, when an entity is selected, you can view and edit general settings, code generation settings, Data Management Service (DMS) annotations, filters, methods, security settings, and clustering settings for that entity.

For entities in the context of LiveCycle Workbench ES2, you can select Create LiveCycle data types on deploy so that data types that entities define are available to LiveCycle processes. For more information, see Creating Processes Using LiveCycle Workbench ES2 in the LiveCycle ES2 help.

When you select a property, constraint, or variant in an entity, you can view and edit attributes such as general settings, DMS annotations, and styles in the Properties view.

By default, the values of derived properties in an entity are recalculated automatically in ActionScript code generated from a model. You can uncheck the Auto-recalc checkbox to change this behavior. The `calculate_propertyName()` method on a model-level metadata object is available for every entity object instance; this method lets you manually trigger recalculation of derived properties for which automatic recalculation is disabled. For example, you would call `myEntityObject._model.calculate Derived1()`, to recalculate the derived property named derived1. For more information about generated ActionScript code for derived properties, see “Derived properties in entities” on page 157.

The Styles pane for properties and constraints provides a full-featured style editor in which you can view, add, and edit global and inline styles. The style editor lets you easily extend any global style already in the model. For information about styles, see “style” on page 49.

While in the General pane for properties, constraints, and variants, you can also click the Expression Builder button (fx) to the right of the value expression field to open the Expression Builder, which is an editor for writing modeling language expressions. For information about the Expression Builder, see “Expression Builder” on page 7.
Objects bar
The objects bar, located on the left side of the Design view canvas, displays graphical representations of model elements that you can drag onto objects in the Design view canvas. For example, you can drag an entity to the canvas, and you can drag a property or a variant to the entity.

Styles view
The Styles view lets you add, view, and edit global (model-level) styles. You can apply these styles to any model elements that support styles.

The styles Properties view lets you configure style masks, messages, validations, and annotations.

When you select an element that supports styles, the Properties view for that element shows a Styles pane in which you can apply styles. You can also create an inline style for the selected element instead of applying a global style. The Styles pane provides a full-featured style editor in which you can add, view, and edit styles. The style editor lets you easily extend a global style.

For information about styles, see “style” on page 49.

Services view
The Services view lets you add, view, and edit services. The Services view shows only the explicit services in the model. The corresponding Properties view lets you configure service functions and annotations.

The service element is only allowed at the top level of the model.

In the General tab of the Services view, add or view a service name. In the Functions tab of the Services view, add or view function arguments, remote function name, function return type, and whether the function returns a collection.

Expression Builder
The Expression Builder is a specialized editor for writing expressions. When you select a property, constraint, or variant in Design view and select the General properties pane, you can click the fx button to the right of the value expression field in the Properties view to open the Expression Builder. The Expression Builder button is also available on the Source view toolbar.

The top pane of the Expression Builder is the editing window where you write an expression. You can write expressions manually, or use the toolbar below the editing window and the items listed in the right side of the middle pane.

When you select an entity in the left side of the middle pane, the right side lists the entity properties for use in an expression. When you select a built-in function category in the left side, the right side lists the functions in that category for use in an expression.

The bottom pane of the Expression Builder provides help for the currently selected item in the middle pane.

*Note:* To add a line break in an expression, use `chr(13)`. The `chr` function is a built-in function that returns a character with the specified ANSI character code. The ANSI character code for a line break is 13.

More Help topics
“Expression syntax” on page 91

“Built-in functions” on page 112
RDS data view
The RDS data view displays SQL databases that you have configured as JDBC data sources on the server. RDS stands for Remote Development Services. Drag database tables from the RDS data view to the Modeler Design view to generate matching entity elements and model-level datasource and hibernate.dialect annotations in the model. Each table corresponds directly to a generated entity element.

Drag the top-level node of the database to the Modeler Design view canvas to generate model-level datasource and hibernate.dialect annotations in the model without also generating entities.

You configure JDBC data sources and RDS on your LiveCycle Data Services server. For information about configuring RDS, see “Model-driven applications” in Using LiveCycle Data Services.

Problems view
The Problems view lists the validation errors in a model. When you double-click an item in the list, the Modeler highlights that section of the model in the Source view. If you start in the Design view, the Modeler switches to the Source view when you double-click an item in the list.

Model Explorer view
The Model Explorer view lets you view and undeploy models deployed to the LiveCycle Data Services server. You can also refresh the active RDS server, show the RDS configuration dialog, and download Database Description Language (DDL) statements from deployed models.

Modeler configuration preferences
To configure the Modeler at the workspace level, on the Eclipse menu bar select Window > Preferences > Adobe > Data Model and expand the Data Model entry. You can configure code generation and editor preferences.

Code generation configuration
The Code Generation dialog lets you configure ActionScript, Java, and model-driven Form code generation. You can set each type of generator to the default generator or a custom generator if one is available. By default, Java generation is set to None.

Click the Generator Configuration buttons to set the Template Lookup Path and Base Output Path information. The {project_root} token resolves to the root of the Flash Builder project in which you are generating code. The {project_as_source_folder} variable resolves to the configured source directory for the Flash Builder project.

You can also configure code generation at the project level. First, you must enable project-level settings. To do so, right-click a project name in the Package Explorer and select Data Model > Project Specific Settings > Add.

After you have added project-specific settings, from the main menu bar select Project > Properties > Code Generation > Use Project Settings to configure code generation at the project level. The initial project settings are based on the current workspace settings. Click Restore Defaults to return to the workspace settings. Click Apply to save settings.

Design mode configuration
The Design Mode dialog lets you configure default settings for the following items. These settings are applied to the model when you work in the Modeler Design view.

- Default Server Type
• Newly created entities are persistent
• Create LiveCycle datatypes on application deploy
• ID’s are generated values

Editor configuration
The Editor dialogs let you configure preferences for the Modeler Source view. You can configure many aspects of XML editing, syntax coloring, model file templates, and typing preferences.

RDS server configuration
Use the RDS configuration dialog to configure connections to RDS servers on a LiveCycle Data Services server. You must configure an RDS server in order to use the “RDS data view” on page 8.

For information about configuring RDS, see “Model-driven applications” in Using LiveCycle Data Services.

Creating models

Creating a model
You can create a model by selecting File > New > Model to open the New Data Model wizard. You can access the same functionality from a right-click context menu. This option is available when you use the Modeler in LiveCycle Workbench or Flash Builder. You can create a blank model or a model from a database. If you choose to create a model from a database, you specify the RDS server, data source, and database tables. You can give the model any name and directory location.

For ActionScript generation in Flash Builder, by default the model must be saved in a directory named .model in the root directory of the Flex project and the model name must match the Flex project name. Select Project > Properties > Data Model to change the default model directory and or filename for the active Flex project.

To create a model when you are using the Modeler with Flash Builder, you can click the Open Model for Active Project button at the right end of the Package Explorer toolbar. When a new model is created, it is saved in a directory named .model in your Flex project. When a model exists, you click the Open Model for Active Project button to open it. The same button appears in the toolbar of the Data/Services view when you are in the Flash perspective.

Generating model content from database tables
LiveCycle Data Services users can generate entities in a model from existing SQL database tables and then modify and add to the generated content. For more information, see “Model-driven applications” in Using LiveCycle Data Services.

Generating database tables from a model
LiveCycle Data Services users can generate database tables from entities in a model. For more information, see “Model-driven applications” in Using LiveCycle Data Services.
Generating a model from an existing data service

The Services Import Wizard button on the Source view and Design view toolbars lets you generate a model from an existing data service implementation. You can leave the model as is or make changes that affect client-side code generation. This feature is primarily intended for LiveCycle Workbench users. When you use the Modeler in Flash Builder, the Services Import wizard is available as a standard part of Flash Builder.

Since the server-side implementation is not generated from the model when you import a service, avoid code changes specific to server-side functionality, such as DMS annotations. For model features that can have both client and server aspects, such as styles and constraints, the server-side functionality is ignored, but the client aspects of these features are useful even in the absence of server-side functionality.
Chapter 2: Model XML elements

This XML element reference lists the application modeling language elements in alphabetical order.

The top-level element of a model is the “model” on page 38 element, which can have zero or more “annotation” on page 11, “entity” on page 20, “service” on page 47, and “style” on page 49 elements.

annotation


Application modeling technology uses specific sets of annotations for client and server code generation. For information about annotations for client-side ActionScript and server-side Java code generation, see “Model annotations” on page 62 and “Client code generation” on page 156.

Modeler Design view

You create and edit annotations in the Properties view in the context of the currently selected item in the Design view. All settable properties of an element are shown in the various tabs of the Properties view and many of these are annotations. For example, when you set the Server Type of an entity in the Properties view, you are actually setting the ServerType DMS annotation. To see if something you set is an annotation, switch to Source view after setting it.

You can also display, edit and create annotations for the currently selected element with the Edit Annotations button in the General tab of the Properties view.

Can appear

One or more annotation elements can appear as the initial child elements of any definition element. Definition elements include “model” on page 38, “entity” on page 20, “property” on page 39, “constraint” on page 16, “variant” on page 59, “case” on page 14, “default” on page 18, “inline” on page 35, “function” on page 31, and “style” on page 49. You cannot apply more than one annotation element with the same name attribute value to a definition element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>required</td>
<td></td>
<td>Annotation group name.</td>
</tr>
</tbody>
</table>

Child elements

An annotation element can contain zero or more elements of the following form:

<item name="name">value</item>

Each instance of the name attribute within a given annotation element must be unique. There are no other constraints on the contents of the name attribute.
Example

The entity in the following example contains annotations that application modeling technology code generators use when creating ActionScript (client-side) and Java (server-side) implementations of the entity:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Product" persistent="true">
    <annotation name="ServerProperties">
      <item name="ServerType">LCDS</item>
    </annotation>
    <annotation name="DMS">
      <item name="Table">PUBLIC.PRODUCT</item>
    </annotation>
    <id name="productid" type="integer">
      <annotation name="DMS">
        <item name="ColumnName">PRODUCTID</item>
      </annotation>
    </id>
    <property name="description" type="string" length="255">
      <annotation name="DMS">
        <item name="ColumnName">DESCRIPTION</item>
      </annotation>
    </property>
    <property name="price" type="float">
      <annotation name="DMS">
        <item name="ColumnName">PRICE</item>
      </annotation>
    </property>
    <property name="productname" type="string" length="255">
      <annotation name="DMS">
        <item name="ColumnName">PRODUCTNAME</item>
      </annotation>
    </property>
  </entity>
</model>
```

annotation validators

Annotation validators are custom Java classes that you can use to validate annotations. You can register any number of annotation validators with a model at any time before or after resolution. Once registered, an annotation validator is called every time the model is resolved. In the process, the application modeling technology runtime traverses the complete structure of the model and calls the appropriate methods on the validator.

Before using a model, the server verifies that the model meets certain requirements. To validate annotations in the model, you can write a server-side annotation validator that extends the fiber.core.impl.AbstractAnnotationValidator class and use it in the following way:
model = loader.loadModel(reader, location, buildContext, false);
model.registerAnnotationValidator(new ServerSideAnnotationValidator());
if (!model.resolve(buildContext))
{
    // see what went wrong using the buildContext and report an error
}
else
{
    // carry on with model deployment
}

The model is loaded unresolved and a new AnnotationValidator instance is registered. The call to the
model.resolve() method carries out the core language resolution, followed by any logic implemented in the server-
side annotation validator. The following example shows a server-side annotation validator implementation:

```java
public class ServerSideAnnotationValidator extends AbstractAnnotationValidator
{
    public static final String ID = "SERVER_SIDE_ANNOTATION_VALIDATOR";
    public String getId() { return ID; }

    public boolean isLCDSManaged(Entity entity)
    {
        return "LCDS".equals(entity.getAnnotationItem("ServerProperties", "ServerType"));
    }
    public boolean visitEntity(Entity entity)
    {
        if (isLCDSManaged(entity))
        {
            // check if an LCDS entity has aggregated properties
            // loop over associations and verify that each is LCDS-managed.

            // if there was a problem, log it on the BuildContext and return false;
        }
        return true;
    }
}
```

You can extend the modeling language by using annotation validators to add logic to the resolution process. To do so,
you implement an annotation validator and create a definition factory that adds your validator by default to any model
it instantiates. Definition factories must extend the fiber.core.impl.AbstractDefinitionFactory class.

The following example shows a definition factory implementation:

```java
public class MyDefinitionFactory extends DefinitionFactory
{
    public Model newModel()
    {
        Model model = super.newModel();
        model.registerAnnotationValidator(new BodyExprValidator());
        return model;
    }
}
```

The BodyExprValidator object must look for the BodyExpr annotation on functions and methods. Each
AnnotationValidator instance can use a Resolver instance, which provides methods for in-scope resolution of
expressions and types.
public class BodyExprValidator extends AbstractAnnotationValidator
{
    public static final String ID = "BODY_EXPR_VALIDATOR";
    public final Map<Function, Term> bodyTerms = new IdentityHashMap<Function, Term>();
    public String getId() { return ID; }
    public boolean visitFunction(Function function)
    {
        String expr = function.getAnnotationItem("SuperAwesome", "BodyExpr");
        if (expr != null)
        {
            Term exprTerm = getResolver().parseTerm(expr, function.getLocation(),
                                                     getBuildContext());
            if (exprTerm == null || !exprTerm.getType().canCoerceTo(
                function.getSignature().getReturnType()))
            {
                return false;
            }
            // Store the resolved body expression term and set the flag that
            // this function instance will need to redo the custom resolution steps if
            // there are changes to any of its annotations.
            bodyTerms.put(function, exprTerm);
            function.setRequiresAnnotationResolution(true);
        }
        return true;
    }
    public boolean visitMethod(Method method)
    {
        return visitFunction(method);
    }
    public Term getBodyTerm(Function function)
    {
        Term bodyTerm = bodyTerms.get(function);
        return bodyTerms != null ? bodyTerm : function.getBody();
    }
}

With the definition factory and annotation validator in place, you can work with any model that follows your
annotation pattern by relying on the annotation validator instance that is guaranteed to be there. Your code can access
body expressions as the following example shows:

ModelLoader loader = new FMLModelPersistenceFactory().newModelLoader(new
MyDefinitionFactory());
... // set up reader, location and buildContext
Model model = loader.loadModel(reader, location, buildContext);
BodyExprValidator exprValidator = (BodyExprValidator)model.getAnnotationValidator(BodyExprValidator.ID);
...
exprValidator.getBodyTerm(myMethod);

case

Defines a case within a “variant” on page 59, which is a particular collection of child definitions present in an entity
when the selector of the variant yields the value that the case specifies. The legal children of a variant case are the same
as those of a non-persistent entity.
As with top-level child definitions in an entity, expressions in child definitions in a case are subject to scoping rules. In expressions, references to properties whose definitions are separated from the site of reference by variance can require guard terms. For example, suppose you have a derived property from which you want reference another property in a particular case. In the derived property’s expression, you can use the built-in missing function as a guard term for a property in the case, and only use the property when the missing function returns false. Alternatively, you could write an expression that must satisfy specific criteria before using a property in the case.

Modeler Design view
To create a case in the Modeler Design view, drag the case icon from the tools palette onto a variant.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Value Expression sets value expression that specifies constant value that selects this case</td>
</tr>
<tr>
<td></td>
<td>fx button opens Expression Builder</td>
</tr>
<tr>
<td></td>
<td>Annotations:</td>
</tr>
<tr>
<td></td>
<td>Edit Annotations opens Edit Annotations dialog for adding and removing annotation groups and items</td>
</tr>
</tbody>
</table>

Can appear
A case element can appear only as the immediate child of a “variant” on page 59 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td></td>
<td>A value expression that specifies the constant value that selects this case. For example, when the enclosing variant’s selector yields this value, the children of this case are part of the entity instance’s data state. The type of this value must agree with the type of the enclosing variant’s selector expression. Also, the value specified must be unique within the enclosing variant, in the sense that no other cases specify it.</td>
</tr>
</tbody>
</table>

Child elements
A case element can contain zero or more “annotation” on page 11 elements followed by a mix of zero or more “property” on page 39, “constraint” on page 16, “variant” on page 59, and “inline” on page 35 elements.

Example
This example shows an entity with a gift property of type boolean and a giftnote property of type string. The giftnote property is contained in a variant case and is only available when the gift property value is true. As an example of how this variant is useful in a Flex user interface, when you generate a Model-driven Form for this entity in Flash Builder the giftnote field only appears when the gift checkbox is checked.
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Storeorder" persistent="true">
    <id name="ordernumber" type="integer"/>
    <property name="expressshipping" type="boolean"/>
    <property name="gift" type="boolean"/>
    <property name="orderdate" type="date"/>
    <property name="shipdate" type="date"/>
    <variant>
      <selector><![CDATA[gift]]></selector>
      <case>
        <value><![CDATA[true]]></value>
        <property name="giftnote" type="string" length="255"/>
      </case>
    </variant>
  </entity>
</model>

**constraint**

A constraint is a boolean-typed derived property whose value constrains the data state of an entity. The entity is not valid unless the constraint's value expression evaluates to true.

A constraint differs from a “style validation” on page 57 in two ways:

- An expression in a style validation can refer only to the property to which the style is applied.
- You can easily apply a style validation to properties of multiple entities while constraints must be copied and pasted to achieve similar results.

**Modeler Design view**

To create a constraint in the Modeler Design view, drag the constraint icon from the tools palette onto an entity.
Can appear
A constraint element can appear as the immediate child of an “entity” on page 20, “case” on page 14, or “default” on page 18 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies the constraint name. This attribute is required and must be a valid name, and must be unique among properties and constraints of the defining entity.</td>
</tr>
<tr>
<td>expr</td>
<td>Required</td>
<td></td>
<td>Specifies the value expression of the constraint. It is the value expression of a derived property, with the additional requirement that it is of type boolean.</td>
</tr>
<tr>
<td>style</td>
<td>Optional</td>
<td></td>
<td>Specifies the name of style defined in the same model.</td>
</tr>
</tbody>
</table>
Child elements
A constraint element contains zero or more “annotation” on page 11 elements followed by zero or more “case” on page 14 elements, and an optional “default” on page 18 element.

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Address" persistent="true">
    <annotation name="ServerProperties">
      <item name="ServerType">LCDS</item>
    </annotation>
    <annotation name="DMS">
      <item name="Table">Address</item>
    </annotation>
    <annotation name="ActionScriptGeneration">
      <item name="DisplayColumn">streetAddress</item>
    </annotation>
    <id name="id" type="integer"/>
    <property name="streetAddress" type="string" length="255">
      <annotation name="DMS">
        <item name="ColumnName">address</item>
      </annotation>
    </property>
    <property name="state" type="State"/>
    <property name="zipcode" type="string" length="255">
      <annotation name="DMS">
        <item name="ColumnName">zipcode</item>
      </annotation>
    </property>
    <constraint name="ProperAddress">
      <expr>
        <![CDATA[(!missing(streetAddress) and !missing(state))
            or (!missing(zipcode))]}>]
      </expr>
    </constraint>
  </entity>
  <entity name="State" persistent="true">
    <id name="id" type="integer"/>
    <property name="Name" type="string"/>
  </entity>
</model>

default
Defines the default “case” on page 14 within a “variant” on page 59. The default case is the collection of child definitions present in an entity when the selector of the variant element yields a value not specified by any of the variant’s cases.
As with top-level child definitions in an entity, expressions in child definitions in a default case are subject to scoping rules. In expressions, references to properties whose definitions are separated from the site of reference by variance can require guard terms. For example, suppose you have a derived property from which you want reference another property in a default case. In the derived property’s expression, you can use the built-in missing function as a guard term for a property in the case, and only use the property when the missing function returns false. Alternatively, you could write an expression that must satisfy specific criteria before using a property in the case.

**Modeler Design view**

To create a default case in the Modeler Design view, drag the default icon from the tools palette onto a variant.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
</table>
| Annotations  | • Add/remove annotation group  
|              | • Add/remove annotation item   |

**Can appear**

A default element can appear only as the immediate child of a “variant” on page 59 element.

**Attributes**

None.

**Child elements**

A default element can contain zero or more “annotation” on page 11 elements followed by a mix of zero or more “property” on page 39, “constraint” on page 16, “variant” on page 59, and “inline” on page 35 elements.

**Example**

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">  
  <entity name="Storeorder" persistent="true">  
    <id name="ordernumber" type="integer"/>  
    <property name="expressshipping" type="boolean"/>  
    <property name="gift" type="boolean"/>  
    <property name="orderdate" type="date"/>  
    <property name="shipdate" type="date"/>  
    <variant>  
      <selector><![CDATA[gift]]></selector>  
      <case>  
        <value><![CDATA[true]]></value>  
        <property name="giftnote" type="string" length="255"/>  
      </case>  
    </variant>  
    <default>  
      <property name="prop1" type="string"/>  
    </default>  
  </entity>  
</model>
```
entity

Defines an entity within a model. An entity represents a custom data type. For more information, see “Entities” on page 3.

Modeler Design view
To manually create an entity in the Modeler Design view, drag the entity icon from the tools palette onto a model. To automatically generate entities when working with the Model Assembler feature, drag database tables from the RDS data view onto a model.
<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
</table>
| **General**  | **Name** sets name attribute of entity element  
**This Entity Is Persistent** sets persistent attribute of entity element  
**Extra Entity Information:**  
**Display Property** sets DisplayColumn annotation  
**Server Type** sets ServerType annotation  
**Database Information:**  
**DB Table Name** sets Table annotation  
**Use Query Cache** sets use-query-cache annotation  
**Version Column** sets "version" on page 60 element  
**Annotations:**  
**Edit Annotations** opens Edit Annotations dialog for adding and removing annotation groups and items |
| **Code Gen**  | **ActionScript Generation:**  
**Create Offline Adapter** sets GenerateOfflineAdapter annotation  
**Model Driven Form Package** sets MDFPackage annotation  
**Offline Adapter Package** sets OfflineAdapterPackage annotation  
**Package** sets Package annotation  
**Service Package** sets ImplicitServicePackage annotation  
**Java Generation:**  
**Assembler Package** sets AssemblerPackage annotation  
**Generate Assembler** sets GenerateAssembler annotation  
**Package** sets Package annotation |
| **Data Mgmt** | **General Data Management Information:**  
**Cache Items** sets cache-items annotation  
**Delete Conflict Mode** sets delete-conflict-mode annotation  
**Load On Demand** sets load-on-demand annotation  
**Reconnect Strategy** sets reconnect-fetch annotation  
**Update Conflict Mode** sets update-conflict-mode annotation  
**Use Transactions** sets use-transactions annotation  
**Paging Information:**  
**Dynamic Sizing** sets dynamic-sizing annotation  
**Paged Updates** sets paged-updates annotation  
**Page Size** sets page-size annotation  
**Paging Enabled** sets paging-enabled annotation  
**Assembler Information:**  
**Assembler Class** sets AssemblerClass annotation |
Can appear
An entity element can only as an immediate child of a “model” on page 38 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Must be a valid name, and must be unique among entities within the model.</td>
</tr>
<tr>
<td>persistent</td>
<td>Optional</td>
<td>false</td>
<td>Specifies whether the entity is independently persistent. Legal values are <code>true</code> and <code>false</code>. For more information about persistent and non-persistent entities, see “Entities” on page 3.</td>
</tr>
</tbody>
</table>
Child elements
An entity element can contain zero or more server annotations and client annotations followed by a mix of (zero or more) “property” on page 39, “constraint” on page 16, “variant” on page 59, “filter” on page 24, “inline” on page 35, and “method” on page 36 elements. Persistent entities must also contain at least one “id” on page 32 element, and can contain at most one optional “version” on page 60 element.

Examples
The following example shows a simple entity:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

The following example shows an entity with annotations specific to the LiveCycle Data Services Model Assembler feature. The annotations specify the database table and column names that correspond to the entity and its properties. For more information about annotations you can apply to entities, see “Model annotations” on page 62.
A filter in an entity describes a call to a persistence layer that returns a collection of entity instances. A filter translates to a fill method on a LiveCycle Data Services destination. You only use filters with the Model Assembler feature in LiveCycle Data Services. Each persistent entity can specify any number of filters. Non-persistent entities cannot contain filters.

Modeler Design view
To create a filter in the Modeler Design view, select the entity to which you want to add the filter and then select the Filters tab in the Properties view. You can add, remove, and edit filters on the Filters tab. For more information, see “entity” on page 20.
Criteria-based filters
A filter usually specifies a list of criteria and optionally the sort order of the returned collection. This type of filter is called a criteria-based filter. The simplest criteria-based filters are implicit; you only specify them in cases where their names must be different than the ones provided by convention.

You declare criteria-based filters with the following syntax:

```xml
<filter name="[name]" criteria="[criteria]" order="[order_list]"/>
```

In this syntax:

- **name** is the filter name.
- **criteria** is a list of property/operation pairs. In the following example, `lastName` and `age` are properties, while `like` and `>` are operations:
  "lastName like, age >"
- **order** is a list of order specifiers. Order can be **ASC** for ascending or **DESC** for descending. If you omit the **ASC** or **DESC** after a property name, the collection is placed in ascending order. In the following example, `lastName` and `firstName` are both in ascending order:
  "lastName ASC, firstName"

If you omit **ASC** or **DESC** after a property name, items are placed in ascending order.

When working with greater than or less than comparisons in filters, you can use date, string, and numeric types.

You can use any of the following operations in a filter criteria:

- `==`, `eq`, `! =`, `neq`, `>`, `gt`, `>=`, `gte`, `<`, `lt`, `<=`, `lte`, `like`, `not like`, `contains`, `not contains`

If no operation is specified, a default operation is assumed based on the type of the criteria property. The default operation is `==` (same as `eq`) for single-valued properties and `contains` for collection-valued properties. The following table summarizes valid combinations of propRef types and operations:

<table>
<thead>
<tr>
<th>Property type</th>
<th>Allowed operation</th>
<th>JPQL meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection types</td>
<td>contains, not contains</td>
<td>MEMBER OF, NOT MEMBER OF</td>
</tr>
<tr>
<td>String types</td>
<td><code>like</code>, <code>not like</code>, <code>==</code>, <code>eq</code>, <code>! =</code>, <code>neq</code></td>
<td>LIKE, NOT LIKE</td>
</tr>
<tr>
<td>Numeric types</td>
<td><code>==</code>, <code>eq</code>, <code>! =</code>, <code>neq</code>, <code>&gt;</code>, <code>gt</code>, <code>&gt;=</code>, <code>gte</code>, <code>&lt;</code>, <code>lt</code>, <code>&lt;=</code>, <code>lte</code></td>
<td><code>==</code>, <code>! =</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code></td>
</tr>
<tr>
<td>All other types</td>
<td><code>==</code>, <code>eq</code>, <code>! =</code>, <code>neq</code></td>
<td><code>==</code>, <code>! =</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code></td>
</tr>
</tbody>
</table>

**Note:** In criteria expressions, you can escape `_` and `%` characters by using `_\_` and `%\%`.

Pass-through filters
Filters that use an explicit JPQL string are called pass-through filters. Pass-through filter syntax lets you specify more complex queries than criteria-based filter syntax.

You declare pass-through filters with the following syntax:

```xml
<filter name="[name]" arguments="[arg_list]" query="[query_language]:[query_string]"/>
```

In this syntax:

- **name** is the filter name
- **arguments** is a typed list of arguments to the query. For example, "a:string, b:integer".
- **query** is the query language (always use the value `jpql`), followed by a colon and the query string.
Filter query strings use JPQL statements. For more information about JPQL statements, see openjpa.apache.org/builds/1.2.0/apache-openjpa-1.2.0/docs/manual/jpa_langref.html.

Transorms in filters
A transform on an entity is a simple reshaping of the structure of the entity. You apply a transform in a property specifier annotation item in a filter, as the following example shows:

    <filter name="getSomeBooks" criteria="...">
        <annotation name="DMS">
            <item name="propertySpecifier">title,author</item>
        </annotation>
    </filter>

The specified properties and the ID are retrieved when the filter is translated into a fill request on the server. On the client side, the ActionScript generator creates a method based on the filter name. You call that method to call the fill request on the server.

Can appear
A filter element can appear as the immediate child of an “entity” on page 20 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Yes</td>
<td>Filter name.</td>
</tr>
<tr>
<td>criteria</td>
<td>For criteria-based filters</td>
<td>List of property operation pairs.</td>
</tr>
<tr>
<td>order</td>
<td>For criteria-based filters</td>
<td>List of order specifiers.</td>
</tr>
<tr>
<td>arguments</td>
<td>For pass-through filters</td>
<td>Typed list of arguments to the query.</td>
</tr>
<tr>
<td>query</td>
<td>For pass-through filters</td>
<td>Query language (always use the value <code>jpql</code>), followed by a colon and the query string.</td>
</tr>
</tbody>
</table>

Child elements
A filter element can contain zero or more “annotation” on page 11 elements.

Examples
The filter scenarios that follow reference and build on this model:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Book" persistent="true">
    <id name="id" type="integer"/>
    <property name="title" type="string"/>
    <property name="author" type="Author"/>
    <property name="price" type="float"/>
    <property name="chapters" type="Chapter[]"/>
  </entity>
  <entity name="Chapter">
    <property name="title" type="string"/>
    <property name="summary" type="string"/>
  </entity>
  <entity name="Author" persistent="true">
    <id name="id" type="integer"/>
    <property name="firstName" type="string"/>
    <property name="lastName" type="string"/>
  </entity>
  <entity name="BookReview" persistent="true">
    <id name="id" type="integer"/>
    <property name="book" type="Book"/>
    <property name="title" type="string"/>
    <property name="text" type="string"/>
    <property name="score" type="integer"/>
    <property name="date" type="date"/>
  </entity>
</model>

**Using implicit criteria-based filters**

In this implicit filter scenario, a developer wants to build a MXML application that lets users browse and update a library of books. The developer starts with the model above and wants to use LiveCycle Data Services as her server-side data management solution. The developer uses Flash Builder to generate the client-side services and objects representing each entity in the model.

Flash Builder generates the following services, which the developer sees in the Data/Service panel in Flash Builder:

**BookDataService**

getAll()
getById(id:int)
getByTitle(t:string)
getByAuthor(a:AuthorVO)
getByPrice(p:Number)
getByChapter(c:ChapterVO)

**AuthorDataService**

getAll()
getById(id:int)
getByFirstName(fn:string)
getByFirstName(fn:string)
getByFirstName(fn:string)
getByLastName(ln:string)

**BookReviewDataService**
getAll() getById(id:int)
getByBook(b:BookVO)
getByTitle(t:string)
getByText(t:string)
getByScore(s:int)
getByDate(d:Date)

The functions listed are fill functions derived automatically from the implicit filters available on the entities.

Implicit filters on the model let you build a functional data-managed application with no client or server code, and with no knowledge of the model. You can build much of the application in the Flash Builder design view, using dragging gestures to bind visual components (DataGrids and Forms) to the methods of the services. For this sample application, the developer could create a simple master-detail interface on the BookDataService with a DataGrid with a binding to the `getAll()` method and a Form bound to the DataGrid’s `selectedItem` property. In the detail view, the developer can provide a list of other books by the same author, by calling the `myBookDataService.getByAuthor(selectedItem.author)` method. The developer could also list the reviews for the current book by calling the `myBookReviewDataService.getByBook(selectedItem)` method.

**Using declared criteria-based filters**
In this declared filter scenario, a developer wants to build a more functional version of the library application. The developer uses the Modeler to create new filters on some of the entities in the model, as the following code shows:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Book" persistent="true">
    <filter name="getByAuthorLastName" criteria="author.lastName"/>
    <filter name="getByPriceRange" criteria="price &gt;, price &lt;"/>
    <id name="id" type="integer"/>
    <property name="title" type="string"/>
    <property name="author" type="Author"/>
    <property name="price" type="float"/>
    <property name="chapters" type="Chapter[]"/>
  </entity>
  <entity name="Chapter" persistent="true">
    <id name="id" type="integer"/>
    <property name="title" type="string"/>
    <property name="summary" type="string"/>
  </entity>
  <entity name="Author" persistent="true">
    <id name="id" type="integer"/>
    <property name="firstName" type="string"/>
    <property name="lastName" type="string"/>
  </entity>
  <entity name="BookReview" persistent="true">
    <filter name="getByBookOrderByDate" criteria="book" order="date DESC"/>
    <id name="id" type="integer"/>
    <property name="book" type="Book"/>
    <property name="title" type="string"/>
    <property name="text" type="string"/>
    <property name="score" type="integer"/>
    <property name="date" type="date"/>
  </entity>
</model>

When the model is updated, Flash Builder regenerates the BookDataService and BookReviewDataService. The Model Assembler on the LiveCycle Data Services server is reconfigured to support the new fill methods. The new method signatures are derived from the list of properties specified in the filter. For example, `myBookDataService.getByPriceRange(price: Number, price_2: Number)`.

**Using pass-through filters**

In this pass-through filter scenario, a developer adds more advanced features to the library application. As an experienced Java developer, the developer understands the model and the associations between the entities. The developer adds the `getByAvgReviewScore` filter declaration to the Book entity.
When the model change is deployed to the LiveCycle Data Services server, a new named fill is configured on the Book destination. Flash Builder generates the corresponding client-side method. The developer does not write anything other than the JPQL query.

### Using a transform in a filter element

A transform on an entity is a simple reshaping of the structure of the entity. You apply a transform in a property specifier annotation item in an entity filter.

Transforms provide the following benefits:

- Reduce network traffic by transferring only relevant properties over the wire to the client.
- Make it easier for user interface components and client code to deal with a smaller subset of properties of the entity.

You apply a property specifier annotation item on an entity filter, as a part of a DMS annotation. The following example shows a property specifier annotation.
When the model is deployed in Flash Builder, the generated ActionScript code includes an implicit BookDataService with a function named `getSomeBooks()`. When the function is called, the results contain the `title` and `author` properties, as specified, in addition to any `id` properties, which cannot be excluded but do not need to be explicitly written out in the property specifier. Accessing any other property causes an ItemPendingError, unless the property has been fetched previously and is already on the Flex client. A data component or custom code that handles these errors would send a page_items call for the desired missing properties.

**function**

Defines the prototype of an externally defined function within a service. A service element must be an immediate child of the model element. You can create services in the Modeler Source view only.

Functions can be called from entity expressions, which gives model definitions access to external data and functionality using a type mapping mediated through application modeling technology.

**Can appear**

A function element can only appear as an immediate child of a service element.
**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies the function name. Must be a valid name and must be unique across all functions specified in the service, irrespective of signature (overloaded functions and their prototypes are not supported).</td>
</tr>
<tr>
<td>arguments</td>
<td>Optional</td>
<td></td>
<td>Specifies the names and types of the function's arguments. A missing arguments attribute indicates that the method takes zero arguments. If present, the value specified for arguments takes the form name:type, where name and type denote argument names and types, respectively. Names must be valid application modeling language names, and types must be valid application modeling language type expressions.</td>
</tr>
<tr>
<td>return-type</td>
<td>Required</td>
<td></td>
<td>Specifies the return type of the function. Required and must be a valid application modeling language type expression.</td>
</tr>
</tbody>
</table>

**Child elements**

A function element can contain zero or more “annotation” on page 11 elements.

**Example**

The model in the following example contains a service with two functions, both of which are called from expressions in properties of an entity.

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <property name="score" expr="CreditScore.getCreditScore(ssn)"/>
    <property name="risk" expr="CreditScore.getRisk(state, age, score)"/>
    <id name="id" type="integer"/>
  </entity>
  <service name="CreditScore">
    <function name="getCreditScore" arguments="ssn:string" return-type="integer"/>
    <function name="getRisk" arguments="state:string, age:integer, score:integer" return-type="integer"/>
  </service>
</model>
```

**id**

An id property is a data property, the value of which represents part or all of the identity of a persistent entity instance. Persistent entities must specify at least one id property; one or more id elements must appear as top-level child tags of the entity element, following zero or more annotation elements but preceding any other child elements. Non-persistent entities cannot specify any id properties.
For each id property that is of an entity type, the server generates an internal id member variable. The id member variable serves as both the primary key column and the foreign key column. To specify the name of the id member variable, you must add a Column or JoinColumn annotation to the id property.

When working with the Model Assembler feature in LiveCycle Data Services, the id element supports two special annotations: strategy and generator. The possible values for the strategy annotation are AUTO, SEQUENCE, TABLE and IDENTITY. The default value is AUTO. These values map to the identifier generation strategies supported in Hibernate. The value of the generator annotation is a string that identifies the generator defined in the Hibernate configuration.

The following example shows an id element that includes the strategy and generator annotations:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <id name="myid" type="long">
      <annotation name="DMS">
        <item name="strategy">SEQUENCE</item>
        <item name="generator">my_generator</item>
      </annotation>
    </id>
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <property name="score" expr="CreditScore.getCreditScore(ssn)"/>
    <property name="risk" expr="CreditScore.getRisk(state, age, score)"/>
  </entity>
  <service name="CreditScore">
    <function name="getCreditScore" arguments="ssn:string" return-type="integer"/>
    <function name="getRisk" arguments="state:string, age:integer, score:integer" return-type="integer"/>
  </service>
</model>
```

**Modeler Design view**

To create an id in the Modeler Design view, drag the ID icon from the tools palette onto an entity. On the tools palette, you can set an id as Boolean ID, Date ID, Integer ID, Opaque ID, or Text ID.
### Property tab

<table>
<thead>
<tr>
<th>General</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-generated</td>
<td>indicates whether database assigns an id for every item created</td>
</tr>
<tr>
<td>ID Property</td>
<td>indicates property is an id property</td>
</tr>
<tr>
<td>Name</td>
<td>sets name attribute of id element</td>
</tr>
<tr>
<td>Length</td>
<td>sets maximum blob or string length</td>
</tr>
<tr>
<td>Type Information:</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>sets type attribute of id element</td>
</tr>
<tr>
<td>Database Information:</td>
<td></td>
</tr>
<tr>
<td>DB Column Name</td>
<td>sets ColumnName annotation</td>
</tr>
<tr>
<td>Annotations</td>
<td>opens Edit Annotations dialog for adding and removing annotation groups and items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Mgmt</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paging Information:</td>
<td></td>
</tr>
<tr>
<td>Cannot enable these properties for an ID property.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Styles</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Style:</td>
<td></td>
</tr>
<tr>
<td>Create Style</td>
<td>opens &quot;style&quot; on page 49 Properties view; creates new global style</td>
</tr>
<tr>
<td>Edit Style</td>
<td>opens style Properties view for existing global style</td>
</tr>
<tr>
<td>Inline Style:</td>
<td></td>
</tr>
<tr>
<td>Advanced Style Configuration</td>
<td>opens Inline Style Properties dialog</td>
</tr>
<tr>
<td>Caption</td>
<td>sets text value of caption message</td>
</tr>
<tr>
<td>Create Style</td>
<td>opens style Properties view; creates new inline style</td>
</tr>
<tr>
<td>Description</td>
<td>sets text value of description message</td>
</tr>
<tr>
<td>Edit Style</td>
<td>opens style Properties view for existing inline style</td>
</tr>
<tr>
<td>Error Bundle</td>
<td>only available when Use Resource Bundles for Strings is selected; specifies resource bundle and key for localization of error message</td>
</tr>
<tr>
<td>Error Message</td>
<td>sets text value of validation element</td>
</tr>
<tr>
<td>Extends Style</td>
<td>selects style to extend</td>
</tr>
<tr>
<td>Use Resource Bundles For Strings</td>
<td>specifies &quot;style message group&quot; on page 52 uses resource bundle for message localization</td>
</tr>
<tr>
<td>Validation Expr</td>
<td>sets expression of validation element; fx button opens Expression Builder</td>
</tr>
</tbody>
</table>

**Can appear**

An id element can appear only as the immediate child of a persistent "entity" on page 20 element.

**Child elements**

An id element can contain zero or more “annotation” on page 11 elements.
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generated</td>
<td>Optional</td>
<td>true</td>
<td>Specifies whether the database on the server assigns an id for every item created.</td>
</tr>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies the property name. Required and must be a valid name, and must be unique among properties and constraints of the defining entity.</td>
</tr>
<tr>
<td>type</td>
<td>Optional</td>
<td></td>
<td>Specifies the type of the id property's value. Model-driven service providers and code generators are only required to support integer. The type of an id property cannot be float, double, blob, collection, or the entity that it represents. Other types are supported on a best-effort basis.</td>
</tr>
<tr>
<td>length</td>
<td>Optional</td>
<td></td>
<td>When type is string, this attribute, if present, specifies the maximum length of the string value held by the property. If absent, there is no model-specified limit, although consumers of the model could need to impose physical limits.</td>
</tr>
</tbody>
</table>

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <id name="id" type="integer"/>
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <property name="score" expr="CreditScore.getCreditScore(ssn)"/>
    <property name="risk" expr="CreditScore.getRisk(state, age, score)"/>
  </entity>
  <service name="CreditScore">
    <function name="getCreditScore" arguments="ssn:string" return-type="integer"/>
    <function name="getRisk" arguments="state:string, age:integer, score:integer" return-type="integer"/>
  </service>
</model>
```

inline

An inline element adds the child definitions of a specified target entity element to the contents of the inlining entity element. The child definitions of the target entity are present as if declared at the point of the inline element. Only a non-persistent entity element can be the target of an inline element. The annotation elements of the target entity element are not merged with the annotations of the inlining entity element.
Modeler Design view
To add an inline entity in the Modeler Design view, drag the Inline icon from the tools palette onto an entity.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Entity specifies name of non-persistent entity with definitions to inline</td>
</tr>
<tr>
<td></td>
<td>Annotations:</td>
</tr>
<tr>
<td></td>
<td>Edit Annotations opens Edit Annotations dialog for adding and removing annotation groups and items</td>
</tr>
</tbody>
</table>

Can appear
An inline element can appear as the immediate child of an “entity” on page 20, “case” on page 14, or “default” on page 18 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>Required</td>
<td></td>
<td>Specifies the name of the non-persistent entity with definitions to inline.</td>
</tr>
</tbody>
</table>

Child elements
An inline element can contain zero or more “annotation” on page 11 elements.

Example
```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="SingleInline" persistent="true">
    <id name="id1" type="integer"/>
    <id name="id2" type="integer"/>
    <version name="version"/>
    <inline entity="Structure1"/>
  </entity>
  <entity name="Structure1">
    <property name="data1" type="integer"/>
    <property name="derived1" expr="data1"/>
    <constraint name="constraint1" expr="data1 == derived1"/>
  </entity>
</model>
```

method
Defines the prototype of an externally defined method within an entity. For a method to be valid, the signature of the called method and the arguments passed to the invocation must match in number and type.

The major differences between methods and functions is scope. You can only call methods from within derived properties (including constraints) of the entity that contains the methods. You can call functions from within the properties of any entity in the same model.
Modeler Design view
To create a method in the Modeler Design view, select the entity to which you want to add the method and then select the Methods tab in the Properties view. You can add, remove, and edit methods on the Methods tab. For more information, see “entity” on page 20.

Can appear
A method element can only appear as a child of an “entity” on page 20 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies the method name. Required and must be a valid name, and must be unique across all methods specified in the entity, irrespective of signature (overloaded methods and their prototypes are not supported).</td>
</tr>
<tr>
<td>arguments</td>
<td>Optional</td>
<td></td>
<td>Specifies the names and types of the method’s arguments. A missing arguments attribute indicates that the method takes zero arguments. If present, the value specified for arguments takes the form name:type, where name and type denote argument names and types, respectively. Names must be valid application modeling language names, and types must be valid application modeling language type expressions.</td>
</tr>
<tr>
<td>return-type</td>
<td>Required</td>
<td></td>
<td>Specifies the return type of the function. Must be a valid application modeling language type expression.</td>
</tr>
</tbody>
</table>

Child elements
A method element can contain zero or more “annotation” on page 11 elements.

Example
In the following model example, the convertToKilometers() method is called in an expression in a property of the Address entity, which also contains the method. The method is also called in a property of the Customer entity, which has a property of type Address.
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Address" persistent="true">
    <property name="distanceToNYC" type="double"/>
    <property name="distanceToNYCinKM" expr="convertToKilometers(distanceToNYC)"/>
    <id name="id" type="integer"/>
    <method name="convertToKilometers" arguments="miles:double" return-type="double"/>
  </entity>
  <entity name="Customer" persistent="true">
    <property name="name" type="string"/>
    <property name="address" type="Address"/>
    <property name="distance" type="double"/>
    <property name="distanceInKM" expr="address.convertToKilometers(distance)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

Application modeling technology would not be able to evaluate the `convertToKilometers()` method by default. Methods let you specify signatures in the model and then refer to them. When the ActionScript generator creates code from the Address entity, it generates a superclass that represents the entity and a subclass that you can edit. In this case, you would implement the `convertToKilometers()` method in the subclass.

**Note:** Application modeling technology has an extensive built-in function library. Application modeling technology also supports out-of-the-box RemoteObject, WebService, and HTTPService services. When you refer to built-in functions or RemoteObject, WebService, or HTTPService services from a derived property, you do not need to create an implementation.

### model

The model element is the root element of a model. It contains the closed set of definitions that make up the model. For more information, see “About models” on page 2.

**Modeler Design view**

To select a model in the Modeler Design view, click a blank area of the canvas.
Model XML elements

Can appear
The model element can only appear as the root of a model.

Attributes
None.

Child elements
The model element can contain zero or more “annotation” on page 11 elements followed by a mix of zero or more “entity” on page 20, “service” on page 47, and “style” on page 49 elements.

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <id name="id" type="integer"/>
  </entity>
</model>

property
Defines a data property or derived property of an entity. A data property stores typed values. A derived property is a value expression involving other properties of the entity.

Properties are uniquely named within the entity that defines them.
Modeler Design view

To create a property in the Modeler Design view, drag the property icon from the tools palette onto an entity. On the tools palette, you can set a property as Boolean Property, Date Property, Integer Property, Opaque Property, or Text Property. After resetting the property type, you drag the cursor onto an entity without reclicking the property icon.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Auto-generated indicates whether database assigns an id for every item created; only available when ID Property is checked</td>
</tr>
<tr>
<td></td>
<td>ID Property changes property to an &quot;id&quot; on page 32 property when checked</td>
</tr>
<tr>
<td></td>
<td>Name sets name attribute of property</td>
</tr>
<tr>
<td></td>
<td>Required indicates whether property value is required</td>
</tr>
<tr>
<td></td>
<td>Unique for primitive properties only, indicates whether property value is unique for each instance of entity</td>
</tr>
<tr>
<td></td>
<td>Type Information:</td>
</tr>
<tr>
<td></td>
<td>Length sets maximum blob or string length</td>
</tr>
<tr>
<td></td>
<td>This Is A Collection sets property as a collection</td>
</tr>
<tr>
<td></td>
<td>Type sets type attribute of property</td>
</tr>
<tr>
<td></td>
<td>Value Information:</td>
</tr>
<tr>
<td></td>
<td>Auto-recalc sets DisableAutoRecalc annotation</td>
</tr>
<tr>
<td></td>
<td>Value Expression sets value expression of derived property</td>
</tr>
<tr>
<td></td>
<td>Default Value sets default value of a property</td>
</tr>
<tr>
<td></td>
<td>fx button opens Expression Builder</td>
</tr>
<tr>
<td></td>
<td>Database Information:</td>
</tr>
<tr>
<td></td>
<td>Column Name sets ColumnName annotation</td>
</tr>
<tr>
<td></td>
<td>Lazy Load This Property sets lazy annotation for custom types</td>
</tr>
<tr>
<td></td>
<td>Annotations:</td>
</tr>
<tr>
<td></td>
<td>Edit Annotations opens Edit Annotations dialog for adding and removing annotation groups and items</td>
</tr>
</tbody>
</table>
Can appear
A property element can appear as the immediate child of an “entity” on page 20, “case” on page 14, or “default” on page 18 element.

Attributes
Data properties and derived properties share some common attributes and child tags, and each also has definition elements exclusive to it.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
</table>
| Relationship | **Cardinality** sets cardinality of association between property and entity that contains it  
**Join Column(s)** sets JoinColumns annotation  
**Join Tables(s)** sets JoinTables annotation  
**Mapped By** names a property on target entity that holds this entity instance as a value  
**Inverse Join Column(s)** sets InverseJoinColumns annotation |
| Data Mgmt | **Enable Paged Updates** sets enable-paging annotation for collection properties not mapped by another property  
**Page Size** sets page-size annotation for collection properties not mapped by another property |
| Styles | **Global Style:**  
• **Create Style** opens “style” on page 49 Properties view; creates new global style  
• **Edit Style** opens style Properties view for existing global style  
**Inline Style:**  
• **Advanced Style Configuration** opens Inline Style Properties dialog  
• **Caption** sets text value of caption message  
• **Create Style** opens style Properties view; creates new inline style  
• **Description** sets text value of description message  
• **Edit Style** opens style Properties view for existing inline style  
• **Error Bundle and Key** only available when Use Resource Bundles for Strings is selected; specifies resource bundle and key for localization of error message  
• **Error Message** sets text value of validation element  
• **Extends Style** selects style to extend  
• **Use Resource Bundles For Strings** specifies “style message group” on page 52 uses resource bundle for message localization  
• **Validation Expr** sets expression of validation element; fx button opens Expression Builder |
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>(data and derived properties) Specifies the property name. Required and must be a valid name, and must be unique among properties and constraints of the defining entity.</td>
</tr>
<tr>
<td>default</td>
<td>Optional for data properties</td>
<td></td>
<td>Specifies the default value of a property.</td>
</tr>
<tr>
<td>expr</td>
<td>Required for derived properties</td>
<td></td>
<td>(derived properties only) Specifies the value expression of a derived property. Data properties do not accept this attribute. A derived property's value expression is built out of the syntactical elements of the expression syntax, including operators and predefined functions, and can reference other properties and constraints of the entity, provided they are in scope. If no variants are defined in the entity, all properties and constraints are in scope. Otherwise, scope is determined by the variant context in which the value expression is defined, the variant context in which the referenced property is defined, and the guard term within which the reference occurs in the value expression. For more information about variants, see “variant” on page 59. If external functions are prototyped using the function or method element, they can be called within a value expression. For more information, see “function” on page 31 and “method” on page 36.</td>
</tr>
<tr>
<td>type</td>
<td>Required for data properties Optional for derived properties</td>
<td></td>
<td>Specifies the type of the value stored (in the case of data properties) or expressed (in the case of derived properties) by the property. This data type can be a base type, an entity defined within the model, or a collection of another type. The type attribute is optional for derived properties. If it is absent, the type of the derived property is inferred from its value expression, as specified by the expr attribute. If both expr and type are specified, the calculated type of expr must be a subtype of type.</td>
</tr>
<tr>
<td>style</td>
<td>Optional</td>
<td></td>
<td>If present, the style attribute names the style definition to be associated with the property.</td>
</tr>
</tbody>
</table>
Child elements

A property element can contain zero or more “annotation” on page 11 elements. Also, a derived property can specify the expr property as a child tag instead of an attribute.
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Book" persistent="true">
    <id name="id" type="integer"/>
    <property name="title" type="string"/>
    <property name="author" type="Author"/>
    <property name="price" type="float"/>
    <property name="chapters" type="Chapter[]"/>
  </entity>
  <entity name="Chapter" persistent="true">
    <id name="id" type="integer"/>
    <property name="title" type="string"/>
    <property name="summary" type="string"/>
  </entity>
  <entity name="Author" persistent="true">
    <id name="id" type="integer"/>
    <property name="firstName" type="string"/>
    <property name="lastName" type="string"/>
  </entity>
  <entity name="BookReview" persistent="true">
    <id name="id" type="integer"/>
    <property name="book" type="Book"/>
    <property name="title" type="string"/>
    <property name="text" type="string"/>
    <property name="score" type="integer"/>
    <property name="date" type="date"/>
  </entity>
</model>
```

**entity associations**

You specify the relationships or *associations* between entities in the property elements of the associated entities.

The `mappedBy` attribute is allowed only on association properties. It specifies a property on the target entity that holds this entity instance as a value. In other words, this attribute specifies that the property carrying it is expressing the same associative relationship as the property the attribute names, from the opposite direction.

The `cardinality` attribute is also allowed only on association properties. It specifies the type of cardinality of the association between a property and the entity that contains the property. Types of cardinality include one-to-one, one-to-many, many-to-one, and many-to-many.

As an example of how you can define associations, suppose you have persistent User and Account entities. The following one-directional associations are possible. Property 1 defaults to the explicit configuration in property 3. Property 4 defaults to the explicit configuration in property 6.

```xml
<entity name="User" persistent="true">
  ...
  1.  <property name="account" type="Account"/>
  2.  <property name="account" type="Account" cardinality="one-to-one"/>
  3.  <property name="account" type="Account" cardinality="many-to-one"/>
  4.  <property name="accounts" type="Account[]"/>
  5.  <property name="accounts" type="Account[]" cardinality="one-to-many"/>
  6.  <property name="accounts" type="Account[]" cardinality="many-to-many"/>
  ...
</entity>
```
If you have a user property on the Account entity, you can define a bidirectional association with the mappedBy attribute. You cannot use the mappedBy and cardinality attributes on the same property. Therefore, the owning side of a bidirectional association can contain the cardinality attribute (or use the default) and the inverse (mapping) side of the association must contain the mappedBy attribute and cannot contain the cardinality attribute.

Property 5 (one-to-many on the owning side) is not allowed for bidirectional associations. The following example is invalid:

```xml
<entity name="User" persistent="true">
    <property name="accounts" type="Account[]" cardinality="one-to-many"/>
    ...
</entity>
<entity name="Account" persistent="true">
    <property name="user" type="User" mappedBy="accounts"/>
    ...
</entity>
```

The type of the property on the inverse side must agree with the cardinality of the owning side. For example, if the cardinality on the owning side is one-to-one, the inverse must be a single valued property. If the cardinality of the owning side is many-to-one or many-to-many, the inverse must be a collection. You can have what seems like a two-directional association, but if neither side specifies the mappedBy attribute, you end up with two separate associations. Both of these default to cardinality many-to-many because they are collections. However, this does not guarantee that if an Account 1 is in the Accounts collection of User 1, then User 1 must be in the Users collection of Account 1. In other words, these are two independent associations.

The model in the following example contains each type of entity relationship that is supported with the LiveCycle Data Services Model Assembler functionality:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <!-- database connection information for model -->
    <annotation name="DMS">
        <item name="hibernate.connection.datasource">java:comp/env/jdbc/testDB</item>
        <item name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</item>
    </annotation>
    <!-- table name for entity -->
    <entity name="Person" persistent="true">
        <annotation name="ServerProperties">
            <item name="ServerType">LCDS</item>
        </annotation>
        <id name="id" type="long"/>
        <property name="lastName" type="string"/>
        <property name="firstName" type="string"/>
        <property name="phoneNumber" type="string"/>
        <property name="group" type="Group">
            <annotation name="DMS">
                <item name="JoinColumn">groupId</item>
            </annotation>
        </property>
        <property name="affliations" type="Organization[]"/>
    </entity>
    <entity name="Organization" persistent="true">
        <id name="id" type="string"/>
        <property name="members" type="Person[]" mappedBy="affliations"/>
    </entity>
</model>
```
<entity name="Group" persistent="true">
  <annotation name="ServerProperties">
    <item name="ServerType">LCDS</item>
  </annotation>
  <id name="id" type="string"/>
  <property name="name" type="string"/>
  <property name="leader" type="Person" cardinality="one-to-one">
    <annotation name="DMS">
      <item name="JoinColumn">leaderId</item>
    </annotation>
    <item name="JoinColumn">leaderId</item>
  </property>
  <!-- one-to-many inverse of many-to-one -->
  <property name="members" type="Person[]" mappedBy="group"/>
</entity>

<entity name="Husband" persistent="true">
  <id name="ssn" type="integer"/>
  <!-- one-to-one -->
  <property name="wife" type="Wife" cardinality="one-to-one">
    <annotation name="DMS">
      <item name="lazy">false</item>
    </annotation>
  </property>
</entity>

<entity name="Wife" persistent="true">
  <id name="ssn" type="integer"/>
  <!-- one-to-one inverse of one-to-one -->
  <property name="husband" type="Husband" mappedBy="wife"/>
  <!-- one-to-many -->
  <property name="children" type="Child[]" cardinality="one-to-many">
    <annotation name="DMS">
      <item name="lazy">true</item>
    </annotation>
  </property>
</entity>

<entity name="Child" persistent="true">
  <id name="snn" type="integer"/>
</entity>

<!-- one-to-many (own) -->
<!-- many-to-one inverse of one-to-many -->
<!-- not allowed by the mappedBy syntax -->
<!--
<entity name="Troop" persistent="true">
  <id name="name" type="string"/>
  <property name="soldiers" type="Soldier[]" cardinality="one-to-many"/>
</entity>
-->

<!--
<entity name="Soldier" persistent="true">
  <id name="id" type="string"/>
  <property name="troop" type="Troop" mappedBy="soldiers"/>
</entity>
-->
selector

Specifies a value expression that selects among the cases in a variant and possibly a default value. You can specify a selector in an attribute or an element. The Modeler design view inserts a selector element when you create a variant.

Can appear
A selector element can only appear as an immediate child of a “variant” on page 59 element.

Child elements
A selector element contains no child elements.

Example
This example shows an entity with a gift property of type boolean and a giftnote property of type string. The giftnote property is contained in a variant case and is only available when the gift property value is `true`. As an example of how this is variant is useful in a Flex user interface, when you generate a Model-driven Form for this entity in Flash Builder the giftnote field only appears when the gift checkbox is checked.

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <entity name="Storeorder" persistent="true">
        <id name="ordernumber" type="integer"/>
        <property name="expressshipping" type="boolean"/>
        <property name="gift" type="boolean"/>
        <property name="orderdate" type="date"/>
        <property name="shipdate" type="date"/>
        <variant>
            <selector><![CDATA[gift]]></selector>
            <case>
                <value><![CDATA[true]]></value>
                <property name="giftnote" type="string" length="255"/>
            </case>
        </variant>
    </entity>
</model>
```

service

Defines a set of strongly-typed function signatures that represent external functions available to expressions within the model. Services share the same namespace as entities within the model. References to service functions within model expressions must be fully qualified, with the exception of references to built-in functions.

A service element must be an immediate child of the model element.

When an LiveCycle Data Services Data Management Service destination backs an entity, an explicit service declaration is not necessary to describe the destination. The ActionScript generator uses an implicit service to get the information required to generate a DataService class and a wrapper around instances of it from the entity element and its associated filter elements. The entity must include a ServerProperties annotation that specifies LCDS as the server type, as the following example shows:

```xml
<!-- example -->
```
An implicit service has implicit filters from which the ActionScript generator creates fill methods for the properties on the entity.

**Modeler Design view**
The Modeler Design view provides a Services view and corresponding services Properties view in which you can add, edit, and view services.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
</table>
| General      | Name sets service name  
Annotations:  
Edit Annotations opens Edit Annotations dialog for adding and removing annotation groups and items |
| Functions    | Add/remove functions  
Arguments sets function arguments  
Remote Function Name sets remoteFunction annotation  
Return Type sets return-type value of the function  
This Is A Collection indicates whether return type is a collection |

**Can appear**
A service element can only appear as an immediate child of a “model” on page 38 element.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies the service name. This is required and must be a valid name, and must be unique across all services and entities specified in the model.</td>
</tr>
</tbody>
</table>

**Child elements**
A service element can contain zero or more “annotation” on page 11 elements followed by zero or more “function” on page 31 elements.

**Example**
The model in the following example contains a service with two functions, both of which are called from expressions in properties of an entity.
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <service name="CreditScore">
    <function name="getCreditScore" arguments="ssn:string" return-type="integer"/>
    <function name="getRisk" arguments="state:string, age:integer, score:integer" return-type="integer"/>
  </service>
  <entity name="Applicant" persistent="true">
    <id name="id" type="integer"/>
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <property name="score" expr="CreditScore.getCreditScore(ssn)"/>
    <property name="risk" expr="CreditScore.getRisk(state, age, score)"/>
  </entity>
</model>

**style**

Defines a style available within a model. A style is a collection of user-interface-related aspects that can be associated with a property element in the model. You express the association between a property and a style by referencing the style in the `style` attribute of a property element or by declaring the style in-line in the property.

Styles let you reuse user interfaces aspects across multiple properties. A model can include the following aspects:

- "style message group" on page 52, which can contain aspects such as caption, description, error, speak text
- "style edit mask and display mask" on page 56
- "style validation" on page 57

The application modeling technology runtime provides APIs for accessing all aspects of styles. Consumers, such as model-driven user interface generators, leverage these APIs during generation. Generated ActionScript objects also provide APIs for accessing styles. Code can be generated that relies on these APIs for style information at runtime, for example to bind a user interface control attribute to a property that contains a style that varies because it is used in a variant. For more information, see "Style variation with variants" on page 51.

**Note:** The edit-mask and display-mask style aspects use the Adobe picture clause syntax. For more information, see partners.adobe.com/public/developer/en/xml/picture_clause_2.0.pdf.

**Modeler Design view**

To create a style in the Modeler Design view, select the item to which you want to add the style, select the Styles tab in the Properties view, and then select . You can create, remove, and edit styles in the Style Properties view.

Can appear

A style element can appear as an immediate child of a "model" on page 38, "property" on page 39, "variant" on page 59, or "constraint" on page 16 element.
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies a style name. Must be a valid name, and must be unique among styles within the model.</td>
</tr>
</tbody>
</table>

Child elements

A style element can contain zero or more “annotation” on page 11 elements as its first child elements followed by zero or more message, validation, display-mask, or edit-mask elements.

Overriding styles

You can assign aspects from multiple styles to a single property and to inline override/set a particular aspect for a property. To support reuse of aspects from multiple styles, the style element has an `extends` attribute. The `extends` attribute can contain a comma-separated list of names of other styles whose aspects this style inherits. The precedence order is right to left. To support inline style aspect specification, a property element can contain a single nameless style child element. A nameless style can itself extend one or more styles. Inline styles take precedence over extended styles. Since styles can have at most one message aspect with a given name, one edit-mask, and one display-mask, these aspects are overridden when styles extend each other or are specified in line. However, since a style can contain more than one validation aspect, these aspects are compounded during extension.

The following example shows style extension. In this example, the `bidAmount` property gets its caption aspect from the `bidType` style and its validation aspects from both the `positiveInteger` and `lessThanMillion` styles. It overrides the `bidAmount` property’s description aspect with its own specification.

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="lessThanMillion">
    <validation>
      <expr><![CDATA[value < 1000000]]></expr>
    </validation>
  </style>
  <style name="bidAmount" extends="lessThanMillion">
    <message name="caption" bundle="Auction" key="bid_amount"/>
    <message name="description" bundle="Auction" key="bid_amount"/>
  </style>
  <style name="positiveInteger">
    <validation>
      <expr><![CDATA[value > 0]]></expr>
    </validation>
  </style>
  <entity name="AuctionBid" persistent="true">
    <id name="id" type="integer"/>
    <property name="bidAmount" type="integer">
      <style extends="bidAmount, positiveInteger">
        <message name="description" bundle="SilentAuction" key="silent_bid_amount"/>
      </style>
    </property>
  </entity>
</model>
```
Style variation with variants

Similar to derived property expression variation using variants, you can also make a property style vary via variants. Using variants lets you tie style information to entity state, and therefore the generated ActionScript objects. The following example shows a variant used with styles that contain display masks:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="USPostalCode">
    <display-mask>
      <mask>
        <![CDATA['99999-9999']]>  
      </mask>
    </display-mask>
  </style>

  <style name="UKPostalCode">
    <display-mask>
      <![CDATA['AA9A 9AA|AA99 9AA|AA9 9AA|A9A 9AA|A99 9AA|A9 9AA*']]>  
    </display-mask>
  </style>

  <entity name="Address" persistent="true">
    <id name="id" type="integer"/>
    <property name="countryCode" type="string"/>
    <variant>
      <selector><![CDATA[countryCode]]></selector>
      <case><value><![CDATA['US']]]></value>
        <property name="postalCode" type="string" style="USPostalCode"/>
      </case>
      <case><value><![CDATA['UK']]]></value>
        <property name="postalCode" type="string" style="UKPostalCode"/>
      </case>
    </variant>
  </entity>
</model>
```

The following example shows a variant used with a style that contains a validator:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="requiredString">
    <validation>
      <expr><![CDATA[!(missing(value))]]></expr>
    </validation>
  </style>
  <entity name="User" persistent="true">
    <id name="id" type="integer"/>
    <property name="wantSPAM" type="boolean" required="true"/>
    <variant selector="wantSPAM">
      <case value="true">
        <property name="emailAddress" type="string" style="requiredString"/>
      </case>
      <case value="false">
        <property name="emailAddress" type="string"/>
      </case>
    </variant>
  </entity>
</model>

**style message group**

The style message feature lets you reuse message text across an open-ended set of messages. It can also facilitate localization of model-driven generated applications. Additionally, individual generators can specify a set of messages, such as caption, description, and speak text, that they handle strongly.

In the Modeler Design view, you can create and edit messages in the Messages tab of the Style Properties view.

A message element takes the following attributes: bundle, key, tokens, text. Populate either the text attribute or bundle and key attributes. The tokens attribute is optional and lets you tokenize either literal text or bundled messages.

A message aspect consists of bundle, key, tokens, and text attributes. The bundle, key, and tokens are used to look up and tokenize a message against a resource bundle. Text is used as the default for the case where the resource bundle lookup failed, for example, because the bundle did not contain a message for the specified key. Most consumers want to bind to the localized tokenized result and leave it up to the generated code to make the bundle lookup. The generated ActionScript code allows consumers to do that with a bindable message property. For example given the following model elements:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Person" persistent="true">
    <id name="id" type="integer"/>
    <property name="isDoctor" type="boolean"/>
    <variant selector="isDoctor">
      <case value="true">
        <property name="lastName" type="string">
          <style>
            <message name="caption" bundle="PersonMessages" key="doctor prefix"/>
          </style>
        </property>
      </case>
      <case value="false">
        <property name="lastName" type="string">
          <style>
            <message name="caption" bundle="PersonMessages" key="regular prefix"/>
          </style>
        </property>
      </case>
    </variant>
  </entity>
</model>

and an ActionScript label whose text field had the following binding:

```
person.model.lastNameStyle.caption.message
```

The label would start out as "Mrs." if your application's default language is English and if isDoctor is false by default. Then when isDoctor is set to true, the caption would automatically switch to "Dr.". Later, if the user switched languages to French, the caption would automatically switch to "Docteur".

You can specify a literal string or resource bundle and key value names in the style. When code generators process properties associated with this style, they hard code the literal text or use the bundle information to generate a resource bundle call specific to the target language.

The following example shows a simple caption message with a static text value:

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Person" persistent="true">
    <id name="id" type="integer"/>
    <property name="productName" type="string" required="true" length="40">
      <style>
        <message name="caption" text="Product"/>
      </style>
    </property>
  </entity>
</model>
```

In an MXML application, you could bind to the productName property of a generated ActionScript object to display the caption text.

Consider the following properties file and model snippets for working with localization resource bundles:

**Properties file snippet**
# /locale/en_US/Auction.properties
bid_type_caption=Type of Bid
bid_type_description=Select "standard" to make a single bid. Select "incremental" to make a bid that will increase up to its limit in response to other bidders.

Model

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="bid_type">
    <message name="caption" bundle="Auction" key="bid_type_caption"/>
    <message name="description" bundle="Auction" key="bid_type_description"/>
  </style>
  <entity name="AuctionBid" persistent="true">
    <id name="id" type="integer"/>
    <property name="bidType" type="string" style="bid_type"/>
  </entity>
</model>
```

A model-driven form could let the generated ActionScript entity object handle the resource manager invocation with the following code:

```xml
<mx:FormItem id="bidTypeInput" label="{bid._model.bidTypeStyle.caption.message}" toolTip="{bid._model.bidTypeStyle.CustomMessageType.message}"
  <mx:TextInput/>
</mx:FormItem>
```

Generated ActionScript objects have strong type support for the following types of messages:

<table>
<thead>
<tr>
<th>Message type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caption</td>
<td>Use for property labels in user interface components such as forms. Applicable to data properties, derived properties, and constraints.</td>
</tr>
<tr>
<td>Description</td>
<td>Use for property help or hover text in user interface components such as forms. Applicable to data properties, derived properties, and constraints</td>
</tr>
<tr>
<td>Error</td>
<td>Use to specify an error message that is displayed when a constraint fails. Only applicable to constraints; otherwise ignored.</td>
</tr>
</tbody>
</table>

For the following resource bundle:

```
bid_amount_caption=Bid Amount
bid_amount_description=The amount that you'd like to bid cannot exceed your current balance of {0}.
bid_amount_error=Invalid Bid
```

and the following model elements:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="bid_amount">
    <message name="caption" bundle="Auction" key="bid_amount_caption"/>
    <message name="description" bundle="Auction" key="bid_amount_description"/>
    <message name="error" bundle="Auction" key="bid_amount_error"/>
  </style>
  <entity name="AuctionBid" persistent="true">
    <id name="id" type="integer"/>
    <property name="bidAmount" type="integer" style="bid_amount"/>
    <constraint name="validBid" style="bid_amount">
      <expr>
        <![CDATA[bidAmount > 10000]]>
      </expr>
    </constraint>
  </entity>
</model>

The caption and description aspects of the bid_amount style would be applied to the bidAmount property, but the error aspect would be ignored. All three aspects would be applicable to the validBid constraint but user interface generators would most likely not use the caption and description aspects of constraints.

Localized messages and literal text can be tokenized and application modeling technology supports substitution of these tokens with expressions of type string. So if the properties file above was augmented with the following messages:

   bid_amount_caption=Bid Amount
   bid_amount_description=The amount that you'd like to bid cannot exceed \ your current balance of {0}.

and the model was modified to contain the following elements:

<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="bid_type">
    <message name="caption" bundle="Auction" key="bid_type_caption"/>
    <message name="description" bundle="Auction" key="bid_type_description"/>
  </style>
  <style name="bid_amount">
    <message name="caption" bundle="Auction" key="bid_amount_caption"/>
    <message name="description" bundle="Auction" key="bid_amount_description"/>
  </style>
  <entity name="Account" persistent="true">
    <id name="id" type="integer"/>
    <property name="balance" type="integer"/>
  </entity>
  <entity name="AuctionBid" persistent="true">
    <id name="id" type="integer"/>
    <property name="account" type="Account"/>
    <property name="bidType" type="string" style="bid_type"/>
    <property name="bidAmount" type="integer" style="bid_amount">
      <message name="description" tokens="toString(account.balance)"/>
    </property>
  </entity>
</model>

A model-driven form could access the message with the following code:
<mx:FormItem id="bidAmountInput"
    label="{bid._model.bidAmountStyle.caption.message}"
    toolTip="{bid._model.bidAmountStyle.CustomMessageType.message}">
    <mx:TextInput/>
</mx:FormItem>

If you generate a model-driven form for an entity with a property that uses a global style, at run time the form label for that property is replaced with the style's caption text. The default caption message is "Caption." To work around this issue, you can use one of three options:

- Delete the caption message of the global style in Source view
- Use an inline style instead of a global style
- Use a bundle and key instead of the caption text property

### style edit mask and display mask

A style **display mask** lets you specify how a particular set of properties appear while they are displayed in a Flex user interface. A style **edit mask** lets you control how input values for properties are parsed. Style masks are not implemented for client-side code generation in Flash Builder, but are implemented for client-side code generation for Guides in LiveCycle Workbench.

In the Modeler Design view, you can create and edit masks in the Masks tab of the Style Properties view.

**Note:** The edit-mask and display-mask style elements use the Adobe picture clause syntax. For more information, see partners.adobe.com/public/developer/en/xml/picture_clause_2.0.pdf.

The edit-mask and display-mask elements take one attribute, mask, but you can alternatively specify the mask value in the body of the element. The edit-mask specifies how an input value for a property is parsed. It is only applicable to data properties and is ignored for derived properties and constraints.

You can represent a phone number property in a model with a 10-character string type, but suppose you want the user input field to look like the following example:

(   ) -   -

Suppose you also want to make sure that the user interface does not accept anything other than numeric characters. You also want an existing phone number to appear with the format in the following example:

1 -   - .

Edit and display masks let you express this type of scenario with something like the model elements in the following example:
The type of property to which a style that contains a mask attribute is applied to determines the type (date, time, numeric, text). The application modeling language only contains a single date/time type called date. This type encompasses both date and time and you can apply date, time, or combination masks to date properties. You can only apply numeric masks to properties of the application modeling language double, float, long, and integer types. You can only apply text masks to properties of application modeling language string and char types. Masks are ignored for properties of any other application modeling language types (entity, collections, opaque).

**style validation**

The style validation feature lets you validate the value of an entity property. The validity state of an entity is determined by the combination of all constraints and validations.

In the Modeler Design view, you can create and edit style validations in the Validations tab of the Style Properties view.

You can use expressions or Flex validators in style validations. An expression in a style validation element must be of type Boolean. It can reference a special token named `value`, which is replaced by a reference to the property to which the style is applied.

A style validation differs from a "constraint" on page 16 in two ways:

- An expression in a validation aspect can refer only to the property to which the style is applied.
- You can easily apply a style validation to properties of multiple entities while constraints must be copied and pasted to achieve similar results.

The following example shows a style validation applied to a property of an entity:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <style name="priceValidation">
    <validation text="Price needs to be greater than zero">
      <expr><![CDATA[value > 0]]></expr>
    </validation>
  </style>
  <entity name="Applicant" persistent="true">
    <id name="id" type="integer"/>
    <property name="price" style="priceValidation" type="float">
      <annotation name="DMS">
        <item name="ColumnName">price</item>
      </annotation>
    </property>
  </entity>
</model>

A message element used with a validation element takes the following attributes: bundle, key, tokens, text. You populate either the bundle and key attribute, or the text attribute. The tokens attribute is optional when the bundle or text attribute is populated.

**Code generator support for style validation**

Target generation languages can have built-in support for a particular set of validators. For example, the Flex SDK provides the ActionScript-based mx.validators.Validator superclass and its subclasses in the mx.validators package; for more information, see the Flex documentation.

Since the set of validators available in a particular generation target varies significantly, application modeling technology does not attempt to generalize support for such functionality and leaves it as a communication detail between the model and individual generators. This type of validation information is provided in a model annotation and interpreted in a language-specific way by a particular generator.

For example, an ActionScript generator handles the following annotations on a style:

```xml
<property name="socialSecNum" type="string">
  <style>
    <validation>
      <annotation name="ActionScriptValidator">
        <item name="ValidatorClass">mx.validators.SocialSecurityValidator</item>
        <item name="allowedFormatChars">-()"</item>
      </annotation>
    </validation>
  </style>
</property>
```

The ValidatorClass annotation specifies the class of the built-in Flex validator and all other annotations in the group are interpreted as property settings on that validator. The allowedFormatChars property is also set. The values of property settings are interpreted as ActionScript code; to set a string property, you must surround it with quotes and to set a Boolean property you must use true or false without the quotes.

You can configure a Flex validator in the Style Properties view. Select the Validations tab and then select the AS Validator radio button. Select a Flex validator from the dropdown list of validators and enter values for the validator properties you want to use.
variant

Defines a part of an entity for which the structure varies in a precise way. It acts as a conditional property. Structurally, a variant is similar to the switch construct available in most programming languages. An expression associated with the variant, called a selector expression, produces a value that is used to select among a collection of children, with an optional default value available if none of the explicitly specified keys matches the current value of the selector expression.

Unlike a switch, which selects a block of code to run, a variant selects a data substructure from a set of possible choices. This substructure only exists when the entity instance is in a state that produces the specified selector value.

Modeler Design view
To create a variant in the Modeler Design view, drag the variant icon from the tools palette onto an entity.

<table>
<thead>
<tr>
<th>Property tab</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Selector Expression sets expression value</td>
</tr>
<tr>
<td></td>
<td>fx button opens Expression Builder</td>
</tr>
<tr>
<td></td>
<td>Annotations:</td>
</tr>
<tr>
<td></td>
<td>Edit Annotations opens Edit Annotations dialog for adding and removing annotation groups and items</td>
</tr>
</tbody>
</table>

Can appear
A variant element can appear as the immediate child of an “entity” on page 20, “case” on page 14, or “default” on page 18 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>selector</td>
<td>Required</td>
<td></td>
<td>Specifies the value expression that selects among the variant’s cases and possibly a default value. Can be specified in an attribute or an element. The Modeler design view inserts a selector element when you create a variant.</td>
</tr>
</tbody>
</table>

Example
This example shows an entity with a gift property of type boolean and a giftnote property of type string. The giftnote property is contained in a variant case and is only available when the gift property value is true. As an example of how this is variant is useful in a Flex user interface, when you generate a Model-driven Form for this entity in Flash Builder the giftnote field only appears when the gift checkbox is checked.
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Storeorder" persistent="true">
    <id name="ordernumber" type="integer"/>
    <property name="expressshipping" type="boolean"/>
    <property name="gift" type="boolean"/>
    <property name="orderdate" type="date"/>
    <property name="shipdate" type="date"/>
    <variant>
      <selector><![CDATA[gift]]></selector>
      <case>
        <value><![CDATA[true]]></value>
        <property name="giftnote" type="string" length="255"/>
      </case>
    </variant>
  </entity>
</model>

Child elements
A variant element can contain zero or more “annotation” on page 11 elements followed by zero or more “case” on page 14, “default” on page 18, and selector elements. It can specify a selector property in a child tag or an attribute.

version
Defines an integer-typed data property within a persistent entity; its value represents the version of the entity instance. Persistent entities can specify at most one version element. A version element can appear as a top-level child of an entity, following all entity-level annotations. Non-persistent entities cannot specify a version property.

Aside from the integer type, no assumptions are made about the concrete values held by a version property over time. Model-driven service providers and code generators that implement persistence can use whatever strategies make sense. Other model-driven functionality should make no assumptions about the values held by version properties.

Modeler Design view
To create a version element, select an “entity” on page 20 and enter a value in version column field in the General tab of the Properties view.

Child elements
A version element can contain zero or more “annotation” on page 11 elements.

Can appear
A version element can appear as a child of a persistent “entity” on page 20 element.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req/Opt</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required</td>
<td></td>
<td>Specifies the version name.</td>
</tr>
</tbody>
</table>
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="SimpleTest" persistent="true">
    <version name="version1"/>
    <id name="id1" type="integer"/>
    <id name="id2" type="integer"/>
    <property name="data1" type="integer"/>
    <property name="derived1" expr="data1"/>
    <constraint name="constraint1" expr="data1 == derived1"/>
    <property name="data2" type="integer" required="true"/>
    <property name="derived2" expr="data2"/>
    <constraint name="constraint2" expr="data2 == derived2"/>
  </entity>
</model>
```
Chapter 3: Model annotations

Adobe application modeling technology supports annotations for server-side Java code generation for use with the LiveCycle Data Services Model Assembler feature and client-side ActionScript code generation.

Annotations for server-side generation

The following tables list annotations for server-side code generation for the Model Assembler feature, which depends on the LiveCycle Data Services Data Management Service. These annotations result in server-side code generation. For more information about the Model Assembler Feature, see “Model-driven applications” in Using LiveCycle Data Services. For more information about the Data Management Service, see “Introducing the Data Management Service” in Using LiveCycle Data Services.

It is not necessary to include an annotation when using its default value. Annotations are optional unless otherwise noted.

model-level server annotations

The following annotations can appear as child elements of a model element:
<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| DMS   | AssemblerClass      | Specifies a custom assembler to use for a model or an entity. The specified assembler must extend the standard Model Assembler. You can use this annotation at both the model and entity levels in the same model. If there is an annotation on an entity, it is used. If not, the model-level annotation is used. If there is no AssemblerClass annotation on the model or entities, the standard Model Assembler is used. <model xmlns="http://ns.adobe.com/Fiber/1.0">  
  <annotation name="DMS">  
    ...
  </annotation>
  ...
  For more information, see Customizing server-side functionality in *Using LiveCycle Data Services*. |
|       | datasource          | Required. JDBC data source configured through the application server. <model>  
  <annotation name="DMS">  
    <item name="datasource">jdbc/samplesDB</item>
  </annotation>
  ...  
  For more information, see Building your first model-driven application in *Using LiveCycle Data Services*. |
| DMS   | hibernate.dialect   | The Model Assembler feature uses the Hibernate object/relational persistence and query service. <model>  
  <annotation name="DMS">  
    <item name="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</item>
  </annotation>
  ...  
  See the Hibernate configuration guide for possible values. | false         |
## Model annotations

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| DMS   | hibernate.*           | The Model Assembler feature uses the Hibernate object/relational persistence and query service. Set any Hibernate property in an annotation by prefixing the Hibernate property name with hibernate. For example, the following annotation sets the Hibernate connection pool size:  

```xml
<model...>
  <annotation name="DMS">
    <item name="hibernate.connection.poolsize">1
  </item>
</annotation>
```
See the Hibernate configuration guide for possible values. |               |
| DMS   | LikeOperatorCaseSensitive | When true, LIKE clauses in filters are evaluated in a case-sensitive manner. This functionality depends on the capabilities of the database management system you use. It has no effect on the HSQLDB database included in LiveCycle Data Services.  

```xml
<model...>
  <annotation name="DMS">
    <item name="LikeOperatorCaseSensitive">false</item>
  </annotation>
```
true |
| DMS   | ServerGeneratedEntities | If set to false, no entity classes are generated on the server for the entities in the model. It is expected that JPA-annotated entity classes will exist on the server. The class name is the value of the JavaGeneration.Package annotation plus the entity name, or the model name plus the entity name.  

```xml
<model...>
  <annotation name="DMS">
    <item name="ServerGeneratedEntities">false</item>
  </annotation>
```
For more information, see Customizing server-side functionality in Using LiveCycle Data Services. | true |
entity-level server annotations

The following annotations can appear as child elements of an entity element:

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| JavaGeneration | AssemblerPackage | Overrides the JavaGeneration.Package annotation to give you finer control over the package name for assembler skeleton classes when you generate assembler code. Using this annotation at the entity level overrides its use at the model level.  
```xml
<model...>  
  <annotation name="JavaGeneration">  
    <item name= "AssemblerPackage">  
     com.adobe.assemblers  
   </item>  
  </annotation>  
</model>  
```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>For more information, see Customizing server-side functionality in <em>Using LiveCycle Data Services</em>.</th>
<th></th>
</tr>
</thead>
</table>
| JavaGeneration | Package        | Package for Java code when you enable Java code generation in the Modeler. The default package name is the model name. Specifies the default package for generated Java code. Using this annotation at the entity level overrides its use at the model level.  
```xml
<model...>  
  <annotation name="JavaGeneration">  
    <item name= "Package">  
     com.adobe  
   </item>  
  </annotation>  
</model>  
```
|               |               | For more information, see Customizing server-side functionality in *Using LiveCycle Data Services*. |               |
| ServerProperties | Implementation | When you use the Model Assembler, you can specify custom method and function implementations in Implementation annotations.  
```xml
<service name="CreditCheck">  
  <annotation name="ServerProperties" implementation= "myPackage.CreditCheckServerSideImpl "/>  
</service>  
```
|               |               | For more information, see Handling method and function calls.                       |               |
| Group  | Item                    | Description/example                                                                                                                                                                                                                                                                                                                                                     | Default value |
|--------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------ian                                                                                                            |               |
| DMS    | AssemblerClass          | Specifies a custom assembler to use for a model or an entity. The specified assembler must extend the standard Model Assembler. You can use this annotation at both the model and entity levels in the same model. If there is an annotation on an entity, its used. If not, the mode-level annotation is checked and used. If there is no AssemblerClass annotation on the model or entities, the standard Model Assembler is used. <model xmlns="http://ns.adobe.com/Fiber/1.0 ">
  <annotation name="DMS">
  ...
  <item name="AssemblerClass">com.adobe.people.PersonAssembler</item>
  </annotation>
  ...
  For more information, see Customizing server-side functionality in Using LiveCycle Data Services. |               |
| DMS    | auto-sync-enabled       | Controls the value of the client-side DataService.autoSyncEnabled property for clients that are using a Model Assembler destination.                                                                                                                                      | true          |
|        |                         | <entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="auto-sync-enabled">true</item>
  </annotation>
  </entity>
  For more information, see Synchronizing data automatically in Using LiveCycle Data Services. |               |
By default, the Data Management Service caches items returned from `fill()` and `getItem()` calls and uses cached items to implement paging and to build object graphs on the server when implementing lazy associations. This keeps a complete copy of the managed state of all active clients in each server’s memory. You can turn off this caching of memory by setting `cache-items` to `false`.

When the `cache-items` property is `false`, the Data Management Service only caches the id values of the items on the server. Doing so greatly reduces the footprint of data kept on the server, but there is still some memory used on the server for each managed object on each client. To eliminate this overhead entirely, you set `DataService.autoSyncEnabled` to `false` on the client, and either manually refresh clients or use the manual synchronization feature to route changes to clients.

```xml
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="cache-items">false</item>
  </annotation>
...</entity>
```

For more information, see Caching data items in Using LiveCycle Data Services.

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMS</td>
<td>cache-items</td>
<td>By default, the Data Management Service caches items returned from <code>fill()</code> and <code>getItem()</code> calls and uses cached items to implement paging and to build object graphs on the server when implementing lazy associations. This keeps a complete copy of the managed state of all active clients in each server’s memory. You can turn off this caching of memory by setting <code>cache-items</code> to <code>false</code>. When the <code>cache-items</code> property is <code>false</code>, the Data Management Service only caches the id values of the items on the server. Doing so greatly reduces the footprint of data kept on the server, but there is still some memory used on the server for each managed object on each client. To eliminate this overhead entirely, you set <code>DataService.autoSyncEnabled</code> to <code>false</code> on the client, and either manually refresh clients or use the manual synchronization feature to route changes to clients.</td>
<td>true</td>
</tr>
<tr>
<td>DMS</td>
<td>count-security-constraint-ref</td>
<td>Applies the referenced security constraint to the type of request specified (count, create, read, update, or delete).</td>
<td>none</td>
</tr>
<tr>
<td>DMS</td>
<td>create-security-constraint-ref</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>DMS</td>
<td>read-security-constraint-ref</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>DMS</td>
<td>update-security-constraint-ref</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>DMS</td>
<td>delete-security-constraint-ref</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>Group</td>
<td>Item</td>
<td>Description/example</td>
<td>Default value</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>---------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>DMS</td>
<td>cluster-ref</td>
<td>Must match the value of the id attribute of a cluster defined in the services-config.xml file. &lt;entity name=&quot;Book&quot; persistent=&quot;true&quot;&gt; &lt;annotation name=&quot;DMS&quot;&gt; &lt;item name=&quot;cluster-ref&quot;&gt; default-cluster &lt;/item&gt; &lt;/annotation&gt; ... &lt;/entity&gt; For more information, see Clustering in Using LiveCycle Data Services.</td>
<td>default-cluster</td>
</tr>
<tr>
<td>DMS</td>
<td>dynamic-sizing</td>
<td>Lets you avoid potentially expensive count queries. When you set dynamic-sizing to true, the count operation automatically returns -1. With dynamic sizing, the paged fill method of the assembler is called with a startIndex value of 0 and the number of items is set to the pageSize + 1. If the assembler method returns less than the number requested, the size of the fill is known. If it returns the pageSize+1 items requested, pageSize items are returned to the client but the client sets the collection size to pageSize + 1 with one empty slot at the end. When the client requests that empty item, the next pageSize+1 items are requested and the process repeats until the assembler returns less than pageSize+1 items. &lt;entity name=&quot;Book&quot; persistent=&quot;true&quot;&gt; &lt;item name=&quot;dynamic-sizing&quot;&gt; true &lt;/item&gt; ... &lt;/entity&gt; For more information, see Server-to-data-source paging in Using LiveCycle Data Services.</td>
<td>false</td>
</tr>
<tr>
<td>DMS</td>
<td>delete-conflict-mode</td>
<td>Verifies that delete data from a Flex client is not stale. If omitted, no data verification occurs. Valid values are NONE and OBJECT. A data conflict results in a DataConflictEvent on the client. &lt;entity name=&quot;Book&quot; persistent=&quot;true&quot;&gt; &lt;annotation name=&quot;DMS&quot;&gt; &lt;item name=&quot;delete-conflict-mode&quot;&gt; NONE &lt;/item&gt; &lt;/annotation&gt; ... &lt;/entity&gt; For more information, see Configuring destinations to use Hibernate in Using LiveCycle Data Services.</td>
<td>NONE</td>
</tr>
<tr>
<td>Group</td>
<td>Item</td>
<td>Description/example</td>
<td>Default value</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
</tbody>
</table>
| DMS   | page-size       | Sets the default page size for client-to-server paging. When you enable client-to-server paging, the initial fill request from a client causes the assembler associated with a Data Management Service destination to retrieve the entire collection of objects. The server then sends the first page of items to the Flex client. As the client code tries to access ArrayCollection elements that are not resident, additional pages are retrieved from the server.  
```xml
<property name="name" type="string">
  <annotation name="DMS">
    <item name="page-size">5</item>
  </annotation>
  ...
</property>
```
For more information, see Client-to-server paging in Using LiveCycle Data Services. | 0

| DMS   | paged-updates   | Controls how changes are propagated between clients and the server. When you set paged-updates to false (default value), the entire collection is sent when the association value changes. When you set paged-updates to true, only the id properties of the newly added or removed items are sent. If the client uses paged updates to update a collection property, when those changes are committed to the server, the updateCollectionProperty() method on the Assembler interface is called to update the collection property. When association values are paged (page-size value is set to a value greater than zero), paged updates are automatically enabled. However, enabling load-on-demand for associations does not automatically enable paged updates.  
```xml
<property name="name" type="string">
  <annotation name="DMS">
    <item name="paged-updates">false</item>
  </annotation>
  ...
</property>
```
For more information, see Association property loading in Using LiveCycle Data Services. | false
<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| DMS   | paging-enabled    | Enables client-to-server paging. When client-to-server paging is enabled, the initial fill request from a client causes the assembler associated with a Data Management Service destination to retrieve the entire collection of objects. The server then sends the first page of items to the Flex client. As the client code tries to access ArrayCollection elements that are not resident, additional pages are retrieved from the server. <entity name="Book" persistent="true">  
  <annotation name="DMS">  
    <item name="paging-enabled">true</item>  
  </annotation>  
  ...  
</entity> For more information, see Client-to-server paging in Using LiveCycle Data Services. | false         |
| DMS   | reconnect-fetch   | Determines what the client-side DataService object should fetch in the event of a reconnection to the remote destination. There are two options: IDENTITY (just retrieve the sequence id) and INSTANCE (retrieve the complete contents of the fill). <entity name="Book" persistent="true">  
  <annotation name="DMS">  
    <item name="reconnect-fetch">INSTANCE</item>  
  </annotation>  
  ...  
</entity> For more information, see Setting a reconnection policy in Using LiveCycle Data Services. | IDENTITY      |
| DMS   | Table             | Name of database table that corresponds to entity. When you do not use a Table annotation, the entity and table names are the same. You can use a Table annotation to override that behavior. <entity name="Book" persistent="true">  
  <annotation name="DMS">  
    <item name="Table">OrderTable</item>  
  </annotation>  
  ...  
</entity> | Hibernate Type class name |
<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMS</td>
<td>update-conflict-mode</td>
<td>Verifies that update data from a Flex client is not stale. If omitted, no data verification occurs. Valid values are NONE, PROPERTY, and OBJECT. A data conflict results in a DataConflictEvent on the client.</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;entity name=&quot;Book&quot;persistent=&quot;true&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;annotation name=&quot;DMS&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;item name=&quot;update-conflict-mode&quot;&gt;</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/item&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/annotation&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/entity&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see Configuring destinations to use Hibernate in Using LiveCycle Data Services.</td>
<td></td>
</tr>
<tr>
<td>DMS</td>
<td>use-query-cache</td>
<td>Enables use of the Hibernate query cache feature.</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;entity name=&quot;Book&quot;persistent=&quot;true&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;annotation name=&quot;DMS&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;item name=&quot;use-query-cache&quot;&gt;</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>false</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/item&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/annotation&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/entity&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see Configuring destinations to use Hibernate in Using LiveCycle Data Services.</td>
<td></td>
</tr>
<tr>
<td>DMS</td>
<td>use-transactions</td>
<td>By default, LiveCycle Data Services encapsulates client-committed operations in a single J2EE distributed transaction. All client data changes are handled as a unit. If one value cannot be changed, changes to all other values are rolled back. Distributed transactions are supported as long as your Java 2 Enterprise Edition (J2EE) server provides support for J2EE transactions (JTA). For model-driven development, you enable or disable transactions in the use-transactions annotation in an entity.</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;entity name=&quot;Book&quot;persistent=&quot;true&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;annotation name=&quot;DMS&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;item name=&quot;use-transactions&quot;&gt;</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>false</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/item&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/annotation&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/entity&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see Using transactions in Using LiveCycle Data Services.</td>
<td></td>
</tr>
</tbody>
</table>
## DMS

**Item:** shared-backend  

**Description/example:** Configures a shared-backend for a Model Assembler destination when clustering is enabled. When the shared-backend value is true, a cluster node directs the other nodes to broadcast messages to connected clients, but it does not direct the other nodes to reprocess the message.

```xml
<entity name="User"
persistent="true">
  <annotation name="DMS">
    <item name="cluster-ref">default-cluster</item>
    <item name="shared-backend">true</item>
  </annotation>
</entity>
```

For more information, see Configuring a shared back-end system in Using LiveCycle Data Services.

## JavaGeneration

**Item:** AssemblerPackage  

**Description/example:** Overrides the JavaGeneration.Package annotation to give you finer control over the package name for assembler skeleton classes. Using this annotation at the entity level overrides its use at the model level.

```xml
<model...>
  <annotation name="JavaGeneration">
    <item name="AssemblerPackage">com.adobe.assemblers</item>
  </annotation>
</model>
```

For more information, see Customizing server-side functionality in Using LiveCycle Data Services.

**Item:** GenerateAssembler  

**Description/example:** When set to false, overrides the generator-level setting for generating assembler skeleton classes.

```xml
<model...>
  <annotation name="JavaGeneration">
    <item name="GenerateAssembler">false</item>
  </annotation>
</model>
```

For more information, see Customizing server-side functionality in Using LiveCycle Data Services.
The following annotations can appear as child elements of a property element:

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServerProperties</td>
<td>CustomDestinationFlag</td>
<td>You do not manually add this annotation. A CustomDestinationFlag annotation set to true indicates that Flash Builder created the entity after introspecting an existing Data Management Service destination.</td>
<td></td>
</tr>
<tr>
<td>ServerProperties</td>
<td>ServerType</td>
<td>Type of server backing an entity. Required for deploying a model to the LiveCycle Data Services server. Triggers implicit client-side service generation when set to LCDS.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;entity name=&quot;OrderLine&quot; persistent=&quot;true&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;annotation name=&quot;ServerProperties&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;item name=&quot;ServerType&quot;&gt;LCDS&lt;/item&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/entity&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see &quot;Implicit service generation&quot; on page 187 in Using LiveCycle Data Services.</td>
<td></td>
</tr>
<tr>
<td>ServerProperties</td>
<td>Implementation</td>
<td>When you use the Model Assembler, you can specify custom method and function implementations in Implementation annotations.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;entity name=&quot;Application&quot; persistent=&quot;true&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;annotation name=&quot;ServerProperties&quot; implementation=&quot;myPackage.ApplicationMethodsImpl&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/entity&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see Handling method and function calls in Using LiveCycle Data Services.</td>
<td></td>
</tr>
</tbody>
</table>

**property-level server annotations**

The following annotations can appear as child elements of a property element:
<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMS</td>
<td>ColumnName</td>
<td>Name of database column that corresponds to an id or property element.</td>
<td>Hibernate Type property name</td>
</tr>
</tbody>
</table>

```xml
<entity name="Book" persistent="true">
  <annotation name="DMS">
    <item name="ColumnName">
      FullName
    </item>
  </annotation>
  ...
</entity>
```

| DMS   | InverseJoinColumns   | The foreign key columns in the JoinTable (the association table) that point to the child table in an association. Separate multiple column names with commas. |                                  |

```xml
<entity name="User" persistent="true">
  <annotation name="Server Properties">
    <item name=" Server Type">LCDS</item>
  </annotation>
  <id name="userId" type="integer" generated="true" />
  <property name="name" type="string" />
  <property name="accounts" type="Account[]" >
    <annotation name="DMS">
      <item name="InverseJoinColumns">
        AccountID
      </item>
      <item name="JoinTable">UserAccount</item>
      <item name="JoinColumns">UserID</item>
    </annotation>
  </property>
</entity>

<entity name="Account" persistent="true">
  <annotation name="ServerProperties">
    <item name="ServerType">LCDS</item>
  </annotation>
  <id name="accountId" type="string" />
  <property name="accountName" type="string" />
  <property name="users" type="User[]" mappedBy="accounts" />
</entity>
```

| DMS   | InverseJoinColumnReference dNames | When you specify more than one InverseJoinColumns annotation, use the InverseJoinColumnReferencedNames annotation to specify the correct order. |                                  |

```xml
<entity name="User" persistent="true">
  <annotation name="Server Properties">
    <item name=" Server Type">LCDS</item>
  </annotation>
  <id name="userId" type="integer" generated="true" />
  <property name="name" type="string" />
  <property name="accounts" type="Account[]" >
    <annotation name="DMS">
      <item name="InverseJoinColumns">
        AccountID
      </item>
      <item name="JoinTable">UserAccount</item>
      <item name="JoinColumns">UserID</item>
    </annotation>
  </property>
</entity>
```
JoinColumns
(owning side of associations only)

The foreign key columns in the JoinTable (the association table) that point to the parent table. Separate multiple column names with commas.

You should always use JoinColumns annotations when you have a complex association in a model.

```xml
<entity name="User" persistent="true">
  <annotation name="Server Properties">
    <item name="Server Type">LCDS</item>
  </annotation>
  <id name="userId" type="integer" generated="true" />
  <property name="name" type="string" />
  <property name="accounts" type="Account[]" >
    <annotation name="DMS">
      <item name="InverseJoinColumns">
        AccountID
      </item>
      <item name="JoinTable">
        UserAccount
      </item>
      <item name="JoinColumns">
        UserID
      </item>
    </annotation>
  </property>
</entity>
<entity name="Account" persistent="true">
  <annotation name="ServerProperties">
    <item name="ServerType">LCDS</item>
  </annotation>
  <id name="accountId" type="string" />
  <property name="accountName" type="string" />
  <property name="users" type="User[]" mappedBy="accounts" />
</entity>
```

**JoinColumnReferencedName**s

When you use a JoinColumns annotation to specify more than one join column (associated entity has more than one id property), use the JoinColumnReferencedNames if the order in which the JoinColumns are declared is different from the order in which the primary key columns are declared in the associated entity.
### Model annotations

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| DMS   | JoinTable (owning side of associations only) | Name of join table, which is the association table that links associated entities.  
<entity name="User"  
  persistent="true">  
  <annotation name="Server Properties">  
    <item name="Server Type">LCDS</item>  
  </annotation>  
  <id name="userId" type="integer" generated="true" />  
  <property name="name" type="string" />  
  <property name="accounts" type="Account"/>  
    <annotation name="DMS">  
      <item name="InverseJoinColumns">AccountID</item>  
      <item name="JoinTable">UserAccount</item>  
      <item name="JoinColumns">UserID</item>  
    </annotation>  
  </property>  
</entity>  
<entity name="Account" persistent="true">  
  <annotation name="Server Properties">  
    <item name="Server Type">LCDS</item>  
  </annotation>  
  <id name="accountId" type="string"/>  
  <property name="accountName" type="string"/>  
  <property name="users" type="User" mappedBy="accounts"/>  
</entity>  
<entity name="Company" persistent="true">  
  <property name="employee" type="Employee">  
    <annotation name="DMS">  
      <item name="lazy">true</item>  
    </annotation>  
  </property>  
</entity>  
Note the default value is true when working with models and the Model Assembler, but it is false when using configuration files with non-model assemblers.  
For more information, see Association property loading in Using LiveCycle Data Services. | |
| DMS   | lazy (associations only) | To avoid loading entire associated items when the parent object is loaded, the Data Management Service provides the lazy load option. You apply this annotation on the property child elements of an entity element.  
When lazy is set to true, only the id properties of the associated items are loaded with the parent object. The associated items are passed to the client by reference (id property) instead of by value (entire object). When the client attempts to access an associated item for which it only has the id, an ItemPendingError is thrown.  
<entity name="Company" persistent="true">  
  <property name="employee" type="Employee">  
    <annotation name="DMS">  
      <item name="lazy">true</item>  
    </annotation>  
  </property>  
</entity>  
Note the default value is true when working with models and the Model Assembler, but it is false when using configuration files with non-model assemblers.  
For more information, see Association property loading in Using LiveCycle Data Services. | true |
Although using lazy loading lets you avoid retrieving entire associated objects, it can be inefficient for large association collections. To avoid loading an associated collection until it is needed on the client, the Data Management Service provides the load-on-demand option.

When you set the `load-on-demand` item to `true` on an association, the association property value is ignored in the initial `getItem()` and `fill()` methods. When the client first accesses the property, the `getItem()` method is called again for the parent item, and this time the property value is fetched. If `load-on-demand` is enabled and the `lazy` attribute is set to `true` for the association, when the associated property is first accessed only the id properties of the items are fetched and the `getItem()` method is called on each id property as that item is needed. If `load-on-demand` is enabled and the `lazy` attribute is set to `false`, the entire items are fetched when the association property is accessed.

```xml
<entity name="Company" persistent="true">
  <property name="employee" type="Employee">
    <annotation name="DMS">
      <item name="load-on-demand">true</item>
    </annotation>
    ...
  </property>
</entity>
```

For more information, see Association property loading in Using LiveCycle Data Services.

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMS</td>
<td>load-on-demand</td>
<td>(owning side of associations only)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>(owning side of associations only)</td>
<td>Sets the default page size for client-to-server paging. When you enable client-to-server paging, the initial fill request from a client causes the assembler associated with a Data Management Service destination to retrieve the entire collection of objects. The server then sends the first page of items to the Flex client. As the client code tries to access ArrayCollection elements that are not resident, additional pages are retrieved from the server.</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>page-size</td>
<td>(owning side of associations only, when used at property level)</td>
<td></td>
</tr>
</tbody>
</table>
**filter-level server annotations**

The following annotations can appear as child elements of a filter element:

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| DMS   | paged-updates (owning side of associations only) | Controls how changes are propagated between clients and the server. When you set `paged-updates` to `false` (default value), the entire collection is sent when the association value changes. When you set `paged-updates` to `true`, only the id properties of the newly added or removed items are sent. If the client uses paged updates to update a collection property, when those changes are committed to the server, the `updateCollectionProperty()` method on the Assembler interface is called to update the collection property. When association values are paged (`pageSize` value is set to a value greater than zero), paged updates are automatically enabled. However, enabling load-on-demand for associations does not automatically enable paged updates. <pre>&lt;property name="name" type="string"&gt;
  &lt;annotation name="DMS"&gt;
    &lt;item name="paged-updates"&gt;
      false
    &lt;/item&gt;
  &lt;/annotation&gt;
&lt;/property&gt;</pre> For more information, see Association property loading in Using LiveCycle Data Services. | false |
### Model annotations

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
</table>
| DMS   | auto-refresh | The Model Assembler has logic that determines when a fill needs to be refreshed. Setting the auto-refresh item to false overrides that logic so that the Model Assembler never refreshes the fill automatically. If a fill is not refreshed automatically, a client can change the filled collection directly and the changes it makes are propagated to other clients.  

```xml
<entity name="Book" persistent="true">
    ...
    <filter name="getSomeBooks" criteria="...">
        <annotation name="DMS">
            <item name="auto-refresh">true</item>
        </annotation>
    </filter>
</entity>
```

For more information, see The Assembler interface approach in *Using LiveCycle Data Services*.                                                                                                                                                                                                                                                 | true          |
| DMS   | propertySpecifier | Use the propertySpecifier annotation item to create a fill method on the Model Assembler for a query that retrieves a specific set of properties. The ActionScript generator creates a corresponding method on the Flex client side.  

```xml
<entity name="Book" persistent="true">
    ...
    <filter name="getSomeBooks" criteria="...">
        <annotation name="DMS">
            <item name="propertySpecifier">title,author</item>
        </annotation>
    </filter>
</entity>
```

For more information, see Using property specifiers in *Using LiveCycle Data Services*.                                                                                                                                                                                                                                                                                                        | none          |
The following annotations can appear as child elements of an id element:

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMS</td>
<td>page-queries-from-database</td>
<td>When a fill query is resource-intensive or returns more results than are desirable to cache on the server, you can extend paging from the Flex client all the way to the data source. In this case, the assembler retrieves items from the data source one page at a time, and only when the client requests a page. This type of paging is called server-to-data-source paging. It is only enabled if client-to-server paging is enabled. When you use server-to-data-source paging, each time the client requests a page the assembler is asked for that page of items and the items are sent directly to the client. This type of paging is supported whether or not you set the autoSyncEnabled property of the DataService component on the client to true. To enable server-to-data-source paging, set page-queries-from-database to true. &lt;entity name=&quot;Book&quot; persistent=&quot;true&quot;&gt; ... &lt;filter&gt; &lt;annotation name=&quot;DMS&quot;&gt; &lt;item name=&quot;page-queries-from-database&quot;&gt; true &lt;/item&gt; &lt;/annotation&gt; ... &lt;/entity&gt; For more information, see Server-to-data-source paging in Using LiveCycle Data Services.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
Annotations for client-side generation

The ActionScript generator supports a set of model annotations that it uses to generate particular aspects of ActionScript objects in Flash Builder and LiveCycle Workbench. The following tables list annotations for ActionScript code generation.

**model-level client annotations**

The following annotations can appear as child elements of a model element:
<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
</tr>
</thead>
</table>
| ActionScriptGeneration | FullyQualifyReferences    | When set to true, all references to generated entity object classes from generated ActionScript code are fully qualified. This is necessary when the model contains entities with names that collide with the names of top-level ActionScript classes such as the Array class.  
    
    `<model xmlns="http://ns.adobe.com/Fiber/1.0">  
    <annotation name="ActionScriptGeneration">  
    <item name="FullyQualifyReferences">true</item>  
    </annotation>  
    ...`

| ActionScriptGeneration | ImplicitServicePackage   | Default directory for ActionScript code generated from implicit services. If no implicit service package is specified, the default package is used. Using this annotation at the entity level overrides its use at the model level.  
    
    For information about implicit services, see "service" on page 47.  
    
    `<model xmlns="http://ns.adobe.com/Fiber/1.0">  
    <annotation name="ActionScriptGeneration">  
    <item name="ImplicitServicePackage">servicepackage</item>  
    </annotation>  
    ...`

| ActionScriptGeneration | Package                  | Default package for generated ActionScript code. Using this annotation at the entity level overrides its use at the model level.  
    
    `<model xmlns="http://ns.adobe.com/Fiber/1.0">  
    <annotation name="ActionScriptGeneration">  
    <item name="Package">modlev</item>  
    </annotation>  
    ...`
### Model annotations

The following annotations can appear as child elements of an entity element:

<table>
<thead>
<tr>
<th>Group</th>
<th>Item</th>
<th>Description/example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionScriptGeneration</td>
<td>MDFPackage</td>
<td>Default package for generated model-driven Forms. Using this annotation at the entity level overrides its use at the model level.</td>
</tr>
<tr>
<td>ActionScriptGeneration</td>
<td>SuppressServiceGeneration</td>
<td>When set to true, service code generation is suppressed.</td>
</tr>
<tr>
<td>ASChannelSet-channel</td>
<td>channel</td>
<td>Although you can specify a channel set for contacting a LiveCycle Data Services server in a model as described here, it is a best practice to specify a channel set in MXML or ActionScript code in your client application. For more information, see How channels are assigned to a Flex component.</td>
</tr>
</tbody>
</table>

Although you can specify a channel set for contacting a LiveCycle Data Services server in a model as described here, it is a best practice to specify a channel set in MXML or ActionScript code in your client application. For more information, see How channels are assigned to a Flex component.

Creating a ChannelSet in generated ActionScript.

The first parameter of the annotation value specifies the channel type. The second part specifies the first argument to the constructor of the channel object: the channel id. The third part specifies the second argument to the constructor of the channel object: the URI.

You can reference a model-level channel set from a service definition, as the following example shows:

```xml
<service name="Service2">
  <annotation name="ActionScriptGeneration">
    <item name="ServiceType">RemoteObject</item>
    <item name="Package">myCompany.services.service2</item>
    <item name="ChannelSet">CS1</item>
    <item name="Destination">EchoTester</item>
  </annotation>
  <function name="echoString" arguments="toEcho:string" return-type="string"/>
</service>
```
<table>
<thead>
<tr>
<th>Annotation group</th>
<th>item</th>
<th>Description/example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionScriptGeneration</td>
<td>DisableAutoReCalc</td>
<td>Disables automatic recalculation of all derived properties for this entity. The <code>calculate_propertyName()</code> method on a model-level metadata object available for every entity object instance, lets you manually trigger recalculation of derived properties for which automatic recalculation is disabled. For example, you would call <code>myEntityObject._model.calculate_derived1()</code>, to recalculate the derived property named derived1. For more information about generated ActionScript code for derived properties, see “Derived properties in entities” on page 157. You can also use the DisableAutoReCalc item at the property level to apply it to a single derived property.</td>
</tr>
<tr>
<td>ActionScriptGeneration</td>
<td>DisplayColumn</td>
<td>The model-driven Form generator uses the DisplayColumn annotation to pick the property value to show in a Form’s dropdown list.</td>
</tr>
<tr>
<td>ActionScriptGeneration</td>
<td>Dynamic</td>
<td>Determines whether the entity value object class should have the dynamic keyword. An RPCDataManager instance must be notified of new dynamic properties in a class so that the cache schema can account for the new properties in the class. To allow customizations to the generated value object class, the value object class is only generated once. If the dynamic keyword is added after the fact or the superclass must be changed, the generated source must be edited directly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Annotation group</td>
<td>Item</td>
<td>Description/example</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| ActionScriptGeneration | ImplicitServicePackage | Default directory for ActionScript code generated from implicit services. If no implicit service package is specified, the default package is used. Using this annotation at the entity level overrides its use at the model level. For information about implicit services, see “service” on page 47.  
```xml
<entity name="OrderLine" persistent="true">
  <item name="ImplicitServicePackage">
    packagename
  </item>
</annotation> ...
</entity>
```

| ActionScriptGeneration | Package | Default package for generated ActionScript code. Using this annotation at the entity level overrides its use at the model level.  
```xml
<entity name="OrderLine" persistent="true">
    <annotation name="ActionScriptGeneration">
       <item name="Package">packagename</item>
    </annotation>
...</entity>
```

| ActionScriptGeneration | MDFPackage | Default package for generated model-driven Forms. Using this annotation at the entity level overrides its use at the model level.  
```xml
<entity name="OrderLine" persistent="true">
    <annotation name="ActionScriptGeneration">
       <item name="MDFPackage">modlev</item>
    </annotation>
...</entity>
```

| ActionScriptGeneration | SuperClass | Specifies the class that the entity superclass should extend. The class that is assigned to the itemClass property of an RPCDataManager instance must be dynamic to support dynamic schema. This annotation lets you extend the mx.data.DynamicManagedItem class to make the generated class dynamic.  
```xml
<entity name="OrderLine" persistent="true">
    <annotation name="ActionScriptGeneration">
       <item name="SuperClass">mx.data.DynamicManagedItem</item>
    </annotation>
...</entity>
```

| ActionScriptGeneration | UniqueDataStore | Setting this annotation to true for a LiveCycle Data Services backed entity causes the service wrapper to create a new DataStore instance for its service control. The name passed to the DataStore instance is the wrapped destination's name. The UniqueDataStore annotation lets you support multiple model-driven Forms maintaining separate commit message queues. Do not use this annotation on entities that contain associations because all associated entities must share the same DataStore instance.  
```xml
<entity name="OrderLine" persistent="true">
    <annotation name="ActionScriptGeneration">
       <item name="UniqueDataStore"/>
    </annotation>
```
Model annotations

The following annotation can appear as a child element of a property element:

<table>
<thead>
<tr>
<th>Annotation group</th>
<th>Annotation item</th>
<th>Description/example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionScriptGeneration</td>
<td>DisableAutoRecCalc</td>
<td>Disables automatic recalculation for a single derived property of an entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>calculate_propertyName()</code> method on a model-level metadata object available for every entity object instance, lets you manually trigger recalculation of derived properties for which automatic recalculation is disabled. For example, you would call <code>myEntityObject._model.calculate_derived1()</code>, to recalculate the derived property named <code>derived1</code>. For more information about generated ActionScript code for derived properties, see “Derived properties in entities” on page 157.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;property name=&quot;name&quot; type=&quot;string&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;annotation name=&quot;ActionScriptGeneration&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;item name=&quot;DisableAutoRecCalc&quot;&gt;true&lt;/item&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/annotation&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/property&gt;</td>
</tr>
</tbody>
</table>

This setting is useful for derived properties whose expressions are expensive to calculate, which is most likely for expressions that involve a service call.
**service-level client annotations**

The following annotations can appear as child elements of a service element:

<table>
<thead>
<tr>
<th>Annotation group</th>
<th>Annotation item</th>
<th>Description/example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionScriptGeneration</td>
<td>ChannelSet</td>
<td>Reference to a channel set annotation provided at the model level.</td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td>Package for the generated service wrapper as well as the generation location on disc relative to the generation root.</td>
</tr>
<tr>
<td></td>
<td>remoteFunction</td>
<td>Name of a function implementation. Use for names that are reserved in ActionScript, such as toString.</td>
</tr>
<tr>
<td></td>
<td>SuppressServiceGeneration</td>
<td>When set to true, service code generation is suppressed.</td>
</tr>
<tr>
<td>ASChannelSet</td>
<td>channel1-X</td>
<td>Although you can specify a channel set in a model as described here, it is a best practice to specify a channel set in MXML or ActionScript code in your client application. For more information, see “How channels are assigned to a Flex component” in Using LiveCycle Data Services. Anonymous channel set to be used only in this service.</td>
</tr>
</tbody>
</table>

**service function-level client annotations**

The following annotation can appear as a child element of a service function element:
The following annotation can appear as a child element of a style element:

```
<entity name="OrderLine">
  <annotation name="LiveCycleES">
    <item name="generate_type">true</item>
  </annotation>
...
</entity>
```

**Annotations for LiveCycle**

The `generate_type` annotation in the LiveCycleES annotation group is supported at the entity level for models used in the context of LiveCycle ES2 Workbench. Setting `generate_types` to true ensures that the data types that entities define are available to LiveCycle processes. For more information, see *Creating Processes Using LiveCycle Workbench ES2* in the LiveCycle ES2 help.

The following example shows a `generate_type` annotation in an entity:

```
<entity name="OrderLine">
  <annotation name="LiveCycleES">
    <item name="generate_type">true</item>
  </annotation>
...
</entity>
```
Chapter 4: Expression syntax

Derived property, constraint, and variant elements in a model can contain value expressions that use the value of properties in the model as input. Additionally, style validator elements contain value expressions.

Application modeling technology provides a comprehensive set of built-in functions you use in expressions. The Modeler contains an expression editor called the Expression Builder.

*Note:* To add a line break in an expression, use `chr(13)`. The `chr` function is a built-in function that returns a character with the specified ANSI character code. The ANSI character code for a line break is 13. Also note that any value expression that contains the greater than (>) or less than (<) character must be contained in a CDATA section or written as `&gt;` (for greater than) or `&lt;` (for less than). Model examples often show CDATA sections. The Modeler design view creates CDATA sections for all value expressions.

More Help topics
“Built-in functions” on page 112
“Expression Builder” on page 7

Data types

The application modeling language is a strongly typed language. Data types play a role in the definition of application modeling technology constructs. Types must be specified for the following things:

- Each data property of an entity
- The arguments of an entity method
- The return type of an entity method
- The arguments of a service function
- The return type of a service function

You can specify types for derived properties, including constraints. If a type of a property is not specified, it is deduced from the expression in the property declaration. Single character string literals are treated as the char type.

Built-in data types

Application modeling technology supports the following built-in data types:

- blob
- boolean
- char
- date
- Numeric types
- opaque
- string
- void
**boolean**
The boolean type represents a boolean value.

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Storeorder" persistent="true">
    <id name="ordernumber" type="integer"/>
    <property name="expressshipping" type="boolean"/>
    <property name="gift" type="boolean"/>
    <property name="orderdate" type="date"/>
    <property name="shipdate" type="date"/>
    <variant>
      <selector><![CDATA[gift]]></selector>
      <case>
        <value><![CDATA[true]]></value>
        <property name="giftnote" type="string" length="255"/>
      </case>
    </variant>
  </entity>
</model>
```

**blob**
The blob type represents a binary large object. For example, a blob could be an array of bytes. Properties of type blob can have an optional length attribute. The length attribute is not allowed on collections.

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="BlobExample" persistent="true">
    <annotation name="ActionScriptGeneration">
      <item name="Package">foo.blob</item>
    </annotation>
    <id name="id" type="integer"/>
    <property name="name" type="string"/>
    <property name="thing" type="blob"/>
  </entity>
</model>
```

**char**
The char type represents a single character. Collections of chars are coercible to strings.

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="ChrExample" persistent="true">
    <property name="name" type="char[32]"/>
    <property name="preFixedName" type="string" expr="'Mrs. ' + name"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
date
The date type represents a date and includes time at millisecond granularity. It is represented by top-level Date in ActionScript and java.util.Date in Java, both of which include a timestamp.

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date8" expr="getHour(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

Numeric types
Numeric types follow the hierarchy of double > float > long > integer.

Types at the bottom of the hierarchy are subtypes of the types above them and are acceptable arguments to expressions that expect the supertype. For example, an integer is a valid argument to a function that expects a float. However, a double is not a valid argument to a function that expects a long.

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="sampleProp" type="integer"/>
    <property name="math1" expr="abs(sampleProp)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

opaque
The opaque type represents any type that is not expressible in the application modeling language. It is treated as a pass-through type between the application modeling technology runtime and its code generator consumers. Examples of its use are functions that return maps or complex objects that are already represented in target languages and either do not benefit from being expressed or cannot be expressed in the application modeling language due to properties of unsupported types.
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="age" type="integer"/>
    <property name="state" type="string"/>
    <property name="areaCode" type="string"/>
    <property name="nameToPhone" expr="PhoneMap.getNameToPhoneNumberMap(areaCode)="/n"
    <id name="id" type="integer"/>
  </entity>
  <service name="PhoneMap">
    <function name="getNameToPhoneNumberMap" arguments="areaCode:string"
      return-type="opaque"/>
  </service>
</model>
```

**string**
Properties declared to be of type string support a `length` attribute. The value of this attribute effects the persistence storage that backs a property. The `length` attribute is not allowed on collections.

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="state" type="string"/>
    <property name="age" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**void**
Void is an implicit type that is used when the return type of a function or method is not specified. You cannot use the `void` type in any type declaration in a model.

**Type coercion**
Application modeling technology supports type coercion from subtypes to supertypes. Type A is coercible to type B if type A is a subtype of type B. An array of chars is also coercible to a string. A collection of chars is coercible to a string or a string collection.

**Entities**
Entities are definitions of custom types. The name of an entity is a valid type declaration anywhere in the model in which the entity appears.
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="name" type="string"/>
    <property name="ssn" type="string"/>
    <property name="state" type="string"/>
    <property name="age" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

Collections

Application modeling technology has built-in support for collection types. Collection types are declared using the `[]` suffix following the name of a base type. The base type of a collection can be any built-in type or any entity (custom) type.

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="name" type="string"/>
    <property name="scores" type="Score[]"/>
    <property name="max">
      <expr><![CDATA[collectionMax(scores)]]></expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Score" persistent="true">
    <property name="test" type="string"/>
    <property name="grade" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

Types of derived properties

When a model does not explicitly set the `type` attribute of a derived property, the type is determined implicitly from the expression in the `expr` attribute of the derived property.

In the following example, the `salaryTimesTwo` property is of type float:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Employee">
    <property name="salary" type="float"/>
    <property name="salaryTimesTwo" expr="salary * 2"/>
  </entity>
</model>
```

In the following example, the `myName` property is of type string:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Employee">
    <property name="name" expr="'Ed'"/>
  </entity>
</model>
```
In the following example, the `creditScore` property is of type `integer` and the `goodCredit` property is of type `boolean`:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant">
    <property name="ssn" type="string"/>
    <property name="creditScore" expr="CreditCheckService.checkScore(ssn)"/>
    <constraint name="goodCredit" expr="creditScore > 600"/>
  </entity>
  <service name="CreditCheckService">
    <function name="checkScore" arguments="ssn:string" return-type="integer"/>
  </service>
</model>
```

When the type of a derived property is explicitly declared, and the type of its expression is coercible to the declared type, the derived property is legal. The result of the expression is then coerced at runtime. If the type of its expression is not coercible to the declared type, the derived property definition is invalid.

In the following example, the `nameAsString` property is valid because the type of its expression, `char[]`, is coercible to the declared type, `string`:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="DerivedValid">
    <property name="nameAsCharCollection" type="char[]"/>
    <property name="nameAsString" type="string" expr="nameAsCharCollection"/>
  </entity>
</model>
```

In the following example, the `nameAsCharCollection` property in bold is invalid because the type of its expression, `string`, is not coercible to the declared type, `char[]`:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="DerivedInvalid">
    <property name="nameAsString" type="string"/>
    <property name="nameAsCharCollection" type="char[]" expr="nameAsString"/>
  </entity>
</model>
```

### Interpretation of expressions and types by code generators

The application modeling technology runtime defines expected behavior of built-in types and expressions. Custom code generators must conform to these definitions as much as possible. However, situations can arise where, due to target language limitations, conforming to the definitions is impossible.

It is possible to write a code generator that does not conform to application modeling technology definitions and interprets operations differently. However, that generator would not be expected to be compatible with pieces of the framework that conform to the standard application modeling technology definitions.
Expression scope

The scope of expressions is the same when they appear in the expr attribute of a derived property, the expr attribute of a constraint, or the selector attribute of variants. Expressions can refer to any of the following:

- Any property of the containing entity
- Any method of the containing entity
- Any property of an associated entity via a dot reference (chaining permitted)
- Any built-in function
- Any function declared in a service in the same model as the containing entity

Application modeling technology does not support expressions in method or function bodies.

Expression components

You can chain expression components together. For example, you can pass an expression that evaluates to an integer in line as a parameter to a function that expects an integer argument.

Logic operators

The modeling language supports the following types of logic operators:

- and
- equality (==)
- non-equality (!=)
- not (!)
- or
- greater than (>)
- greater than or equal to (>=)
- less than (<)
- less than or equal to (<=)
- switch
- ternary

and

The and operator is a binary operator. Both of its operands must be of type boolean and it produces a boolean result value. The result is true if both of the operands are true.

Boolean and Boolean -> Boolean
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="And1">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <property name="data3" type="string"/>
    <constraint name="myConstraint" expr="data1 > data2 and data3 == 'test'"/>
  </entity>
  <entity name="And2">
    <property name="data1" type="boolean"/>
    <property name="data2" type="boolean"/>
    <property name="derived1" expr="data1 and data2"/>
  </entity>
</model>
```

**equality**

The equality test operator (==) is a binary operator. Its operands must agree in type, and this common type must support equality testing. Entities (custom types) do not support equality testing. Most built-in types do support equality testing, except for opaque, void, and function. Collections are equality testable if their base types are.

\[ T == T \rightarrow \text{Boolean} \]

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Equality1">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <constraint name="myConstraint" expr="data1 == data2"/>
  </entity>
  <entity name="Equality2">
    <property name="data1" type="string[]"/>
    <property name="data2" type="string[]"/>
    <property name="derived1" expr="data1 == data2"/>
  </entity>
</model>
```

**non-equality**

The non-equality test operator (!=) is a binary operator. Its operands must agree in type, and this common type must support equality testing. Entities (custom types) do not support equality testing. Most built-in types do support equality testing, except for opaque, void, and function. Collections are equality testable if their base types are equality testable.

\[ T != T \rightarrow \text{Boolean} \]
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Nonequality1">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <constraint name="myConstraint" expr="data1 != data2"/>
  </entity>

  <entity name="Nonequality2">
    <property name="data1" type="string[]"/>
    <property name="data2" type="string[]"/>
    <property name="derived1" expr="data1 != data2"/>
  </entity>
</model>
```

**not**
The not operator (!) is a unary operator that can only be applied to boolean expressions.

! Boolean -> Boolean

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Not1">
    <property name="age" type="integer"/>
    <constraint name="ofAge">
      <expr>
        <![CDATA[!(age < 21)]]>  
      </expr>
    </constraint>
  </entity>
</model>
```

**or**
The or operator is a binary operator. Both of its operands must be of type boolean and it produces a boolean result value. The result is true if at least one of the operands is true.

Boolean or Boolean -> Boolean

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Or1">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <property name="data3" type="string"/>
    <constraint name="myConstraint" expr=" data1 > data2 or data3 == 'test'"/>
  </entity>
  <entity name="Or2">
    <property name="data1" type="boolean"/>
    <property name="data2" type="boolean"/>
    <property name="derived1" expr="data1 or data2"/>
  </entity>
</model>
```
greater than
The greater than operator (>) is a binary operator. Its operands must agree in type, and this common type must support order testing. Entities (custom types) do not support order testing. Most built-in types do support order testing, except for the opaque, void, and function types.

T > T -> Boolean

Note: Expressions that contain the greater than (> or less than (<) operator must be contained in a CDATA section or written as &gt; (for greater than) or &lt; (for less than). The Modeler design view creates CDATA sections for all value expressions.

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="GreaterThan1">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <constraint name="myConstraint">
      <expr>
        <![CDATA[data1 > data2]]>
      </expr>
    </constraint>
  </entity>
  <entity name="GreaterThan2">
    <property name="data1" type="string[]"/>
    <property name="data2" type="string[]"/>
    <property name="derived1">
      <expr>
        <![CDATA[data1 > data2]]>
      </expr>
    </property>
  </entity>
</model>

greater than or equal to
The greater than or equal to operator (>=) is a binary operator. Its operands must agree in type, and this common type must support order testing. Entities (custom types) do not support order testing. Most built-in types do support order testing, except for opaque, void, and function.

T >= T -> Boolean

Note: Expressions that contain the greater than (> or less than (<) operator must be contained in a CDATA section or written as &gt; (for greater than) or &lt; (for less than). The Modeler design view creates CDATA sections for all value expressions.
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="GreaterEqual1" persistent="true">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <constraint name="myConstraint">
      <expr><![CDATA[data1 >= data2]]></expr>
    </constraint>
    <id name="id" type="integer"/>
  </entity>

  <entity name="GreaterEqual2" persistent="true">
    <property name="data1" type="string[]"/>
    <property name="data2" type="string[]"/>
    <property name="derived1">
      <expr><![CDATA[data1 >= data2]]></expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**less than**

The less than operator (<) is a binary operator. Its operands must agree in type, and this common type must support order testing. Entities (custom types) do not support order testing. Most built-in types do support order testing, except for opaque, void, and function.

\[ T < T \rightarrow \text{Boolean} \]

*Note:* Expressions that contain the greater than (>) or less than (<) operator must be contained in a CDATA section or written as &gt; (for greater than) or &lt; (for less than). The Modeler design view creates CDATA sections for all value expressions.
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="LessThan1" persistent="true">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <constraint name="myConstraint">
      <expr>
        <![CDATA[data1 < data2]]>
      </expr>
    </constraint>
    <id name="id" type="integer"/>
  </entity>

  <entity name="LessThan2" persistent="true">
    <property name="data1" type="string[]"/>
    <property name="data2" type="string[]"/>
    <property name="derived1">
      <expr>
        <![CDATA[data1 < data2]]>
      </expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
</model>
```

less than or equal to

The less than or equal to operator (<=) is a binary operator. Its operands must agree in type, and this common type must support order testing. Entities (custom types) do not support order testing. Most built-in types do support order testing, except for opaque, void, and function.

T <= T -> Boolean

**Note:** Expressions that contain the greater than (>) or less than (<) operator must be contained in a CDATA section or written as &gt; (for greater than) or &lt; (for less than). The Modeler design view creates CDATA sections for all value expressions.
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="LessThan1" persistent="true">
    <property name="data1" type="integer"/>
    <property name="data2" type="integer"/>
    <constraint name="myConstraint">
      <![CDATA[data1 < data2]]>
    </expr>
  </constraint>
  <id name="id" type="integer"/>
</entity>

  <entity name="LessThan2" persistent="true">
    <property name="data1" type="string[]"/>
    <property name="data2" type="string[]"/>
    <property name="derived1">
      <![CDATA[data1 <= data2]]>
    </expr>
  </property>
  <id name="id" type="integer"/>
</entity>
</model>
```

**switch**

You can use the switch operator to branch the evaluation of an expression based on testing an expression against one or more literal values. The result of a ternary operation can be any type but the type returned by the cases of the switch must agree. The switch operator supports a default clause. The default clause is required if the type of the tested expression is infinite. The default clause is also required if its type is finite, but the selectors of the switch do not exhaust all of its finite values. All the selector literals must agree in type with the selector expression. No selector literal can appear more than once.

Expression1_of_Type1 ? [Selector_Literal_Type1_n : Expression_n_of_Type2, Default_of_Type2]

**Example**

In the first example, the default clause is required and present because string is an infinite type. In the second example, no default clause is necessary because boolean is a finite type of cardinality two, and both values are present in the switch.
ternary
You can use the ternary operator to branch the evaluation of an expression based on a Boolean test. The result of a ternary operation can be any type. However, the type returned in the true case must agree with the type returned in the false case.

Boolean ? Expression1_of_Type1 : Expression2_of_Type1

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Ternary1" persistent="true">
    <property name="age" type="integer"/>
    <property name="discount" expr="age > 21 ? 'None' : 'Student'"/>
    <property name="homePrice" type="integer"/>
    <property name="requiredDownpayment">
      <expr>
        <![CDATA[homePrice > 500000 ? homePrice * 0.2 : homePrice * 0.1]]>
      </expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
</model>

Reference operators
The modeling language supports the following types of reference operators:
- dot (.) reference
- invocation reference
- collection reference
- this reference
**dot reference**

You can use a dot reference to access properties of associated entities and functions of available services. As with other expressions, you can chain dot references together. You use the following syntax for dot references:

- `PropertyOfEntityType.PropertyOfThatEntity`
- `NameOfService.NameOfFunctionInThatService`

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Company" persistent="true">
    <property name="name" type="string"/>
    <property name="state" type="State"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Employee" persistent="true">
    <property name="employer" type="Company"/>
    <property name="employerName" expr="employer.name"/>
    <constraint name="worksInMass" expr="employer.state.code == 'MA'"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="State" persistent="true">
    <property name="code" type="string"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Applicant" persistent="true">
    <property name="ssn" type="string"/>
    <constraint name="goodCredit">
      <expr><![CDATA[CreditCheckService.checkScore(ssn) > 600]]></expr>
    </constraint>
    <id name="id" type="integer"/>
  </entity>
  <service name="CreditCheckService">
    <function name="checkScore" arguments="ssn:string" return-type="integer"/>
  </service>
</model>
```

**invocation references**

Application modeling technology supports invocation references to the following types of targets:

- Custom methods defined in an entity
- Built-in functions
- Custom functions defined in a service

**Invocation precedence and namespaces**

Methods are always non-qualified references. Built-in functions are part of an implicit service named `Builtin`. Generally, you do not need to specify the service name when you use a built-in function in an expression. The exception is when an entity contains a property with the same name as a built-in function that you want to use. In that case, use dot notation to prefix the built-in function name with the service name `Builtin`, as the following example shows:
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Aplication" persistent="true">
    <property name="created" type="date"/>
    <property name="dayOfWeek" type="integer"/>
    <property name="start" expr="Builtin.dayOfWeek(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

**Invocation details**

For a method or function invocation to be valid, the signature of the invoked method or function, and the arguments passed to the invocation, must match in number and type. The major differences between methods and functions is scope. You can only invoke methods from within a derived properties (including constraints) of the entity that contains the methods. You can invoke functions from within the properties of any entity in the same model.

A `this` reference is in the scope of method bodies but not of function bodies. You cannot specify method and function bodies in a model. Instead, provide implementations for desired generation targets (for example, Java or ActionScript). In some cases, the bodies can be fully generated with the help of annotations.

Methods and functions take one of the following forms:

- `MethodName(method arguments)`
- `Built-In-FunctionName(function arguments)`
- `ServiceName.FunctionName(function arguments)`
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <service name="CreditCheckService">
    <function name="checkScore" arguments="ssn:string" return-type="integer"/>
  </service>
  <entity name="Applicant" persistent="true">
    <property name="ssn" type="string"/>
    <constraint name="goodCredit" expr="CreditCheckService.checkScore(ssn) > 600"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Employee" persistent="true">
    <property name="salary" type="float"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Company" persistent="true">
    <property name="employees" type="Employee[]"/>
    <method name="fractionOfTotalSalary" arguments="factor:double" return-type="float"/>
    <constraint name="withinBudget">
      <!CDATA[fractionOfTotalSalary(0.8) < 1000]>
    </constraint>
    <property name="birthdate" type="date"/>
    <property name="ofAge">
      <!CDATA[years(today(), birthdate) > 21]>
    </property>
    <id name="id" type="integer"/>
  </entity>
</model>
```

Collection reference

Application modeling technology supports indexed references into collections. The index must be an integer expression.

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Employee" persistent="true">
    <property name="salary" type="float"/>
    <property name="monthlyBonuses" type="float[]"/>
    <property name="julyBonus" expr="monthlyBonuses[7]"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Company" persistent="true">
    <property name="employees" type="Employee[]"/>
    <property name="firstEmployee" expr="employees[0]"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
**this reference**
The word `this` is an application modeling technology keyword that denotes a reference to the entity in the current scope.

Example
```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Employee" persistent="true">
    <property name="salary" type="float"/>
    <property name="monthlyBonuses" type="float[]"/>
    <property name="julyBonuses" expr="this.monthlyBonuses[7]"/>
    <id name="id" type="integer"></id>
  </entity>
</model>
```

**Operator of precedence**
The following list shows the order of precedence in a compound, unparenthesized expression. The list is ordered from highest precedence to lowest precedence. Operators listed on the same line have equal precedence. There is never be any ambiguity between operators of equal precedence.

1. dot reference, this reference, invocation, collection reference, missing
2. unary minus, not
3. multiplication, division, modulo
4. addition, subtraction
5. greater than, greater than or equal to, less than, less than or equal to
6. equality, non-equality
7. and
8. or
9. switch, ternary

**Literal expressions**
Application modeling technology supports specific syntax for the following literal types:

- boolean literals
- collection literals
- double literals
- float literals
- function literals
- integer literals
- string literals

**boolean literal**
Application modeling technology supports the values `true` and `false` as boolean literals.
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Applicant" persistent="true">
    <property name="ssn" type="string"/>
    <constraint name="goodCredit">
      <expr><![CDATA[CreditCheckService.checkScore(ssn, false) > 600]]></expr>
    </constraint>
    <id name="id" type="integer"/>
  </entity>
  <service name="CreditCheckService">
    <function name="checkScore" arguments="ssn:string, single:boolean" return-type="integer"/>
  </service>
</model>
```

collection literal
Application modeling technology supports collection literals as a mechanism for specifying the values of a collection inline. The collection values themselves do not necessarily have to be literals, that is they can be complex expressions.

Another way to think about this is that collection literals define final collections in terms of size and member expressions. These collections cannot be modified later. However, the values of their members can vary if expressions are used because these expressions can evaluate to different values at different times.

```
<Expression_1_OfBaseType, Expression_2_OfBaseType, Expression_n_OfBaseType]
```

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <service name="HelperFunctions">
    <function name="intArraySum" arguments="arg1:integer[]" return-type="integer"/>
  </service>
  <entity name="Employee" persistent="true">
    <property name="quickSum" expr="HelperFunctions.intArraySum([1, 2])"/>
    <id name="id" type="integer"/>
  </entity>
  <service name="HelperFunctions2">
    <function name="intArraySum" arguments="arg1:integer[]" return-type="integer"/>
  </service>
  <entity name="Employee2" persistent="true">
    <property name="salary" type="integer"/>
    <property name="quickSum" expr="HelperFunctions2.intArraySum([salary, salary * 2])"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

double literal
Application modeling technology expressions treat numbers with a decimal point as double literals.
**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Company" persistent="true">
    <property name="size" expr="100.0"/>
    <id name="id" type="integer"/></id>
  </entity>
</model>
```

**float literal**

Float literals are denoted with either a "f", "F", "d", or "D" suffix.

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Company" persistent="true">
    <property name="size" expr="100.0f"/>
    <id name="id" type="integer"/></id>
  </entity>
</model>
```

**function literal**

Application modeling technology supports inline representation of function signatures together with their bodies. However, no generators currently support function literals. The goal is for version 2 to support these expressions.

```xml
{ArgumentList | Expression}
```

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Employee" persistent="true">
    <property name="myDoubleFunction" expr="{arg1:integer | arg1*2}"/>
    <id name="id" type="integer"/></id>
  </entity>

  <entity name="Employee2" persistent="true">
    <property name="myDoubleFunction" expr="{arg1:integer | arg1*2}(5)"/>
    <id name="id" type="integer"/></id>
  </entity>
</model>
```

**integer literal**

Application modeling technology expressions treat numbers without decimal points as integer literals.

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Company" persistent="true">
    <property name="size" expr="100"/>
    <id name="id" type="integer"/></id>
  </entity>
</model>
```

**string literal**

You can surround string literals with single or double quotes.
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Company" persistent="true">
    <property name="name" expr="'Adobe'"/>
    <id name="id" type="integer"/>
  </entity>

  <entity name="Company2" persistent="true">
    <property name="name" expr="'Adobe'"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**Built-in functions**

Application modeling technology includes a set of built-in functions to use at runtime. These standard functions are treated as a collection of static utility functions. They are not operations on objects. You use built-in functions in entity property elements to produce derived properties, with inputs that come from data properties in the model.

**More Help topics**

“Built-in functions” on page 112

**Assignment**

You can only use a single, possibly compound, expression to define derived properties and variant selectors. Application modeling technology does not support assignment operators because there is no place where multiple sequential expressions would appear.
Chapter 5: Built-in functions

The application modeling language includes a set of built-in functions to use in expressions. For more information about expression syntax, see “Expression syntax” on page 91. The Modeler contains an expression editor called the expression builder; for more information, see “About the Modeler” on page 5.

These standard functions are treated as a collection of static utility functions. They are not operations on objects. You use them in entity property elements to produce derived properties, with inputs that come from data properties in the model.

The following example shows an entity with two properties. The first property is a data property named `created`, which is of type `date`. The second property is a derived property named `date8`. The `date8` property contains an expression that returns the hour from the `created` property:

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date8" expr="getHour(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

The built-in functions are part of an implicit service named Builtin. Generally, you do not need to specify the service name when you use a built-in function in an expression. The exception is when an entity contains a property with the same name as a built-in function you want to use. In that case, use dot notation to prefix the built-in function name with the service name Builtin, as the following example shows:

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="dayOfWeek" type="integer"/>
    <property name="start" expr="Builtin.dayOfWeek(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

Functions by category

The following tables list functions by their category or purpose.

### Collection functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collectionAvg</code></td>
<td>116</td>
</tr>
<tr>
<td><code>isEmpty</code></td>
<td>138</td>
</tr>
<tr>
<td><code>collectionMax</code></td>
<td>116</td>
</tr>
<tr>
<td><code>part</code></td>
<td>142</td>
</tr>
<tr>
<td><code>collectionMin</code></td>
<td>117</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>148</td>
</tr>
<tr>
<td><code>collectionSum</code></td>
<td>118</td>
</tr>
</tbody>
</table>
### Built-in functions

#### Conversion functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>toDouble</code></td>
<td>151</td>
</tr>
<tr>
<td><code>toLong</code></td>
<td>152</td>
</tr>
<tr>
<td><code>toFloat</code></td>
<td>152</td>
</tr>
<tr>
<td><code>toString</code></td>
<td>153</td>
</tr>
<tr>
<td><code>toInteger</code></td>
<td>152</td>
</tr>
</tbody>
</table>

#### Date and time functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createDate</code></td>
<td>119</td>
</tr>
<tr>
<td><code>createDateUTC</code></td>
<td>120</td>
</tr>
<tr>
<td><code>dateAdd</code></td>
<td>121</td>
</tr>
<tr>
<td><code>dateCompare</code></td>
<td>121</td>
</tr>
<tr>
<td><code>dateDiff</code></td>
<td>122</td>
</tr>
<tr>
<td><code>datePart</code></td>
<td>123</td>
</tr>
<tr>
<td><code>datePartUTC</code></td>
<td>124</td>
</tr>
<tr>
<td><code>dayOfYear</code></td>
<td>125</td>
</tr>
<tr>
<td><code>daysInMonth</code></td>
<td>125</td>
</tr>
<tr>
<td><code>daysInYear</code></td>
<td>126</td>
</tr>
<tr>
<td><code>getDay</code></td>
<td>131</td>
</tr>
<tr>
<td><code>getDayUTC</code></td>
<td>127</td>
</tr>
<tr>
<td><code>getHour</code></td>
<td>132</td>
</tr>
<tr>
<td><code>getHourUTC</code></td>
<td>132</td>
</tr>
<tr>
<td><code>getMinute</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getMinuteUTC</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getMonth</code></td>
<td>134</td>
</tr>
<tr>
<td><code>getMonthUTC</code></td>
<td>134</td>
</tr>
<tr>
<td><code>getQuarter</code></td>
<td>135</td>
</tr>
<tr>
<td><code>getSecond</code></td>
<td>135</td>
</tr>
<tr>
<td><code>getSecondUTC</code></td>
<td>136</td>
</tr>
<tr>
<td><code>getYear</code></td>
<td>136</td>
</tr>
<tr>
<td><code>getYearUTC</code></td>
<td>137</td>
</tr>
<tr>
<td><code>isLeapYear</code></td>
<td>138</td>
</tr>
<tr>
<td><code>now</code></td>
<td>142</td>
</tr>
<tr>
<td><code>weekOfYear</code></td>
<td>154</td>
</tr>
</tbody>
</table>

#### Mathematical functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>114</td>
</tr>
<tr>
<td><code>ceiling</code></td>
<td>114</td>
</tr>
<tr>
<td><code>floor</code></td>
<td>130</td>
</tr>
<tr>
<td><code>fix</code></td>
<td>130</td>
</tr>
<tr>
<td><code>sgn</code></td>
<td>148</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max</code></td>
<td>140</td>
</tr>
<tr>
<td><code>min</code></td>
<td>141</td>
</tr>
<tr>
<td><code>round</code></td>
<td>147</td>
</tr>
<tr>
<td><code>sgn</code></td>
<td>148</td>
</tr>
</tbody>
</table>

#### String functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>charAt</code></td>
<td>115</td>
</tr>
<tr>
<td><code>chr</code></td>
<td>115</td>
</tr>
<tr>
<td><code>compare</code></td>
<td>118</td>
</tr>
<tr>
<td><code>compareNoCase</code></td>
<td>119</td>
</tr>
<tr>
<td><code>endsWith</code></td>
<td>126</td>
</tr>
<tr>
<td><code>find</code></td>
<td>127</td>
</tr>
<tr>
<td><code>findNoCase</code></td>
<td>127</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>144</td>
</tr>
<tr>
<td><code>replaceAll</code></td>
<td>144</td>
</tr>
<tr>
<td><code>replaceAllNoCase</code></td>
<td>145</td>
</tr>
<tr>
<td><code>replaceAll</code></td>
<td>144</td>
</tr>
<tr>
<td><code>replaceAllNoCase</code></td>
<td>145</td>
</tr>
<tr>
<td><code>removeChars</code></td>
<td>143</td>
</tr>
<tr>
<td><code>repeatString</code></td>
<td>143</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>146</td>
</tr>
</tbody>
</table>
abs

number abs(number)

Returns the absolute value of a number. The return type matches the input type.

Parameters

| number | Number for which to return the absolute value. |

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="sampleProp" type="integer"/>
    <property name="math1" expr="abs(sampleProp)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

ceiling

number ceiling(number)

Returns the closest integer that is greater than a specified number. Return type matches input type.

Parameters

| number | Number for which to find the closest integer that is greater. |

See also

“floor” on page 130
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="sampleProp" type="integer" />
    <property name="math2" expr="ceiling(sampleProp)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**charAt**

`char charAt(string, integer)`

Returns the character at the indicated position.

Generally, when a string function, such as the `charAt` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>string</th>
<th>Character for which to find position.</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>Position of character.</td>
</tr>
</tbody>
</table>

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="sampleProp" type="string" />
    <property name="string1" expr="charAt(sampleProp, 0)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**chr**

`char chr(int)`

Returns a character with the specified ANSI character code. Reference tables of ANSI character codes are available on many websites.

*Note: To add a line break in an expression, use `chr(13)`. The ANSI character code for a line break is 13.*

Generally, when a string function, such as the `chr` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>int</th>
<th>Character code.</th>
</tr>
</thead>
</table>
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="sampleProp" type="integer"/>
    <property name="string2" expr="chr(sampleProp)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

collectionAvg

double collectionAvg(collection of T)

Returns the average of the numeric values in a collection. If size of the collection is 0 or the collection is null, 0 is returned.

Parameters

<table>
<thead>
<tr>
<th>collection of T</th>
<th>Collection for which to determine the average numeric value.</th>
</tr>
</thead>
</table>

See also

“collectionMax” on page 116, “collectionMin” on page 117

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="name" type="string"/>
    <property name="scores" type="Score[]"/>
    <property name="average">
      <expr><![CDATA[collectionAvg(scores)]]> <![CDATA[</expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Score" persistent="true">
    <property name="test" type="string"/>
    <property name="grade" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

collectionMax

T collectionMax(collection of T)

Returns the largest numeric value in a collection. If size of the collection is 0 or the collection is null, 0 is returned.

Parameters

<table>
<thead>
<tr>
<th>collection of T</th>
<th>Collection for which to determine the largest numeric value.</th>
</tr>
</thead>
</table>
See also
“collectionAvg” on page 116, “collectionMin” on page 117

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="name" type="string"/>
    <property name="scores" type="Score[]"/>
    <property name="max">
      <expr><![CDATA[collectionMax(scores)]]]></expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Score" persistent="true">
    <property name="test" type="string"/>
    <property name="grade" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

collectionMin

T collectionMin(collection of T)

Returns the smallest numeric value in a collection. If size of the collection is 0 or the collection is null, 0 is returned.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection of T</td>
<td>Collection for which to determine the smallest numeric value.</td>
</tr>
</tbody>
</table>

See also
“collectionAvg” on page 116, “collectionMax” on page 116

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="name" type="string"/>
    <property name="scores" type="Score[]"/>
    <property name="min">
      <expr><![CDATA[collectionMin(scores)]]]></expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Score" persistent="true">
    <property name="test" type="string"/>
    <property name="grade" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
collectionSum

collectionSum(T)  
Returns the sum of the numeric values in a collection. If size of the collection is 0 or the collection is null, 0 is returned.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection of T</td>
<td>Collection for which to determine the sum of the numeric values.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="name" type="string"/>
    <property name="scores" type="Score[]"/>
    <property name="average">
      <expr><![CDATA[collectionAvg(scores)]]></expr>
    </property>
    <id name="id" type="integer"/>
  </entity>
  <entity name="Score" persistent="true">
    <property name="test" type="string"/>
    <property name="grade" type="integer"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

compare

```plaintext
integer compare(string1, string2)
```

Performs a case-sensitive comparison of two strings.

- Returns one of the following as an indicator of the difference between the strings:
  - -1, if string1 is less than string2
  - 0, if string1 is equal to string2
  - 1, if string1 is greater than string2

Expressions also allow string1 > string2 style comparison.

Generally, when a string function, such as the compare function, has null as an input, it returns null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>String to compare.</td>
</tr>
<tr>
<td>string2</td>
<td>String to compare.</td>
</tr>
</tbody>
</table>
**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string3" expr="compare('abc', 'ABC')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**compareNoCase**

`integer compareNoCase(string1, string2)`

Performs a case-insensitive comparison of two strings.

Returns one of the following as an indicator of the difference between the strings:

- `-1`, if `string1` is less than `string2`
- `0`, if `string1` is equal to `string2`
- `1`, if `string1` is greater than `string2`

Expressions also allow `string1 > string2` style comparison.

Generally, when a string function, such as the `compareNoCase` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>String to compare.</td>
</tr>
<tr>
<td>string2</td>
<td>String to compare.</td>
</tr>
</tbody>
</table>

**See also**

“*compare*” on page 118

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string3" expr="compare('abc', 'ABC')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**createDate**

`date createDate(year, month, day, hour, minute, second)`

Creates a date object in the local time zone.

**See also**

“*createDateUTC*” on page 120
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Integer in the range 0-9999. Integers in the range 0-29 are converted to 2000-2029. Integers in the range 30-99 are converted to 1930-1999. You cannot specify dates before AD 100.</td>
</tr>
<tr>
<td>month</td>
<td>Integer in the range 1 (January)-12 (December).</td>
</tr>
<tr>
<td>day</td>
<td>Integer in the range 1-31.</td>
</tr>
<tr>
<td>hour</td>
<td>Integer in the range 1-24.</td>
</tr>
<tr>
<td>minute</td>
<td>Integer in the range 1-59.</td>
</tr>
<tr>
<td>second</td>
<td>Integer in the range 1-59.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="date1" expr="createDate(2009, 3, 19, 14, 55, 00)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

createDateUTC

date createDateUTC(year, month, day, hour, minute, second)

Returns a date object in the UTC time zone.

See also

“createDate” on page 119

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Integer in the range 0-9999. Integers in the range 0-29 are converted to 2000-2029. Integers in the range 30-99 are converted to 1930-1999. You cannot specify dates before AD 100.</td>
</tr>
<tr>
<td>month</td>
<td>Integer in the range 1 (January)-12 (December).</td>
</tr>
<tr>
<td>day</td>
<td>Integer in the range 1-31.</td>
</tr>
<tr>
<td>hour</td>
<td>Integer in the range 1-24.</td>
</tr>
<tr>
<td>minute</td>
<td>Integer in the range 1-59.</td>
</tr>
<tr>
<td>second</td>
<td>Integer in the range 1-59.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="date1" expr="createDateUTC(2009, 3, 19, 14, 55, 00)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
dateAdd

date dateAdd("Datepart", integer, date)

Adds units of time to a date and returns the date. Returns 0 if date is null.

Parameters

datepart  String:
  • yyyy: Year
  • q: Quarter
  • m: Month
  • y: Day of year
  • d: Day
  • w: Weekday
  • ww: Week
  • h: Hour
  • n: Minute
  • s: Second
  • l: Millisecond

integer  Number of units of datepart to add to date (positive, to get dates in the future; negative, to get dates in the past). Number must be an integer.

date  Date object, in the range 100 AD–9999 AD.

Example

<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date3" expr="dateAdd('h', 3, created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

dateCompare

integer dateCompare(date1, date2)

Compares two dates. Returns one of the following values:
  • -1, if date1 is earlier than date2
  • 0, if date1 is equal to date2
  • 1, if date1 is later than date2

Returns 0 if either data is null.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
<tr>
<td>date2</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

See also
“dateDiff” on page 122

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date4" expr="dateCompare(created, now())"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

dateDiff

`long dateDiff("Datepart", date1, date2)`

Determines the integer number of units by which `date1` is less than `date2`. Returns the number of units, of type `datepart`. Returns 0 if either date is null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datepart</td>
<td>String that specifies the units in which to count; for example yyyy requests a date difference in whole years.</td>
</tr>
<tr>
<td></td>
<td>- yyyy: Years</td>
</tr>
<tr>
<td></td>
<td>- q: Quarters</td>
</tr>
<tr>
<td></td>
<td>- m: Months</td>
</tr>
<tr>
<td></td>
<td>- y: Days of year (same as d)</td>
</tr>
<tr>
<td></td>
<td>- d: Days</td>
</tr>
<tr>
<td></td>
<td>- ww: Weeks</td>
</tr>
<tr>
<td></td>
<td>- h: Hours</td>
</tr>
<tr>
<td></td>
<td>- n: Minutes</td>
</tr>
<tr>
<td></td>
<td>- s: Seconds</td>
</tr>
<tr>
<td>date1</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
<tr>
<td>date2</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

See also
“dateCompare” on page 121
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date5" expr="dateDiff('m', created, created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

datePart

integer datePart("Datepart", date)

Returns part of a date, as an integer. Returns 0 if date is null.

Parameters

datepart  String:
  • yyyy: Year
  • q: Quarter
  • m: Month
  • y: Day of year
  • d: Day
  • w: Weekday
  • ww: Week
  • h: Hour
  • n: Minute
  • s: Second
  • l: Millisecond

date  Date object, in the range 100 AD–9999 AD.

See also

“datePartUTC” on page 124

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date6" expr="datePart('yyyy', created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
**datePartUTC**

integer datePartUTC("Datepart", date)

Returns a part of a date in UTC time, as an integer. Returns 0 if either date is null.

**Parameters**

<table>
<thead>
<tr>
<th>datepart</th>
<th>String:</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>Year</td>
</tr>
<tr>
<td>q</td>
<td>Quarter</td>
</tr>
<tr>
<td>m</td>
<td>Month</td>
</tr>
<tr>
<td>y</td>
<td>Day of year</td>
</tr>
<tr>
<td>d</td>
<td>Day</td>
</tr>
<tr>
<td>w</td>
<td>Weekday</td>
</tr>
<tr>
<td>ww</td>
<td>Week</td>
</tr>
<tr>
<td>h</td>
<td>Hour</td>
</tr>
<tr>
<td>n</td>
<td>Minute</td>
</tr>
<tr>
<td>s</td>
<td>Second</td>
</tr>
<tr>
<td>l</td>
<td>Millisecond</td>
</tr>
</tbody>
</table>

**date**

Date object, in the range 100 AD–9999 AD.

**See also**

“datePart” on page 123

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date6" expr="datePartUTC('yyyy', created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**dayOfWeek**

integer dayOfWeek(date)

Returns the ordinal for the day of the week in the range 1 (Sunday) to 7 (Saturday). Returns -1 if date is null.

**Parameters**

| date | Date object, in the range 100 AD–9999 AD. |
See also
“getDay” on page 131, “getDayUTC” on page 131, “dayOfYear” on page 125, “firstDayOfMonth” on page 129

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date21" expr="dayOfWeek(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

dayOfYear

integer dayOfYear(date)

Returns the ordinal day of the year as specified by date (1-366). Returns -1 if date is null.

Parameters

| date                  | Date object, in the range 100 AD–9999 AD. |

See also
“getDay” on page 131, “getDayUTC” on page 131, “dayOfWeek” on page 124, “firstDayOfMonth” on page 129

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date22" expr="dayOfYear(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

daysInMonth

integer daysInMonth(date)

Returns the number of days in the month as specified by date (for example, 28, 29, 30, 31). Returns 0 if date is null.

See also
“daysInYear” on page 126

Parameters

| date                  | Date object, in the range 100 AD–9999 AD. |
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date23" expr="daysInMonth(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**daysInYear**

`integer daysInYear(date)`

Returns the number of days in the year as specified by date (365 or 366). Returns 0 if date is null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

**See also**

“daysInMonth” on page 125

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date24" expr="daysInYear(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**endsWith**

`boolean endsWith(substring, string)`

Returns true if string ends with substring.

Generally, when a string function, such as the `endsWith` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>substring</td>
<td>String for which to search.</td>
</tr>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
</tbody>
</table>

**See also**

“startsWith” on page 150
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string51" expr="endsWith('abc', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### find

**integer find(substring, string)**

Returns the index of the first occurrence of a *substring* in a *string* or 0 (zero) if *substring* is not in *string*. The search is case sensitive.

Generally, when a string function, such as the `find` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>substring</th>
<th>String for which to search.</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
</tbody>
</table>

**See also**

“findLast” on page 128, “findNoCase” on page 127, “findLastNoCase” on page 128, “findOneOf” on page 129

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string5" expr="find('abc', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### findNoCase

**integer findNoCase(substring, string)**

Returns the index of the first occurrence of a *substring* in a *string*. If *substring* is not in *string*, returns 0 (zero). The search is case-insensitive.

Generally, when a string function, such as the `findNoCase` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>substring</th>
<th>String for which to search.</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
</tbody>
</table>

**See also**

“find” on page 127, “findLast” on page 128, “findLastNoCase” on page 128, “findOneOf” on page 129

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string5" expr="findNoCase('abc', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string6" expr="findNoCase('abc', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**findLast**

integer findLast(substring, string)

Searches string from right to left and returns the index of the last occurrence of substring. The search is case sensitive. Generally, when a string function, such as the findLast function, has null as an input, it returns null.

**Parameters**

| substring | String for which to search. |
| string    | String in which to search. |

**See also**

“find” on page 127, “findNoCase” on page 127, “findLastNoCase” on page 128, “findOneOf” on page 129

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string7" expr="findLast('abc', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**findLastNoCase**

integer findLastNoCase(substring, string)

Searches string from right to left and returns the index of the last occurrence of substring. The search is case-insensitive. Generally, when a string function, such as the findLastNoCase function, has null as an input, it returns null.

**Parameters**

| substring | String for which to search. |
| string    | String in which to search. |

**See also**

“find” on page 127, “findLast” on page 128, “findNoCase” on page 127, “findOneOf” on page 129
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string8" expr="findLastNoCase('abc', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**findOneOf**

date integer findOneOf(set, string)

Returns the index of the first occurrence of any one of a *set* of characters in a *string*. The search is case sensitive.

Generally, when a string function, such as the `findOneOf` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>String that contains one or more characters to search for.</td>
</tr>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
</tbody>
</table>

**See also**

“compare” on page 118, “compareNoCase” on page 119, “find” on page 127, “findNoCase” on page 127, “findLast” on page 128, “findLastNoCase” on page 128

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string9" expr="findOneOf('abc', 'testing a little')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**firstDayOfMonth**

date integer firstDayOfMonth(date)

Returns an integer that corresponds to the ordinal (day number, in the year) of the first day of the month in which the given date falls.

**See also**

“getDay” on page 131, “getDayUTC” on page 131, “dayOfWeek” on page 124, “dayOfYear” on page 125

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date25" expr="firstDayOfMonth(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**fix**

`number fix(number)`

Converts a real number to an integer. If `number` is greater than or equal to 0 (zero), converts to the closest integer less than or equal to `number`. If `number` is less than 0 (zero), converts to the closest integer greater than or equal to `number`. The return type matches the input type.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Number to convert.</td>
</tr>
</tbody>
</table>

**See also**

“ceiling” on page 114, “floor” on page 130, “round” on page 147

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="math3" expr="fix(-4)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**floor**

`number floor(number)`

Returns the closest number that is smaller than `number`. For example, the function returns 3 for `floor(3.3)` and `floor(3.7)`; it returns -4 for `floor(-3.3)` and `floor(-3.7)`. The return type matches the input type.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Number from which to return the closest number that is smaller.</td>
</tr>
</tbody>
</table>

**See also**

“fix” on page 130, “round” on page 147
Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="math4" expr="floor(3.3)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

getDay

integer getDay(date)

Returns the ordinal for the day of the month, ranging from 1 to 31. Returns 0 if date is null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

See also
“getDayUTC” on page 131, “dayOfWeek” on page 124, “dayOfYear” on page 125, “firstDayOfMonth” on page 129

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date7" expr="getDay(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

getDayUTC

integer getDayUTC(date)

Returns the UTC ordinal day of the month, ranging from 1 to 31. Returns 0 if date is null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

See also
“getDay” on page 131, “dayOfWeek” on page 124, “dayOfYear” on page 125
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date14" expr="getDayUTC(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**getHour**

integer getHour(date)

Returns the current hour of the day in the range 0-23. Returns -1 if date is null.

See also

“datePart” on page 123, “datePartUTC” on page 124, “getHourUTC” on page 132, “getMinute” on page 133, “getMinuteUTC” on page 133, “getSecond” on page 135, “getSecondUTC” on page 136

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date8" expr="getHour(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**getHourUTC**

integer getHourUTC(date)

Returns the current UTC hour of the day in the range 0-23. Returns -1 if date is null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

See also

“datePart” on page 123, “datePartUTC” on page 124, “getHour” on page 132, “getMinute” on page 133, “getMinuteUTC” on page 133, “getSecond” on page 135, “getSecondUTC” on page 136
Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date15" expr="getHourUTC(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

getMinute

integer getMinute(date)

Extracts the minute value from a date object. Returns the ordinal value of the minute in the range 0–59. Returns -1 if date is null.

Parameters

| date | Date object, in the range 100 AD–9999 AD. |

See also
“getMinuteUTC” on page 133, “getSecond” on page 135, “getSecondUTC” on page 136

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date9" expr="getMinute(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

getMinuteUTC

integer getMinuteUTC(date)

Extracts the minute value from a date object. Returns the ordinal value of the minute in the range 0–59. Returns -1 if date is null.

Parameters

| date | Date object in the range 100 AD–9999 AD. |

See also
“getMinute” on page 133, “getSecond” on page 135, “getSecondUTC” on page 136
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date16" expr="getMinuteUTC(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**getMonth**

integer getMonth(date)

Returns the ordinal value of the month, in the range 1 (January) – 12 (December), from a date object. Returns 0 if date is null.

**Parameters**

| date        | Date object, in the range 100 AD–9999 AD. |

**See also**

“datePart” on page 123, “datePartUTC” on page 124, “getMonthUTC” on page 134, “getQuarter” on page 135

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date10" expr="getMonth(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**getMonthUTC**

integer getMonthUTC(date)

Returns the month UTC value from a date object. Returns 0 if date is null.

**Parameters**

| date        | Date object, in the range 100 AD–9999 AD. |

**See also**

“datePart” on page 123, “datePartUTC” on page 124, “getMonth” on page 134, “getQuarter” on page 135
Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date17" expr="getMonthUTC(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### getQuarter

```
integer getQuarter(date)
```

Calculates the quarter of the year in which a date falls. Returns 0 if date is null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

**Returns**

An integer, 1–4.

**See also**

“datePart” on page 123, “datePartUTC” on page 124, “getMonth” on page 134, “getMonthUTC” on page 134

Example

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date27" expr="getQuarter(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### getSecond

```
integer getSecond(date)
```

Returns an integer in the range 0-59 that is the ordinal for the second from a date object. Returns -1 if date is null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object.</td>
</tr>
</tbody>
</table>

**See also**

“datePart” on page 123, “datePartUTC” on page 124, “getMinute” on page 133, “getMinuteUTC” on page 133
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date11" expr="getSecond(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**getSecondUTC**

**integer getSecondUTC(date)**

Returns an integer in the range 0-59 that is the ordinal for the second from a date object. Returns -1 if date is null.

Parameters

| date   | Date object. |

See also

“datePart” on page 123, “datePartUTC” on page 124, “getMinute” on page 133, “getMinuteUTC” on page 133

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date18" expr="getSecondUTC(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**getYear**

**integer getYear(date)**

Returns the year value from a date object. Returns 0 if date is null.

Parameters

| date   | Date object, in the range 100 AD–9999 AD. |

See also

“datePart” on page 123, “isLeapYear” on page 138
Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date12" expr="getYear(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

getYearUTC

integer getYearUTC(date)

Returns the UTC year value from a date object. Returns 0 if date is null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

See also
“datePart” on page 123, “isLeapYear” on page 138

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date19" expr="getYearUTC(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

insert

string insert(substring, string, position)

Inserts a substring in a string after a specified character position. If position equals -1, prefixes the substring to the string. Returns the string, after inserting the substring.

Generally, when a string function, such as the insert function, has null as an input, it returns null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>substring</td>
<td>Substring to insert into string.</td>
</tr>
<tr>
<td>string</td>
<td>String in which to insert substring.</td>
</tr>
<tr>
<td>position</td>
<td>Position at which to insert substring into string.</td>
</tr>
</tbody>
</table>
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string10" expr="insert('tom', 'ABCXYZ', 2)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**isEmpty**

boolean isEmpty(collection)

Returns true if the size of the collection equals 0.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>Collection for which to determine the size.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="otherApplicants" type="OtherApplicants[]"/>
    <property name="collection2" expr="isEmpty(otherApplicants)"/>
    <id name="id" type="integer"/>
  </entity>
  <entity name="OtherApplicants">
    <property name="name" type="string"/>
  </entity>
</model>
```

**isLeapYear**

boolean isLeapYear(integer)

Determines whether a year is a leap year.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="date26" expr="isLeapYear(2000)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
left

string left(string, count)

Returns the leftmost count characters in a string.
Generally, when a string function, such as the left function, has null as an input, it returns null.

Parameters

<table>
<thead>
<tr>
<th>string</th>
<th>String from which to get the leftmost count characters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Positive integer. Number of characters to return.</td>
</tr>
</tbody>
</table>

See also
“right” on page 146, “len” on page 140

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string13" expr="left('abcXYZ', 3)="/n    <id name="id" type="integer"/>
  </entity>
</model>
```

leftTrim

string leftTrim(string)

Remove leading spaces from a string. Returns a copy of the string, without the leading spaces.
Generally, when a string function, such as the leftTrim function, has null as an input, it returns null.

Parameters

| string | String to trim. |

See also
“rightTrim” on page 147

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string12" expr="leftTrim(' space '="/n    <id name="id" type="integer"/>
  </entity>
</model>
```
**len**

**integer len(string)**

Returns the length of a string.

Generally, when a string function, such as the `len` function, has null as an input, it returns null.

**Parameters**

| String from which to get the length. |

**See also**

“left” on page 139, “right” on page 146

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string14" expr="len('abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**lower**

**string lower(string)**

Converts the alphabetic characters in a string to lowercase and returns the string.

Generally, when a string function, such as the `lower` function, has null as an input, it returns null.

**Parameters**

| String to convert. |

**See also**

“upper” on page 154

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string11" expr="lower('ABC')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**max**

**number max(number1, number2)**
Returns the greater of two numbers. The return type matches the input types. If the input types do not match, the return type is the coercion type that can hold either result. For instance, if input types are a long and an integer, a long type is returned. If the input types are a double and a long, a double is returned. If the input types are a float and a long, a double is returned.

Parameters

<table>
<thead>
<tr>
<th>number1</th>
<th>First input number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>number2</td>
<td>Second input number.</td>
</tr>
</tbody>
</table>

See also
“min” on page 141

Example

```
<html xmlns="http://ns.adobe.com/Fiber/1.0">
  <body>
    <head></head>
    <entity name="Application" persistent="true">
      <property name="math5" expr="max(40, 50.5)"/>
      <id name="id" type="integer"/>
    </entity>
  </body>
</html>
```

**min**

number min(number1, number2)

Returns the lesser of two numbers. The return type matches the input types. If the input types do not match, the return type is the coercion type that can hold either result. For instance, if input types are a long and an integer, a long type is returned. If the input types are a double and a long, a double is returned. If the input types are a float and a long, a double is returned.

Parameters

| number1, number2 | Numbers to compare. |

See also
“max” on page 140

Example

```
<html xmlns="http://ns.adobe.com/Fiber/1.0">
  <body>
    <head></head>
    <entity name="Application" persistent="true">
      <property name="math5" expr="min(40, 50.5)"/>
      <id name="id" type="integer"/>
    </entity>
  </body>
</html>
```

**missing**

boolean missing(PropertyReference)
APPLICATION MODELING TECHNOLOGY REFERENCE

Takes a single parameter that must be a reference to an optional or required property. Returns true if the parameter has not been set.

Parameters

| PropertyReference | Reference to a property. |

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Company" persistent="true">
    <property name="industry" type="string" required="false"/>
    <property name="industryWithDefault" expr="!missing(industry) ? 'DefaultIndustry' : industry"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**now**

date Now()

Returns the current date and time. The return value can be passed as a parameter to date functions such as dateCompare, daysInYear or firstDayOfMonth.

Returns

A date object; the current date and time.

See also

“createDate” on page 119, “createDateUTC” on page 120, “datePart” on page 123, “datePartUTC” on page 124

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date4" expr="dateCompare(created, now())"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**part**

integer part(value, collection)

Given a value and a sorted collection of values, returns the number of items less than value. If not sorted, returns the length of the collection prefix made up of items less than value. For example, given a collection C, containing [10, 20, 30], the call part(20, C) returns 1 because there is a single value (10) in the collection that is less than 20.

Inputs must be comparable, otherwise 0 is returned. The comparable types are boolean, char, date, numeric types (integer, long, float, double), and string.
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="collection3" expr="part(20, [10, 20, 30] )"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**removeChars**

```xml
removeChars(string, start, count)
```

Removes characters from a string. Returns a copy of the string, with `count` characters removed from the specified start position. If no characters are found, returns zero.

Generally, when a string function, such as the `removeChars` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>string</th>
<th>String in which to search.</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Positive integer. Position at which to start search.</td>
</tr>
<tr>
<td>count</td>
<td>Positive integer. Number of characters to remove.</td>
</tr>
</tbody>
</table>

**See also**

“insert” on page 137, “len” on page 140

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string17" expr="removeChars('abcXYZabc', 3, 3)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**repeatString**

```xml
string(string, count)
```

Creates a string that contains a specified number of repetitions of the specified string.

Generally, when a string function, such as the `repeatString` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>string</th>
<th>String from which to create new string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Positive integer. Number of repeats.</td>
</tr>
</tbody>
</table>
Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string18" expr="repeatString('abc', 3)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

**replace**

string Replace(string, substring1, substring2)

Replaces the first occurrence of substring1 in a string with substring2, in a specified scope. The search is case sensitive. Returns the string, after making replacements.

Generally, when a string function, such as the replace function, has null as an input, it returns null.

See also
“replaceAll” on page 144, “replaceAllNoCase” on page 145, “replaceNoCase” on page 145

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
<tr>
<td>substring1</td>
<td>String for which to search.</td>
</tr>
<tr>
<td>substring2</td>
<td>String that replaces substring1</td>
</tr>
</tbody>
</table>

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string19" expr="replace('abcTomxyz', 'Tom', 'Zebra')"/>
    <id name="id" type="integer"/>
  </entity>
</model>

**replaceAll**

string replaceAll(string, substring1, substring2)

Replaces all occurrences of substring1 in string with substring2. The search is case sensitive. Returns the string, after making replacements.

Generally, when a string function, such as the replaceAll function, has null as an input, it returns null.

See also
“replace” on page 144
### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
<tr>
<td>substring1</td>
<td>String for which to search.</td>
</tr>
<tr>
<td>substring2</td>
<td>String that replaces substring1.</td>
</tr>
</tbody>
</table>

### Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string21" expr="replaceAll('abcTomxyz', 'Tom', 'Zebra')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### replaceAllNoCase

```java
string replaceAllNoCase(string, substring1, substring2)
```

Replaces all occurrences of `substring1` in `string` with `substring2`. The search is case-insensitive. Returns the string, after making replacements.

Generally, when a string function, such as the `replaceAllNoCase` function, has null as an input, it returns null.

See also

- “replace” on page 144, “replaceAll” on page 144, “replaceNoCase” on page 145

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
<tr>
<td>substring1</td>
<td>String for which to search.</td>
</tr>
<tr>
<td>substring2</td>
<td>String that replaces substring1.</td>
</tr>
</tbody>
</table>

### Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string22" expr="replaceAllNoCase('abcTomxyzTom', 'tom', 'Zebra')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### replaceNoCase

```java
string replaceNoCase(string, substring1, substring2)
```

Replaces the first occurrence of `substring1` in `string` with `substring2`, in a specified scope. The search is case-insensitive. Returns the string, after making replacements.

Generally, when a string function, such as the `replaceNoCase` function, has null as an input, it returns null.
See also
“replace” on page 144, “replaceAll” on page 144, “replaceAllNoCase” on page 145

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
<tr>
<td>substring1</td>
<td>String for which to search</td>
</tr>
<tr>
<td>substring2</td>
<td>String that replaces substring1</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string20" expr="replaceNoCase('abcTomxyz', 'tom', 'Zebra')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**reverse**

```xml
string reverse(string)
```

Returns a copy of `string`, with the characters in reverse order.

Generally, when a string function, such as the `reverse` function, has null as an input, it returns null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String to reverse.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string23" expr="reverse('abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**right**

```xml
string right(string, count)
```

Returns the rightmost `count` characters in a string.

Generally, when a string function, such as the `right` function, has null as an input, it returns null.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String from which to get the rightmost <code>count</code> characters.</td>
</tr>
<tr>
<td>count</td>
<td>Positive integer. Number of characters to return.</td>
</tr>
</tbody>
</table>
See also
“left” on page 139, “len” on page 140

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string24" expr="right('abcXYZ', 3)"/>
    <id name="id" type="integer"/>
  </entity>
</model>

**rightTrim**

string rightTrim(string)

Remove trailing spaces from a string. Returns a copy of the string, without trailing spaces.

Generally, when a string function, such as the `rightTrim` function, has null as an input, it returns null.

Parameters

<table>
<thead>
<tr>
<th></th>
<th>String to trim.</th>
</tr>
</thead>
</table>

See also
“leftTrim” on page 139

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string16" expr="rightTrim(' space ')"/>
    <id name="id" type="integer"/>
  </entity>
</model>

**round**

number round(number)

Rounds a number to the closest integer and returns the rounded number. Rounds numbers that end with .5 up to the nearest integer. It rounds 3.5 to 4 and -3.5 to -3. The return type matches the input types.

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Number to round.</th>
</tr>
</thead>
</table>

See also
“ceiling” on page 114, “fix” on page 130
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="math7" expr="round(3.5)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### sizeof

describe

integer sizeof(collection)

Returns the number of elements in a collection.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>Collection in which to find the number of elements.</td>
</tr>
</tbody>
</table>

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="otherApplicants" type="OtherApplicants[]"/>
    <property name="collection1" expr="sizeof(otherApplicants)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### sgn

describe

integer sgn(number)

Determines the sign of a number. Returns one of the following:

- 1, if `number` is positive.
- 0, if `number` is 0.
- -1, if `number` is negative.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Number for which to determine the sign.</td>
</tr>
</tbody>
</table>

**See also**

“abs” on page 114
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="math8" expr="sgn(-150)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**spanExcluding**

string spanExcluding(string, set)

Returns characters from a string, from the beginning to a character that is in a specified set of characters. The search is case sensitive.

Generally, when a string function, such as the `spanExcluding` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String from which to get characters.</td>
</tr>
<tr>
<td>set</td>
<td>Set of characters in a string.</td>
</tr>
</tbody>
</table>

See also

“spanIncluding” on page 149

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string25" expr="spanExcluding('abc', 'ABC')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**spanIncluding**

string spanIncluding(string, set)

Returns characters from a string, from the beginning to a character that is not in a specified set of characters. The search is case sensitive.

Generally, when a string function, such as the `spanIncluding` function, has null as an input, it returns null.

**See also**

“spanExcluding” on page 149

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String from which to get characters.</td>
</tr>
<tr>
<td>set</td>
<td>Set of characters in a string.</td>
</tr>
</tbody>
</table>
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string26" expr="spanIncluding('abcxyz', 'xyz')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**startsWith**

boolean `startsWith(substring, string)`

Return `true` if `string` starts with `substring`.

Generally, when a string function, such as the `startsWith` function, has null as an input, it returns null.

See also

“endsWith” on page 126

**Parameters**

<table>
<thead>
<tr>
<th>substring</th>
<th>String to search for.</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String in which to search.</td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string261" expr="startsWith('now', 'now ABC then abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**substring**

`string substring(string, start, count)`

Extracts and returns a substring from a `string`, beginning at `start`, of length `count`.

Generally, when a string function, such as the `substring` function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>string</th>
<th>String to search for.</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>String in which to search.</td>
</tr>
<tr>
<td>count</td>
<td>Positive integer. Number of characters to extract.</td>
</tr>
</tbody>
</table>
Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <entity name="Application" persistent="true">
        <property name="string15" expr="substring('abcONExyz', 3, 3)="/>
        <id name="id" type="integer"/>
    </entity>
</model>

today

date today()

Returns the current date and time. The return value can be passed as a parameter to date functions such as dateCompare, DaysInYear or FirstDayOfMonth.

Returns
A date object; the current date and time.

See also
“createDate” on page 119, “createDateUTC” on page 120, “datePart” on page 123, “datePartUTC” on page 124

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <entity name="Application" persistent="true">
        <property name="created" type="date="/>
        <property name="date4" expr="dateCompare(created, today())="/>
        <id name="id" type="integer"/>
    </entity>
</model>

toDouble

double toDouble(string)

Converts a string to a double and returns the double.

Parameters

<table>
<thead>
<tr>
<th>string</th>
<th>String to convert.</th>
</tr>
</thead>
</table>

Example
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <entity name="Application" persistent="true">
        <property name="conversion4" expr="toDouble(123.456)="/>
        <id name="id" type="integer"/>
    </entity>
</model>
**toFloat**

float toFloat(string)

Converts a string to a float and returns the float.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String to convert.</td>
</tr>
</tbody>
</table>

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <entity name="Application" persistent="true">
        <property name="conversion3" expr="toFloat(123.456)"/>
        <id name="id" type="integer"/>
    </entity>
</model>
```

**toInteger**

int toInteger(string)

Converts a string to an integer and returns the integer.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String to convert.</td>
</tr>
</tbody>
</table>

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <entity name="Application" persistent="true">
        <property name="conversion1" expr="toInteger(123.456)"/>
        <id name="id" type="integer"/>
    </entity>
</model>
```

**toLong**

long toLong(string)

Converts a string to a long and returns the long.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String to convert.</td>
</tr>
</tbody>
</table>
Parameters

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="conversion2" expr="toLong(123.456)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**toString**

string `toString(value)`

Returns the string representation of `value`. Generally, when a string function, such as the `toString` function, has null as an input, it returns null.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
</tr>
</tbody>
</table>

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="conversion5" expr="toString(123.456)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**trim**

string `trim(string)`

Removes leading and trailing spaces from a string and returns a copy of the string. Generally, when a string function, such as the `trim` function, has null as an input, it returns null.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
</tr>
</tbody>
</table>

See also

"leftTrim" on page 139, "rightTrim" on page 147
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string27" expr="trim('  space  ')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**upper**

string upper(string)

Converts the alphabetic characters in a string to uppercase and returns the string.

Generally, when a string function, such as the upper function, has null as an input, it returns null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String to convert.</td>
</tr>
</tbody>
</table>

**See also**

"lower" on page 140

Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="string28" expr="upper('abc')"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

**weekOfYear**

integer weekOfYear(date)

Returns the week number within the year from a date object. Returns an integer in the range 1–53; the ordinal of the week, within the year. Returns 0 if date is null.

**See also**

"getDay" on page 131, “getDayUTC” on page 131

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>
Example

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date28" expr="weekOfYear(created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```

### years

date integer years(endDate, startDate)

Returns the number of years between `startDate` and `endDate`. Returns -1 if either date is null.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date object, in the range 100 AD–9999 AD.</td>
</tr>
</tbody>
</table>

**Example**

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Application" persistent="true">
    <property name="created" type="date"/>
    <property name="date20" expr="years(now(), created)"/>
    <id name="id" type="integer"/>
  </entity>
</model>
```
Chapter 6: Client code generation

The application modeling technology ActionScript generator creates client-side ActionScript classes based on the entity definitions in a model. The ActionScript generator also creates service wrapper classes based on the implicit and explicit service definitions in a model.

The ActionScript generator is integrated into Flash Builder. Flash Builder provides a common programming model for Flex clients that use generated ActionScript classes.

When you deploy a valid model to the .model directory (default directory location) of a Flex project in Flash Builder, the ActionScript generator creates corresponding ActionScript classes. The model you create must have the same name as the Flex project. The Flex SDK contains supporting classes and interfaces on which the generated ActionScript classes depend. Once the ActionScript classes are generated, you can use them in Flash Builder to create MXML applications that access remote services.

Using Flash Builder data service wizards
Flash Builder provides data service wizards that introspect remote services and generate models based on those services. This type of model-driven development generates only client-side code. It has no effect on the code in the remote service that is introspected.

The ActionScript generator creates ActionScript code based on the model that data service wizards generate. Flash Builder supports this type of client-only code generation for Flex clients that access PHP, ColdFusion, SOAP-based web services, REST (HTTP) services, as well as BlazeDS and LiveCycle Data Services remote objects, and LiveCycle service processes.

By default, the models that data service wizards generate are hidden and are not manually edited. However, you can change Flash Builder settings so that the .model directory and the model files contained in it are visible in the Flash Builder Package Explorer. To do so, in the Package Explorer, click the Filters button. In the Filters dialog, deselect ".*" and ".model".

Using the Modeler with Flash Builder
When working with LiveCycle Data Services, you use Flash Builder with the Modeler tool to create ActionScript objects for a Flex client. Unlike the data service wizard use cases, you have complete control over the content of your model and you can generate code for both the client and server sides of your application. In the Modeler tool, you can start with a blank model and build the model manually, or you can generate entity elements from a SQL database by dragging database tables onto the model.

You can deploy a model to the LiveCycle Data Services server to generate corresponding Java code that works directly with the underlying SQL database and matches the APIs of the ActionScript code generated for the Flex client.

The entity utility generates the server-side code. It generates Data Management Service destinations based on the entities in a model. These destinations use instances of an assembler called the Model Assembler, which extends the standard Hibernate Assembler in LiveCycle Data Services. By default, the Model Assembler instances are only stored in memory. They are not saved to disk.

Note: If you do not have existing database tables that match the entities in your model, the entity utility automatically generates the tables when it generates Java code on the server based on annotations in the model.

The ActionScript generator in Flash Builder and the entity utility in LiveCycle Data Services provide a powerful and fast way to create end-to-end model-driven applications.
ActionScript objects

The ActionScript generator creates a set of three classes for each entity in a model. These classes provide the following functionality:

- Act as a bean (provide getters and setters) for data properties of an entity
- Provide access to derived properties of an entity and a cache for them
- Provide a validation API that checks all relevant constraints
- Provide APIs that determine when guarded properties (properties used in variants) are available
- Provide a reflection API that lets users determine the set of currently available properties, determine dependency information for derived properties, and invalidate the derived property cache
- Provide an API for retrieving style information
- Provide an entry point for plugging in custom behavior, such as custom method implementations

Note: The terms value object and value object are sometimes used interchangeably in code and documentation to refer to objects that represent entities in ActionScript.

Data properties in entities

Private variables and corresponding public getters and setters in a generated ActionScript object represent the data properties of the entities in a model.

Getters and setters for data properties contained in variants do not throw an exception if the current state of the object is such that the property does not exist. Consumers of the object are expected to call the `isAvailable(propertyName)` method or bind to the `ispropertyNameAvailable` property on the model helper object to determine if a property is available before interacting with it. This pattern allows access to such properties outside try-catch blocks that would otherwise be required for all interactions with guarded properties, including binding.

More Help topics
“Reflection API” on page 159

Derived properties in entities

A private variable and a corresponding getter method in generated ActionScript classes represent the derived properties of an entity in a model. When a calculation of a derived property finishes, a property change event is triggered to let consumers to bind to derived property values and accurately reflect changed values as the data properties that they depend on change. The ActionScript object caches derived properties once they have been calculated. This caching can result in a significant performance gain for derived properties whose expressions involve complex calculations or external function calls. The cached value is stored in the private variable corresponding to the derived property.

The cached value of a derived property is cleared whenever a property that the derived property depends on is modified. The `invalidateDependentOnpropertyName()` method lets consumers of the object invalidate the cache whenever necessary. The value of a derived property is recalculated when its cache is cleared. Whenever a data property on which a derived property depends is modified or the cache is invalidated explicitly, the value is recalculated and bindings to the property are triggered when the calculation finishes.
Automatic recalculation of derived properties
The recalculation of derived properties is automatically triggered whenever a leaf data property on which it depends is modified. This recalculation ensures that all derived properties are kept up to date with respect to data property changes. For derived properties that do not involve a network (server) call, this recalculation occurs synchronously and immediately.

For derived properties with expressions that involve a function call to a server, the calculation occurs asynchronously. The derived property value is stale while the calculation is performed. A PropertyChange event is dispatched for the property as soon as the calculation completes. The calculation of a derived property can be expensive and you may not want the property recalculated each time a data property changes. To support such cases, you can disable automatic recalculation of derived properties in an annotation. The \texttt{ActionScriptGeneration.DisableAutoReCalc} annotation can appear at both the entity and property levels. If the annotation appears at the entity level then none of the entity's derived properties are automatically recalculated. If it appears at the derived property level, then only that derived property is no longer automatically recalculated.

The \texttt{calculate\_propertyName()} method on a model-level metadata object available for every value object instance, lets you manually trigger recalculation of derived properties for which automatic recalculation is disabled. For example, you would call \texttt{myEntityObject.model.calculate\_derived1()}, to recalculate the derived property named derived1.

All derived properties, disabled or not, are calculated on their initial access and the \texttt{DisableAutoReCalc} annotation only effects subsequent recalculation. Also, calling the \texttt{invalidateDependentOn\_propertyName} APIs for data properties does not cause recalculation of disabled derived properties.

If you disable the automatic recalculation of a derived property in a derived property chain, until the disabled property is manually recalculated, all derived properties down the chain return stale values even if automatic recalculation is not disabled.

Constraint properties in entities
Constraint elements in an entity are Boolean derived properties. They include notation that sets the value object as invalid when at least one of constraint property evaluates to \texttt{false}. A private Boolean variable is generated for each constraint element in an entity and is used for caching its value in the same manner that derived properties are cached. The ActionScript generator creates public getter methods for each constraint. Getter methods for constraints inside variants do not throw an exception when accessed during a non-existent state. Instead, they return \texttt{true} (valid).

The public \texttt{isValid} Boolean bindable property performs validation of the entire value object by iterating over constraints given the current object state. It returns \texttt{false} as soon as a constraint fails or \texttt{true} if all the constraints are all valid.

In addition to the \texttt{isValid} property, the \texttt{getInvalidConstraints} bindable property returns an Array of failed constraint names. Consumers that only care about whether the value object is valid should use the \texttt{isValid} property because it performs better in the case of multiple invalid constraints; it quits as soon as any constraint is found to be invalid. Consumers that must know the names of invalid constraints should call the \texttt{getInvalidConstraints} property and test for validity by checking whether the returned Array is of length zero.

Required properties and types in entities
Data properties in an entity can optional or required. Data properties are optional by default. The type of a value property (Boolean and numeric) is affected by this option because null is a valid member of the type of any property that is optional. The type of a reference property (aggregation, association, collection, string, and so on) always has null as part of its value set regardless of whether the property is optional or not.
The representation of optional value types is handled elegantly in Java by boxed types (Integer, Boolean versus int, boolean). ActionScript does not provide this convenience. Instead, ActionScript value types that are optional default to the type's default value if they are not explicitly present. For example, if an optional Boolean arrives from the server as null, its representation is `false` on the client.

Required reference-type data properties (properties whose type allows null but are required) have an impact on validation and derived properties. An implicit validation is required for every such property whenever the value object is validated with the `isValid` property or a derived property that depends on a required reference-type data property is evaluated. In the latter case, the derived property returns null if it is of a nullable type. It returns its default value if it is of a non-nullable type if any required properties are currently null.

Derived properties continue uninterrupted if optional properties have a null value. There is no way to determine whether that scenario is acceptable, especially in the case of derived properties that reference external function calls.

**Reflection API**

It is often necessary for a consumer of generated value objects to loop over the object properties and perform an action on each; for example, display the property name and value. The value objects provide a set of APIs for retrieving the set of available properties and getting and setting property values.

One case where these APIs are useful is for displaying multiple rows in a Flex DataGrid component.

Application modeling technology provides both entity-level and instance-level reflection APIs. The entity-level APIs in the `com.adobe.fiber.valueobjects.IModelType` interface determine all the derived properties of a value object or what the instance type of this collection property is. The instance-level APIs in the `com.adobe.fiber.valueobjects.IModelInstance` interface get the current value of a property or determine if a particular property is currently available. For more information see the ActionScript 3.0 Reference for the Adobe® Flash® Platform at www.adobe.com/go/learn_lcds31_asapiref_en.

**Generated entity classes**

The output of ActionScript generation for a single entity in a model includes the following classes. These classes are often referred to as value objects.

- `_Super_EntityName.as` is an internal class that contains implementation of all the logic around derived property and metadata API calculation in the model_internal namespace. This class implements the `com.adobe.fiber.valueobjects.IValueObject` interface. It contains public getters and setters for data properties and public getters for derived properties. It also contains a `model` property that exposes the metadata/reflection APIs for the entity.

- `EntityNameEntityMetadata.as` is an internal class that contains public implementations of the value object reflection APIs. This class extends the `com.adobe.fiber.valueobjects.AbstractEntityMetadata` class, which implements the `com.adobe.fiber.valueobjects.IModelType` and `IModelInstance` interfaces. Consumers of value objects can access this object with the `valueObject.model` prefix.

- `EntityName.as` is a public class that extends the `_Super_EntityName.as` class. All references in consumer code are to this class. You can also customize this class by overriding methods. This class is only generated initially and is not overwritten if it exists on disk.

The following illustration shows the value object class hierarchy:
Value object class hierarchy

Available of entity properties within variants
The instance-level reflection API specifies an `isAvailable` function that lets consumers of generated value objects determine whether a particular property exists with respect to variant conditions and the current state of the value object. Since this method is a single entry point for determining the availability of every property of the generated value object, it is not binding friendly. To support binding, the generated value objects have bindable `is<prop-name>Available` get functions for every property of the value object.

Insulation from interface changes
To insulate consumers from potential interface changes, the `AbstractEntityMetadata.as` class throws errors for every method. The generated `entity-name.EntityMetadata` classes provide the actual implementation. `AbstractEntityMetadata` is not abstract; it shields existing consumer code from any changes to the interface. If a client picks up a new version of the interface and abstract class, no code changes are necessary to retain backward compatibility.

Entity method implementation
Method declarations in a model let you declare method prototypes and reference them from derived property expressions. You can plug in implementations for the prototypes outside the model. For example, consider the following model snippet where a developer wants to provide a custom implementation of a `compoundByInterestRate` method:

```
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="BankAccount">
    <property name="balance" type="integer"/>
    <method name="compoundByInterestRate"
      arguments="rate:float" return-type="void"/>
  </entity>
</model>
```

You would modify the `EntityName.as` class and override the `compoundByInterestRate()` method to provide an implementation. A method element can only appear as an immediate child of an entity element in a model.

References to service functions in entities
Generated value objects must be able to call all functions referenced from their derived property expressions. A service declared in a model can become one of the following objects in a Flex client:

1. A custom service with an implementation the user provides
2. A non-managed data service (RemoteObject, HTTPMultiService, WebService)
A managed data service (DataService, or RemoteObject, HTTPMultiService, or WebService used with RPCDataManager)

Services of case 1 are meant as prototype declarations only. Developers must be able to plug in their own service objects that implement the IFiberService interface. You do this by modifying the .as subclass file rather than _.as superclass. The methods of the _.as class throw an exception and it is up to the developer to provide a proper extension of the class. The location of these classes is configurable.

The generation of services in cases 2 and 3 is covered in “Service wrappers” on page 164. The service elements in a model are identified as case two or three by using annotations. Derived properties can reference functions in all three types of services, although it is unlikely that they would reference case 2 service functions. The generated value object must be able to retrieve references to any service that it requires for derived property calculations and call the appropriate functions.

In case one, the value object must have a reference to the typed service so that it can call one of the declared functions. Services in cases 2 and 3 extend AbstractService and the value object can thus call functions dynamically. The com.adobe.fiber.valueobjects.IFiberService and com.adobe.fiber.valueobjects.IFiberManagingService interfaces facilitate this pattern. For more information see ActionScript 3.0 Reference for the Adobe® Flash® Platform at www.adobe.com/go/learn_lcds31_asapiref_en.

Each generated value object has a managingService member variable of type IFiberManagingService. It is the responsibility of the client-side data management layer to initialize this variable as soon as the value object instance becomes managed.

**Derived Properties with Function References**

The calculation of derived properties that do not depend on external function calls can occur synchronously. However, the calculation of derived properties that reference at least one external function must take place asynchronously. When the value of an uncached derived property is requested, an _ItemPendingError is thrown and the calculation is triggered. Once the calculation completes, the value is cached and a property change event is triggered for the derived property. Any existing bindings to the property execute as soon as the calculation is complete.

There is no way to predict when a cached value of a derived property that depends on an external function call must be invalidated. For example, suppose a remote function that calculates a person's credit score returns different values with time even if its argument does not change, while the return value of a remote function that calculates a city's distance to New York city does not change over time as long as its argument does not change. By default, it is assumed that external functions are not volatile and it is up to the consumer of value objects to invalidate the cached value of a derived property by calling the invalidateDependentOnpropertyName() method for one of the argument properties of the function if its value may have changed even though its arguments have not. This invalidates the derived property cache and triggers a recalculation of the derived property. As above, an appropriate property change event is triggered when the calculation is complete.

**Style implementation**

A style is a collection of user-interface-related attributions that you can reuse across properties in a model or define for a specific property. A generated value object can provide a property's style to querying consumers.

In the following example model, the style API in a generated value object returns different values depending on whether the current country is US or UK.
For the following example model, a Flex validator provides validation of the price property when you generate a model-driven Form for the Product entity in Flash Builder. See the source code for the corresponding model-driven Form in “Model-driven Form generation” on page 192.
<model xmlns="http://ns.adobe.com/Fiber/1.0">
    <annotation name="DMS">
        <item name="datasource">java:/comp/env/jdbc/ordersDB</item>
        <item name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</item>
    </annotation>

    <style name="priceValidation">
        <validation text="Price needs to be greater than 10">
            <expr><![CDATA[value > 10]]></expr>
        </validation>
    </style>

    <entity name="Product" persistent="true">
        <annotation name="ServerProperties">
            <item name="ServerType">LCDS</item>
        </annotation>

        <annotation name="DMS">
            <item name="Table">PUBLIC.PRODUCT</item>
        </annotation>

        <id name="productid" type="integer">
            <annotation name="DMS">
                <item name="ColumnName">PRODUCTID</item>
            </annotation>
        </id>

        <property name="description" type="string" length="255">
            <annotation name="DMS">
                <item name="ColumnName">DESCRIPTION</item>
            </annotation>
        </property>

        <property name="price" style="priceValidation" type="float">
            <annotation name="DMS">
                <item name="ColumnName">PRICE</item>
            </annotation>
        </property>

        <property name="productname" type="string" length="255">
            <annotation name="DMS">
                <item name="ColumnName">PRODUCTNAME</item>
            </annotation>
        </property>
    </entity>
</model>

Lazy loading and loading on demand

Note: This content is specific to working with the LiveCycle Data Services Data Management Service.

When using associations between objects, the LiveCycle Data Services Data Management Service supports lazy loading in which only id properties are loaded initially, and loading on demand in which nothing is initially loaded. It also supports loading on demand for properties of items. Initial access to such properties of a value object result in an ItemPendingError and trigger the retrieval of the missing properties. Bindings to these properties execute once the property is retrieved.

Lazy loading and loading on demand are not supported for client-side data management where the backend is not LiveCycle Data Services. In these cases, all properties are populated initially and if they are found to be null, it is assumed that they are missing.
Service wrappers

The ActionScript generator creates ActionScript service wrapper classes from the service and entity elements and their associated annotations in a model. A service declaration in a model results in the generation of wrappers around three types of underlying services:

1. Custom service for which the user provides the implementation
2. Client-side RPC service: RemoteObject, HTTPMultiService, or WebService
3. Client-side RPC service associated with one or more DataManager objects

Service wrapper generation for case one is addressed in “References to service functions in entities” on page 160.

The generated service wrappers for cases 2 and 3 provide strongly typed entry points to service operations, which in turn enable tooling to perform code hinting and code completion. The wrappers also configure the wrapped service from information contained in the associated model.

In Flash Builder workflows that involve generated service wrappers, you select a service declared in a model from the service explorer and drag one of its operations onto the design view canvas. Code is generated in the following cases:

- Invocation of the service and binding to a user interface component
- Service object used to make the invocation
- Objects that the invocation returns

The generated service wraps one of three types of RPC service objects (RemoteObject, HTTPMultiService, or WebService) or an implicit LiveCycle Data Services DataService object based solely on an entity.

If the wrapped service is an RPC service, you can manage its data by associating one or more DataManager objects with it. If the wrapped service is an implicit LiveCycle Data Services DataService object, its data is already managed.

Metadata describing the wrapped services and their related entities is stored in the model as follows:

- Service signatures are stored as services and functions.
- Data type definitions that services use are stored as entities.
- Service metadata such as URLs and ports are stored as annotations on relevant services.
- Associations between services and data managers that manage their data are stored as annotations on entities.
- Metadata that specifies data managed functions of a service are stored as annotations on entities.

Wrapping RPC services

For an RPC service, the service wrapper provides strongly typed invocation points (functions) for every operation of the service. These functions are represented as function child element of a service element in a model. Since each such invocation is asynchronous, each function returns an AsyncToken token. The wrapper configures less volatile properties of its service while leaving the more volatile ones for the developer to set in MXML. Service configuration options that cannot be expressed in first-class model syntax are stored as annotations of the service and function elements as described in “ActionScript objects” on page 157.

Model configuration for a RemoteObject service

For a RemoteObject service, the following configuration options are presented and stored as ActionScriptGeneration group annotations of the corresponding service in a model:

- A ServiceType=RemoteObject annotation on the service element in a model marks it as a RemoteObject service.
- A Package annotation allows the user to customize the package and generated location for this service.
The following model, generated by a data service wizard in Flash Builder, contains a service element annotated as a RemoteObject service:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <annotation name="ActionScriptGeneration">
    <item name="FullyQualifyReferences">true</item>
  </annotation>
  <service name="ProductService">
    <annotation name="ActionScriptGeneration">
      <item name="ServiceType">RemoteObject</item>
      <item name="Package">services.productservice</item>
      <item name="Destination">productService</item>
    </annotation>
    <annotation name="ASFilesGenerated">
      <item name="ASFilesGenerated">
        services/productservice/_Super_ProductService.as~#~services/productservice/ProductService.as~#~
      </item>
    </annotation>
    <annotation name="ServiceConfig">
      <item name="DEFAULT_ENTITY_PACKAGE">valueObjects</item>
      <item name="contextRoot">/lcds-samples</item>
      <item name="rootUrl">http://localhost:8400/lcds-samples</item>
      <item name="LINKED_FILE"></item>
      <item name="ABSOLUTE_ENDPOINT">
        http://localhost:8400/lcds-samples/messagebroker/amf
      </item>
    </annotation>
    <annotation name="ServiceTypeGroup">
      <item name="ServiceBackendType">LCDS</item>
    </annotation>
    <function name="getProducts" return-type="Product[]">
      <annotation name="analyze group">
        <item name="analyzed">true</item>
      </annotation>
      <annotation name="original signature">
        <item name="inputParams">Product</item>
      </annotation>
    </function>
    <function name="updateProduct" arguments="arg0:Product" return-type="void">
      <annotation name="analyze group">
        <item name="analyzed">true</item>
      </annotation>
      <annotation name="original signature">
        <item name="inputParams">Product</item>
        <item name="returnType">void</item>
      </annotation>
    </function>
    <function name="getProductsByName" arguments="arg0:string" return-type="opaque[]">
      <annotation name="original signature">
        <item name="inputParams">string</item>
        <item name="returnType">opaque</item>
      </annotation>
    </function>
    <function name="createProduct" arguments="arg0:Product" return-type="Product">
      <annotation name="analyze group">
        <item name="analyzed">true</item>
      </annotation>
    </function>
  </service>
</model>
```
The following example shows the source code of a RemoteObject service wrapper superclass generated from a RemoteObject service. The service wrappers for HTTPService services and WebService services are similar.
package services.productservice
{
import mx.rpc.AsyncToken;
import com.adobe.fiber.core.model_internal;
import mx.rpc.AbstractOperation;
import valueObjects.Product;
import mx.collections.ItemResponder;
import mx.rpc.remoting.RemoteObject;
import mx.rpc.remoting.Operation;
import com.adobe.fiber.services.wrapper.RemoteObjectServiceWrapper;
import com.adobe.fiber.valueobjects.AvailablePropertyIterator;
import com.adobe.serializers.utility.TypeUtility;

internal class _Super_ProductService extends RemoteObjectServiceWrapper
{

    // Constructor
    public function _Super_ProductService()
    {
        // initialize service control
        _serviceControl = new RemoteObject();

        var operations:Object = new Object();
        var operation:Operation;
        operation = new Operation(null, "getProducts");
        operation.resultElementType = valueObjects.Product;
        operations["getProducts"] = operation;
        valueObjects.Product._initRemoteClassAlias();

        operation = new Operation(null, "updateProduct");
        operation.resultElementType = Object;
        operations["updateProduct"] = operation;
        operation = new Operation(null, "getProductsByName");
        operation.resultElementType = Object;
        operations["getProductsByName"] = operation;
        operation = new Operation(null, "createProduct");
        operation.resultType = valueObjects.Product;
        operations["createProduct"] = operation;
        valueObjects.Product._initRemoteClassAlias();

        operation = new Operation(null, "deleteProduct");
        operation.resultType = Boolean;
        operations["deleteProduct"] = operation;
        operation = new Operation(null, "getProduct");
        operation.resultType = valueObjects.Product;
        operations["getProduct"] = operation;
        valueObjects.Product._initRemoteClassAlias();

        _serviceControl.operations = operations;
        _serviceControl.convertResultHandler = TypeUtility.convertResultHandler;
        _serviceControl.destination = "productService";

        model_internal::initialize();
    }
}

/**
 * This method is a generated wrapper used to call the 'getProducts' operation. It returns an AsyncToken whose
* result property will be populated with the result of the operation when the server response is received.
* To use this result from MXML code, define a CallResponder component and assign its token property to this method’s return value.
* You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
  *
  * @see mx.rpc.AsyncToken
  * @see mx.rpc.CallResponder
  *
  * @return an AsyncToken whose result property will be populated with the result of the operation when the server response is received.
  */
public function getProducts() : AsyncToken
{
    var _internal_operation:AbstractOperation =
_serviceControl.getOperation("getProducts");
    var _internal_token:AsyncToken = _internal_operation.send();
    return _internal_token;
}

/**
* This method is a generated wrapper used to call the 'updateProduct' operation. It returns an AsyncToken whose
* result property will be populated with the result of the operation when the server response is received.
* To use this result from MXML code, define a CallResponder component and assign its token property to this method’s return value.
* You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
  *
  * @see mx.rpc.AsyncToken
  * @see mx.rpc.CallResponder
  *
  * @return an AsyncToken whose result property will be populated with the result of the operation when the server response is received.
  */
public function updateProduct(arg0:valueObjects.Product) : AsyncToken
{
    var _internal_operation:AbstractOperation =
_serviceControl.getOperation("updateProduct");
    var _internal_token:AsyncToken = _internal_operation.send(arg0);
    return _internal_token;
}

/**
* This method is a generated wrapper used to call the 'getProductsByName' operation. It returns an AsyncToken whose
* result property will be populated with the result of the operation when the server response is received.
* To use this result from MXML code, define a CallResponder component and assign its token property to this method’s return value.
* You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
  *
  * @see mx.rpc.AsyncToken
  * @see mx.rpc.CallResponder
  *
public function getProductsByName(arg0:String) : AsyncToken
{
    var _internal_operation:AbstractOperation =
        _serviceControl.getOperation("getProductsByName");
    var _internal_token:AsyncToken = _internal_operation.send(arg0) ;
    return _internal_token;
}

/**
 * This method is a generated wrapper used to call the 'createProduct' operation. It returns
 * an AsyncToken whose result property will be populated with the result of the operation when the server
 * response is received.
 * To use this result from MXML code, define a CallResponder component and assign its token
 * property to this method's return value.
 * You can then bind to CallResponder.lastResult or listen for the CallResponder.result or
 * fault events.
 * @see mx.rpc.AsyncToken
 * @see mx.rpc.CallResponder
 * @return an AsyncToken whose result property will be populated with the result of the
 * operation when the server response is received.
 */
public function createProduct(arg0:valueObjects.Product) : AsyncToken
{
    var _internal_operation:AbstractOperation =
        _serviceControl.getOperation("createProduct");
    var _internal_token:AsyncToken = _internal_operation.send(arg0) ;
    return _internal_token;
}

/**
 * This method is a generated wrapper used to call the 'deleteProduct' operation. It returns
 * an AsyncToken whose result property will be populated with the result of the operation when the server
 * response is received.
 * To use this result from MXML code, define a CallResponder component and assign its token
 * property to this method's return value.
 * You can then bind to CallResponder.lastResult or listen for the CallResponder.result or
 * fault events.
 * @see mx.rpc.AsyncToken
 * @see mx.rpc.CallResponder
 * @return an AsyncToken whose result property will be populated with the result of the
 * operation when the server response is received.
 */
public function deleteProduct(arg0:valueObjects.Product) : AsyncToken
{
    var _internal_operation:AbstractOperation =
        _serviceControl.getOperation("deleteProduct");
    var _internal_token:AsyncToken = _internal_operation.send(arg0) ;
    return _internal_token;
}
Model configuration for an HTTPService service

You cannot map from a model to a standard HTTPService service because the modeling language allows multiple functions (operations) to be associated with a single service while the HTTPService class only supports a single operation. Instead, the model uses the new HTTPMultiService service.

For an HTTPMultiService service, the following configuration options are presented and stored as ActionScriptGeneration group annotations of the corresponding service in a model:

- A ServiceType=HTTPMultiService annotation on the service in a model marks it as an HTTPMultiService service.
- When the service communicates directly with its source, the rootURL property is configurable in the MXML declaration when you declare an instance on the service in an MXML application. The ActionScript generator configures operation-specific URL extensions in the service wrapper class.
- The rootURL is annotated on the service in a rootURL annotation.
- When service communication is proxied through BlazeDS or LiveCycle Data Services, destination name is not annotated in the model and is set in MXML when an instance of the wrapper is declared.
- Each function of the HTTPMultiService is annotated with a method annotation.
- The result type for each operation is determined from the entity specified as the return type for the function.

The following model, generated by a data service wizard in Flash Builder, contains a service element annotated as an HTTPMultiService service:
Model configuration for a WebService service

For a WebService service, the following configuration options are presented and stored as ActionScriptGeneration group annotations of the corresponding service in a model:

- A ServiceType=WebService annotation on the service element in a model marks it is a WebService service.
- The wsdl property is annotated on the service in a WSDL annotation and is configured from MXML when an instance of the wrapper is declared.
- The service property is annotated on the model service in a WSDL-service annotation and is configured from MXML when an instance of the wrapper is declared.
• The port property is annotated on the model service in a WSDL-port annotation and is configured from MXML when an instance of the wrapper is declared.

The following model, generated by a data service wizard in Flash Builder, contains a service element annotated as a WebService service:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <annotation name="ActionScriptGeneration">
    <item name="FullyQualifyReferences">true</item>
  </annotation>
  <service name="WsCatalog">
    <annotation name="ActionScriptGeneration">
      <item name="WSDL-service">XMLProductServiceService</item>
      <item name="ServiceType">WebService</item>
      <item name="Package">services.wscatalog</item>
      <item name="Destination">ws-catalog</item>
      <item name="WSDL-port">XMLProductService</item>
    </annotation>
    <annotation name="ASFilesGenerated">
      <item name="ASFilesGenerated">
        services/wscatalog/_Super_WsCatalog.as~#~services/wscatalog/WsCatalog.as~#~
      </item>
    </annotation>
    <annotation name="ServiceConfig">
      <item name="DEFAULT_ENTITY_PACKAGE"></item>
      <item name="contextRoot">/lcds-samples</item>
      <item name="rootUrl">http://localhost:8400/lcds-samples</item>
      <item name="WSDL-endpoint">
        http://hiro-xp:8080/soap/services/XMLProductService
      </item>
      <item name="LINKED_FILE"></item>
    </annotation>
    <annotation name="ASGenerationWSDLTypeMap">
      <item name="XML_type">
        http://adobe.com/idp/services,XML
      </item>
    </annotation>
    <annotation name="ServiceTypeGroup">
      <item name="ServiceBackendType">WebService</item>
    </annotation>
    <function name="GetProducts" return-type="XML_type">
      <annotation name="analyze group">
        <item name="analyzed">true</item>
      </annotation>
      <annotation name="WSDLName">
        <item name="GetProducts">
          http://adobe.com/idp/services,GetProducts
        </item>
      </annotation>
      <annotation name="original signature">
        <item name="returnType">XML_type</item>
      </annotation>
    </function>
  </service>
  <entity name="XML_type">
    <annotation name="EntityProperties">
```
<item name="EntityOriginalName">XML_type</item>
</annotation>
<annotation name="service Group">
  <item name="service">WsCatalog</item>
</annotation>
<annotation name="ServerProperties">
  <item name="IncludeRemoteClass">nonempty</item>
</annotation>
<annotation name="ActionScriptGeneration">
  <item name="Package"></item>
</annotation>
<annotation name="ASFilesGenerated">
  <item name="ASFilesGenerated">
    /_Super_XML_type.as~#~/_XML_typeEntityMetadata.as~#~/XML_type.as~#~
  </item>
</annotation>
<property name="document" type="string" required="true"/>
<property name="element" type="string" required="true"/>
</entity>
</model>

Service wrapper classes
The superclass hierarchy for generated service wrappers consists of the following non-generated classes, which are packaged in the fiber.swc file:

- AbstractServiceWrapper is at the top of the class hierarchy. It declares the service and provides the following functionality common to all generated wrappers regardless of type: event propagation, destination, channel set, and operations properties.
  - RemoteObjectServiceWrapper extends AbstractServiceWrapper, instantiates a service as a RemoteObject, and provides access to RemoteObject-specific properties.
  - HTTPServiceWrapper extends AbstractServiceWrapper, instantiates a service as an HTTPMultiService, and provides access to HTTPMultiService-specific properties.
  - WebServiceWrapper extends AbstractServiceWrapper, instantiates a service as a WebService, and provides access to WebService-specific properties.
  - DataServiceWrapper extends AbstractServiceWrapper, instantiates a service as a DataService, and provides access to DataService-specific properties. Used for implicit services in the case of a LiveCycle Data Services backend.

The generated service wrapper classes are named _ServiceName.as and ServiceName.as.
_ServiceName.as extends RemoteObjectServiceWrapper, HTTPServiceWrapper, or WebServiceWrapper, depending on the type of service that it is wrapping. ServiceName.as extends _ServiceName.as and is initially empty. Its purpose is to let developers partially customize the behavior of the wrapper.

ServiceName.as is only generated if it does not exist in the target output folder, allowing custom user behavior to persist when code is regenerated from the model.

Enabling client-side data management for RPC services
When a particular RPC service is described in a model, you can enable client-side data management features for a subset of the service's operations. Client-side data management provides the following data management features with model-driven development. Unlike the Data Management Service on the LiveCycle Data Services server, client-side data management is for RPC services and only manages data on the client.
To enable client-side data management, an RPC service must have one or more data manager objects associated with the entities that it works with. You can add query function annotations to the entity data manager, which the managing service can call. Client-side data management is enabled for the Product entity in the following model. This model also specifies query function annotations for the ProductService, which is the service that manages the Product entity:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Change required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh of unique value objects in Flex clients.</td>
<td>Add an identity property to an entity type of the remote service and an id element on the corresponding entity element in the model.</td>
</tr>
<tr>
<td>Population of value objects in Flex clients with managed data.</td>
<td>Add zero or more getItem, findItem, and fill functions to the remote service and corresponding annotations to the entity in the model.</td>
</tr>
<tr>
<td>Managed updates in Flex clients.</td>
<td>Add one or more create, update, or delete functions to the model and corresponding annotations to the entity in the model.</td>
</tr>
<tr>
<td>Paging of entity instances to Flex clients.</td>
<td>Add a count parameter to a query in the remote service and corresponding annotations to the entity in the model. The count function specified in the model must map to a method on the remote service that provides start and count parameters.</td>
</tr>
</tbody>
</table>
<model xmlns="http://ns.adobe.com/Fiber/1.0">
<entity name="Product" persistent="true">
<!-- This is not a complete model. It demonstrates client-side data management annotations -->
<annotation name="DataManagement">
  <item name="managingService">ProductService</item>
  <item name="create">createProduct</item>
  <item name="update">updateProduct</item>
  <item name="get">getProduct</item>
  <item name="delete">deleteProduct</item>
  <item name="fillFunc">getProducts</item>
</annotation>
<annotation name="createProduct">
  <item name="propertySpecifier">
    productId, qtyInStock, price, category, description, image, name
  </item>
</annotation>
<annotation name="getProduct">
  <item name="propertySpecifier">
    productId, qtyInStock, price, category, description, image, name
  </item>
</annotation>
<annotation name="getProducts">
  <item name="propertySpecifier">
    productId, qtyInStock, price, category, description, image, name
  </item>
</annotation>

$id name="productId" type="integer"/>
<property name="qtyInStock" type="integer" required="true"/>
<property name="price" type="double" required="true"/>
<property name="category" type="string" required="true"/>
<property name="description" type="string" required="true"/>
<property name="image" type="string" required="true"/>
<property name="name" type="string" required="true"/>
</entity>
<!-- ProductService implementation would go here. -->
</model>

There is a subset of functions that data management calls implicitly in response to user interface gestures that modify a variable the service is bound to. These functions are the updateItem(), deleteItem(), and createItem() functions. For example, suppose a Flex client calls fill() on a service and binds the results to an ArrayCollection. When the client updates an item in that collection and then calls the updateItem() method, the service implicitly makes the updateItem() invocation on the remote source that corresponds to it.

For a data manager on an RPC service to call these functions implicitly, you must explicitly identify the functions in function-level annotations in a model.

In addition to being implicitly available, the generated service wrapper exposes the identified functions as strong typed invocation points, allowing a smooth path from non-managed services to managed services. A wrapper for an unmanaged RemoteObjectService could already have an operation that could later become a managed service operation.

You can configure one or more of the create, delete, and update functions for each data manager. You can also modify the parameters of the operations; for example, deleteBook(Book.id) instead of deleteBook(id).
In addition to specifying the above modifier functions, specify the functions that populate the client with managed data:

- A single function as the getItem function; the arguments to this function in the generated wrapper are always a strongly typed list of identity properties of the corresponding entity
- Zero or more functions as findItem functions
- Zero or more functions as the fill functions

When these functions are called, the service handles them appropriately with respect to client-side data management by inspecting the values returned from the server. Functions of the decorated service that are not mapped to by any data managers continue to act as standard RPC functions.

The ProductService service in the following example works with Product entities. It was configured for client-side data management when a developer selected Enable Data Management on the Product data type in the Flash Builder project. That user gesture resulted in changes to the model, which in turn triggered generation of new ActionScript code to support data management. Note that the Product entity contains a DataManagement annotation and the service functions are changed to perform client-side data management operations.

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <annotation name="ActionScriptGeneration">
    <item name="FullyQualifyReferences">true</item>
  </annotation>
  <entity name="Product" persistent="true">
    <annotation name="EntityProperties">
      <item name="EntityOriginalName">Product</item>
    </annotation>
    <annotation name="ServerProperties">
      <item name="RemoteClass">flex.samples.dcd.product.Product</item>
    </annotation>
    <annotation name="service Group">
      <item name="service">ProductService</item>
    </annotation>
    <annotation name="ActionScriptGeneration">
      <item name="Package">valueObjects</item>
    </annotation>
    <annotation name="ASFilesGenerated">
      <item name="ASFilesGenerated">
        valueObjects/_ProductEntityMetadata.as~#~valueObjects/_Super_Product.as~#~valueObjects/Product.as~#~
      </item>
    </annotation>
    <annotation name="DataManagement">
      <item name="managingService">ProductService</item>
      <item name="create">createProduct</item>
      <item name="update">updateProduct</item>
      <item name="get">getProduct</item>
      <item name="delete">deleteProduct</item>
      <item name="fillFunc">getProducts</item>
    </annotation>
    <annotation name="createProduct">
      <item name="propertySpecifier">productId,qtyInStock,price,category,description,image,name</item>
    </annotation>
    <annotation name="getProduct">
      <item name="propertySpecifier">productId,qtyInStock,price,category,description,image,name</item>
    </annotation>
  </entity>
</model>
```
<item name="productId" type="integer"/>
<property name="qtyInStock" type="integer" required="true"/>
<property name="price" type="double" required="true"/>
<property name="category" type="string" required="true"/>
<property name="description" type="string" required="true"/>
<property name="image" type="string" required="true"/>
<property name="name" type="string" required="true"/>
</entity>
<service name="ProductService">
  <annotation name="ActionScriptGeneration">
    <item name="ServiceType">RemoteObject</item>
    <item name="Package">services.productservice</item>
    <item name="Destination">productService</item>
  </annotation>
  <annotation name="ASFilesGenerated">
    <item name="ASFilesGenerated">
      services/productservice/_Super_ProductService.as~#~services/productservice/ProductService.as
    </item>
  </annotation>
  <annotation name="ServiceConfig">
    <item name="DEFAULT_ENTITY_PACKAGE">valueObjects</item>
    <item name="contextRoot">/lcds-samples</item>
    <item name="rootUrl">http://localhost:8400/lcds-samples</item>
    <item name="LINKED_FILE"></item>
    <item name="ABSOLUTE_ENDPOINT">
      http://localhost:8400/lcds-samples/messagebroker/amf
    </item>
  </annotation>
  <annotation name="ServiceTypeGroup">
    <item name="ServiceBackendType">LCDS</item>
  </annotation>
  <function name="createProduct" arguments="arg0:Product" return-type="Product">
    <annotation name="analyze group">
      <item name="analyzed">true</item>
    </annotation>
    <annotation name="original signature">
      <item name="inputParams">Product</item>
      <item name="returnType">Product</item>
    </annotation>
  </function>
  <function name="deleteProduct" arguments="arg0:Product" return-type="boolean">
    <annotation name="analyze group">
      <item name="analyzed">true</item>
    </annotation>
    <annotation name="original signature">
      <item name="inputParams">Product</item>
      <item name="returnType">boolean</item>
    </annotation>
  </function>
</service>

<annotation name="getProducts">
  <item name="propertySpecifier">
    productId,qtyInStock,price,category,description,image,name
  </item>
</annotation>
When you enable client-side data management, you can add query function annotations to the entity data manager, as the following example shows. In the case of Flash Builder, these annotations are automatically generated when you enable data management for a data type in the Data/Services panel. The get annotation specifies the get function, and fillFunc annotation specifies the fill functions.
To enable client-side data paging for an RPC service, the remote service must contain the following operations:

- A `count` operation that returns number of rows in the database. The following example shows a `count` operation in a PHP service:
public function count() {
    $stmt = mysqli_prepare($this->connection, "SELECT COUNT(*) AS COUNT FROM $this->tablename");
    $this->throwExceptionOnError();
    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();
    mysqli_stmt_bind_result($stmt, $rec_count);
    $this->throwExceptionOnError();
    mysqli_stmt_fetch($stmt);
    $this->throwExceptionOnError();
    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);
    return $rec_count;
}

A `getItems_paged` operation that takes two input arguments and returns the data collection. The two input arguments are `startIndex` and `numberOfItems`. The following example shows a `getItems_paged` operation in a PHP service:

```php
public function getItems_paged ($startIndex, $numItems) {
    $stmt = mysqli_prepare($this->connection, "SELECT CUSTOMER_ID,CUSTOMER_NAME,CUSTOMER_TYPE, CUSTOMER_ADDRESS,ENTRY_MODIFIED_DATE FROM $this->tablename LIMIT ?, ?");
    $this->throwExceptionOnError();
    mysqli_bind_param($stmt, 'ii', $startIndex, $numItems);
    mysqli_stmt_execute($stmt);
    $this->throwExceptionOnError();
    $rows = array();
    mysqli_stmt_bind_result($stmt, $row->customerId,$row->customerName,$row->customerType,$row->customerAddress,$row->entryModifiedDate);
    while (mysqli_stmt_fetch($stmt)) {
        $rows[] = $row;
    }
    mysqli_stmt_free_result($stmt);
    mysqli_close($this->connection);
    return $rows;
}
```

In the corresponding model, you use the `GetPagedOperation` annotation group to specify the count operation and enable paging. In the case of Flash Builder, the annotations are automatically generated when you enable paging for a data type in the Data/Services panel. The following example shows this part of a model:
<model xmlns= "http://ns.adobe.com/Fiber/1.0">
<entity name="Product" persistent="true">
<!-- This is not a complete model. It shows just client-side paging annotations. -->
<annotation name="getPagedProducts">
  <item name="countOperation">countProducts</item>
  <item name="pagingEnabled">true</item>
</annotation>
{id name="productId" type="integer"/>
<property name="qtyInStock" type="integer" required="true"/>
<property name="price" type="double" required="true"/>
<property name="category" type="string" required="true"/>
<property name="description" type="string" required="true"/>
<property name="image" type="string" required="true"/>
<property name="name" type="string" required="true"/>
</entity>
<!-- Service implementation would go here. -->
</model>

The following example shows the source code of a data-managed RemoteObject service wrapper generated from the ProductService service in the model. The service wrappers for data-managed HTTPService services and WebService services are similar.

package services.productservice
{
import mx.rpc.AsyncToken;
import com.adobe.fiber.core.model_internal;
import mx.rpc.AbstractOperation;
import valueObjects.Product;
import mx.data.RPCDataManager;
import mx.data.ManagedOperation;
import mx.data.ManagedAssociation;
import mx.data.ManagedQuery;
import mx.data.ItemReference;
import mx.collections.ItemResponder;
import mx.rpc.remoting.RemoteObject;
import mx.rpc.remoting.Operation;
import com.adobe.fiber.services.wrapper.RemoteObjectServiceWrapper;
import com.adobe.fiber.valueobjects.AvailablePropertyIterator;
import com.adobe.serializers.utility.TypeUtility;
[ExcludeClass]
internal class _Super_ProductService extends RemoteObjectServiceWrapper
{
  private var _productRPCDataManager : RPCDataManager;
  private var managersArray : Array = new Array();

  public const DATA_MANAGER_PRODUCT : String = "Product";

  public function getDataManager(dataManagerName:String) : RPCDataManager
  {
    switch (dataManagerName)
    {
      case (DATA_MANAGER_PRODUCT):
        return _productRPCDataManager;
      default:
        return null;
    }
  }
/**
 * Commit all of the pending changes for this DataService, as well as all of the pending
 * changes of all DataServices sharing the same DataStore. By default, a DataService shares the same DataStore with
 * other DataServices if they have managed association properties and share the same set of channels.
 *
 * @see mx.data.DataManager
 * @see mx.data.DataStore
 *
 * @param itemsOrCollections:Array This is an optional parameter which defaults to null when
 * you want to commit all pending changes. If you want to commit a subset of the pending
 * changes use this argument to specify a list of managed ListCollectionView instances
 * and/or managed items. ListCollectionView objects are most typically ArrayCollections
 * you have provided to your fill method. The items appropriate for this method are
 * any managed version of the item. These are any items you retrieve from getItem, createItem
 * or using the getItemAt method from a managed collection. Only changes for the
 * items defined by any of the values in this array will be committed.
 *
 * @param cascadeCommit if true, also commit changes made to any associated
 * items supplied in this list.
 *
 * @return AsyncToken that is returned in <code>call</code> property of
 * either the <code>ResultEvent.RESULT</code> or in the
 * <code>FaultEvent.FAULT</code>.
 * Custom data can be attached to this object and inspected later
during the event handling phase. If no changes have been made
to the relevant items, null is returned instead of an AsyncToken.
*/
public function commit(itemsOrCollections:Array=null,
cascadeCommit:Boolean=false):AsyncToken
{
    return _productRPCDataManager.dataStore.commit(itemsOrCollections, cascadeCommit);
}

/**
 * Reverts all pending (uncommitted) changes for this DataService, as well as all of the pending
 * changes of all DataServices sharing the same DataStore. By default, a DataService shares the same DataStore with
 * other DataServices if they have managed association properties and share the same set of channels.
 *
 * @see mx.data.DataManager
 * @see mx.data.DataStore
 *
 * @return true if any changes were reverted.
 */
public function revertChanges():Boolean
{
    return _productRPCDataManager.dataStore.revertChanges();
}

// Constructor
public function _Super_ProductService()
{
    // initialize service control
    _serviceControl = new RemoteObject();

    var operations: Object = new Object();
    var operation: Operation;

    operation = new Operation(null, "createProduct");
    operation.resultType = valueObjects.Product;
    operations["createProduct"] = operation;
    valueObjects.Product._initRemoteClassAlias();
    operation = new Operation(null, "deleteProduct");
    operation.resultType = Boolean;
    operations["deleteProduct"] = operation;
    operation = new Operation(null, "getProduct");
    operation.resultType = valueObjects.Product;
    operations["getProduct"] = operation;
    valueObjects.Product._initRemoteClassAlias();
    operation = new Operation(null, "getProducts");
    operation.resultElementType = valueObjects.Product;
    operations["getProducts"] = operation;
    valueObjects.Product._initRemoteClassAlias();
    operation = new Operation(null, "getProductsByName");
    operation.resultElementType = Object;
    operations["getProductsByName"] = operation;
    operation = new Operation(null, "updateProduct");
    operations["updateProduct"] = operation;

    _serviceControl.operations = operations;
    _serviceControl.convertResultHandler = TypeUtility.convertResultHandler;
    _serviceControl.destination = "productService";

    var managedAssociation : ManagedAssociation;
    var managedAssocsArray : Array;
    // initialize Product data manager
    _productRPCDataManager = new RPCDataManager();
    managersArray.push(_productRPCDataManager);

    managedAssocsArray = new Array();
    _productRPCDataManager.destination = "productRPCDataManager";
    _productRPCDataManager.service = _serviceControl;
    _productRPCDataManager.identities = "productId";
    _productRPCDataManager.itemClass = valueObjects.Product;

    var dmOperation : ManagedOperation;
    var dmQuery : ManagedQuery;

    dmOperation = new ManagedOperation("deleteProduct", "delete");
    dmOperation.parameters = "item";
    _productRPCDataManager.addManagedOperation(dmOperation);

    dmQuery = new ManagedQuery("getProducts");
    dmQuery.propertySpecifier = "productId,qtyInStock,price,category,description,image,name";
dmQuery.parameters = "";
_productRPCDataManager.addManagedOperation(dmQuery);
dmOperation = new ManagedOperation("getProduct", "get");
dmOperation.parameters = "productId";
_productRPCDataManager.addManagedOperation(dmOperation);

dmOperation = new ManagedOperation("createProduct", "create");
dmOperation.parameters = "item";
_productRPCDataManager.addManagedOperation(dmOperation);

dmOperation = new ManagedOperation("updateProduct", "update");
dmOperation.parameters = "item";
_productRPCDataManager.addManagedOperation(dmOperation);

_serviceControl.managers = managersArray;

model_internal::initialize();
}
/**
 * This method is a generated wrapper used to call the 'createProduct' operation. It returns an ItemReference whose
 * result property will be populated with the result of the operation when the server response is received.
 * To use this result from MXML code, define a CallResponder component and assign its token property to this method's return value.
 * You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
 * @see mx.data.ItemReference
 * @see mx.rpc.CallResponder
 * @return an ItemReference whose result property will be populated with the result of the operation when the server response is received.
 */
public function createProduct(arg0:valueObjects.Product) : ItemReference
{
    var _internal_operation:AbstractOperation =
        _serviceControl.getOperation("createProduct");
    var _internal_token:ItemReference = _internal_operation.send(arg0) as ItemReference;
    return _internal_token;
}
/**
 * This method is a generated wrapper used to call the 'deleteProduct' operation. It returns an AsyncToken whose
 * result property will be populated with the result of the operation when the server response is received.
 * To use this result from MXML code, define a CallResponder component and assign its token property to this method's return value.
 * You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
 * @see mx.rpc.AsyncToken
 * @see mx.rpc.CallResponder
 * @return an AsyncToken whose result property will be populated with the result of the operation when the server response is received.
 */
public function deleteProduct(arg0:valueObjects.Product) : AsyncToken
{
    var _internal_operation:AbstractOperation =
    _serviceControl.getOperation("deleteProduct");
    var _internal_token:AsyncToken = _internal_operation.send(arg0) ;
    return _internal_token;
}

/**
* This method is a generated wrapper used to call the 'getProduct' operation. It returns
an ItemReference whose
* result property will be populated with the result of the operation when the server
response is received.
* To use this result from MXML code, define a CallResponder component and assign its token
property to this method's return value.
* You can then bind to CallResponder.lastResult or listen for the CallResponder.result or
fault events.
* @see mx.data.ItemReference
* @see mx.rpc.CallResponder
* @return an ItemReference whose result property will be populated with the result of the
operation when the server response is received.
*/
public function getProduct(arg0:int) : ItemReference
{
    var _internal_operation:AbstractOperation = _serviceControl.getOperation("getProduct");
    var _internal_token:ItemReference = _internal_operation.send(arg0) as ItemReference;
    return _internal_token;
}

/**
* This method is a generated wrapper used to call the 'getProducts' operation. It returns
an AsyncToken whose
* result property will be populated with the result of the operation when the server
response is received.
* To use this result from MXML code, define a CallResponder component and assign its token
property to this method's return value.
* You can then bind to CallResponder.lastResult or listen for the CallResponder.result or
fault events.
* @see mx.rpc.AsyncToken
* @see mx.rpc.CallResponder
* @return an AsyncToken whose result property will be populated with the result of the
operation when the server response is received.
*/
public function getProducts() : AsyncToken
{
    var _internal_operation:AbstractOperation = _serviceControl.getOperation("getProducts");
    var _internal_token:AsyncToken = _internal_operation.send() ;
    return _internal_token;
}
* This method is a generated wrapper used to call the 'getProductsByName' operation. It returns an AsyncToken whose
  * result property will be populated with the result of the operation when the server response is received.
  * To use this result from MXML code, define a CallResponder component and assign its token property to this method's return value.
  * You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
  *
  * @see mx.rpc.AsyncToken
  * @see mx.rpc.CallResponder
  *
  * @return an AsyncToken whose result property will be populated with the result of the operation when the server response is received.
  */
public function getProductsByName(arg0:String) : AsyncToken
{
    var _internal_operation:AbstractOperation =
    _serviceControl.getOperation("getProductsByName");
    var _internal_token:AsyncToken = _internal_operation.send(arg0) ;
    return _internal_token;
}

/**
 * This method is a generated wrapper used to call the 'updateProduct' operation. It returns an ItemReference whose
 * result property will be populated with the result of the operation when the server response is received.
 * To use this result from MXML code, define a CallResponder component and assign its token property to this method's return value.
 * You can then bind to CallResponder.lastResult or listen for the CallResponder.result or fault events.
 *
 * @see mx.data.ItemReference
 * @see mx.rpc.CallResponder
 *
 * @return an ItemReference whose result property will be populated with the result of the operation when the server response is received.
 * */
public function updateProduct(arg0:valueObjects.Product) : ItemReference
{
    var _internal_operation:AbstractOperation =
    _serviceControl.getOperation("updateProduct");
    var _internal_token:ItemReference = _internal_operation.send(arg0) as ItemReference;
    return _internal_token;
}
Implicit service generation

When a LiveCycle Data Services Data Management Service destination backs an entity, an explicit service declaration is not necessary to describe the destination. The ActionScript generator gets all the information required to generate a DataService class and a wrapper around instances of it from the entity element and its associated filter elements.

Because a Data Management Service destination backs the DataService object on the client, the client only has to send getItem, createItem, updateItem, and deleteItem messages to the destination, just as it would in non-model Data Management Service scenarios.

In the following model example, for an implicit fill on the Book entity's publicationDate property, the generated service wrapper contains a getBooksByPublicationDate function. The implementation of this function calls a fill named BooksByPublicationDate on the associated DataService object. On the server, the assembler can handle this named fill because it is an implicit fill based on standard Data Management Service behavior and is handled by restricting the publicationDate field by the passed-in parameter. The Book entity is augmented with a few explicit filters.

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Book" persistent="true">
    <annotation name="ServerProperties">
      <item name="ServerType">LCDS</item>
    </annotation>
    <id name="id" type="integer"/>
    <property name="title" type="string"/>
    <property name="publicationDate" type="date"/>
    <property name="author" type="Author"/>
    <filter name="BooksByTitleOrdered" order="title asc">
      <annotation name="DMS">
        <item name="propertySpecifier">title</item>
      </annotation>
    </filter>
    <filter name="BooksByTitleAndAuthor">
      <annotation name="DMS">
        <item name="propertySpecifier">title,author</item>
      </annotation>
    </filter>
  </entity>

  <entity name="Author" persistent="true">
    <id name="id" type="integer"/>
    <property name="lastName" type="string"/>
    <property name="firstName" type="string"/>
  </entity>
</model>
```

The ActionScript generator does not create implicit services for all entities in a model. It only generates implicit services for entities with the ServerType annotation set to LCDS in the ServerProperties annotations groups. This annotation must be at the entity level because a model can have a mixture of entities backed by the LiveCycle Data Service Data Management Service and entities not backed by the Data Management Service.

The following example shows a generated service wrapper for the implicit BookService:
package
{
    import flash.events.Event;
    import flash.events.EventDispatcher;
    import mx.data.DataService;
    import mx.data.ItemReference;
    import mx.rpc.AsyncToken;
    import mx.data.ItemReference;
    public class _BookService extends DataServiceWrapper
    {
        public function _BookService()
        {
            // destination name determined by modeling naming convention
            _serviceControl = new DataService("FiberBookDestination");
        }
        // CREATE FUNCTION
        public function createItem(book:Book):ItemReference
        {
            var token:ItemReference = _serviceControl.createItem(book);
            if (_serviceControl.autoCommitCUD)
                _serviceControl.commit(book);
            return token;
        }
        // DELETE FUNCTION
        public function deleteItem(book:Book):AsyncToken
        {
            var token:AsyncToken = _serviceControl.deleteItem(book);
            if (_serviceControl.autoCommitCUD)
                _serviceControl.commit(book);
            return token;
        }
        // UPDATE FUNCTION
        public function updateItem(book:Book):void
        {
            if (_serviceControl.autoCommitCUD)
                _serviceControl.commit(book);
        }
        // GET ITEM
        public function getItem(id:String):ItemReference
        {
            var token:ItemReference = _serviceControl.getItem({"id":id});
            return token;
        }
        // FILL FUNCTIONS
        public function getBooksByTitleOrdered(title:String):AsyncToken
Implicit service for the Model Assembler

When working with the Model Assembler, the union of the sets of explicit and implicit filters in a model determines the set of supported fills. For each entity backed by the Data Management Service, a set of common implicit filters is supported. For example, there is an implicit filter for getAll. For more information, see “filter” on page 24.

Each implicit and explicit filter is translated to a supported named fill for which the Model Assembler handles the server-side execution. Each supported fill corresponds to a strongly typed function on the generated service wrapper. For example, the explicit filter BooksByTitleOrdered is represented in the generated service wrapper by the getBooksByTitleOrdered() function. The implementation of this function calls a fill named BooksByTitleOrdered on the associated client-side DataService object.

On the server, the Model Assembler can handle this named fill because the filter declaration in the model contains all the necessary information: Filter by the title property and order in ascending order.

Implicit service for non-model custom assembler

When the Model Assembler backs an entity, the union of explicit and implicit filters for that entity determines the set of fills that the destination supports. However, for an existing custom assembler destination (non-model), the developer (or the tool, in the case of Flash Builder) must specify the set of fills that the custom assembler supports. The generated ActionScript is based on an implicit service and fills defined in filter elements with no query body.

The following example shows a model generated in Flash Builder by introspecting two existing Data Management Service destinations. The entities in the model, Company and Employee, have a one-to-many relationship.
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <annotation name="ActionScriptGeneration">
    <item name="FullyQualifyReferences">true</item>
  </annotation>
  <entity name="Company" persistent="true">
    <annotation name="EntityProperties">
      <item name="EntityOriginalName">Company</item>
    </annotation>
    <annotation name="ServerProperties">
      <item name="ServerType">LCDS</item>
      <item name="Destination">crm-company</item>
      <item name="CustomDestinationFlag">true</item>
      <item name="RemoteClass">flex.samples.crm.company.Company</item>
    </annotation>
    <annotation name="service Group">
      <item name="service">CompanyService,EmployeeService</item>
    </annotation>
    <annotation name="FbEntityProperties">
      <item name="AssemblerClass">flex.samples.crm.company.CompanyAssembler</item>
      <item name="contextRoot">/lcds-samples</item>
      <item name="rootUrl">http://localhost:8400/lcds-samples</item>
    </annotation>
    <annotation name="ActionScriptGeneration">
      <item name="Package">valueObjects</item>
      <item name="ImplicitServicePackage">services.crmcompany</item>
    </annotation>
    <annotation name="ASFilesGenerated">
      <item name="ASFilesGenerated">valueObjects/_CompanyEntityMetadata.as~#~valueObjects/_Super_Company.as~#~valueObjects/Company.as~#~</item>
    </annotation>
    <filter name="fill" arguments="arg0:string,arg1:string"/>
    <id name="companyId" type="integer"/>
    <property name="industry" type="string" required="true"/>
    <property name="address" type="string" required="true"/>
    <property name="state" type="string" required="true"/>
    <property name="zip" type="string" required="true"/>
    <property name="name" type="string" required="true"/>
    <property name="city" type="string" required="true"/>
  </entity>
  <entity name="Employee" persistent="true">
    <annotation name="EntityProperties">
      <item name="EntityOriginalName">Employee</item>
    </annotation>
    <annotation name="ServerProperties">
      <item name="ServerType">LCDS</item>
      <item name="Destination">crm-employee</item>
      <item name="CustomDestinationFlag">true</item>
      <item name="RemoteClass">flex.samples.crm.employee.Employee</item>
    </annotation>
    <annotation name="service Group">
      <item name="service">EmployeeService</item>
    </annotation>
    <annotation name="FbEntityProperties">
      <item name="AssemblerClass">flex.samples.crm.employee.EmployeeAssembler</item>
      <item name="contextRoot">/lcds-samples</item>
      <item name="rootUrl">http://localhost:8400/lcds-samples</item>
    </annotation>
  </entity>
</model>
Event registration for generated service wrappers

The current set of RPC services allows consumers to register ActionScript function listeners for result and fault events. Service wrappers support event registration for events of their services. For example, you could register an event handler for a ResultEvent directly on the generated RPC service wrapper. The wrapper would propagate the event to this handler when the service dispatches the event.

You can specify an event handler for a service wrapper in MXML, as the following example shows:

```xml
<local:BookService id="bookService" result="showBook(event)" service="bookServ" port="4025"/>
```

You can specify an event handler for a service wrapper in ActionScript, as the following example shows:

```actionscript
bookService.getDataManager(BookService.DATA_MANAGER_BOOK).addEventListener(DataServiceResultEvent.RESULT, showBook);
```

Using unsupported data types

The application modeling language is a strongly typed modeling language and has no support for Object, Map, or loosely typed variables. To represent such unsupported types, you use the opaque type. For client-side data management, the opaque type provides a way to pass data through.

For standard data management, the signature of the `getItem` method on a server-side destination is `Object getItem(Map identity)`. With client-side data management, it is not unlikely that users have `getItem` endpoints that take a single map parameter containing the name-value pairs for the desired item’s identity properties. Since application modeling technology does not support a Map data type, for a `getItem` function that takes a Map argument where the map contains `lastName:String` and `firstName:String` (the id properties of an entity), a service function can use arguments typed as opaque, as the following example shows:
Model-driven Form generation

Model-driven Forms are MXML component that contain aFormItem for each property in an entity. You can generate and use model-driven Forms when using the Model Assembler feature in LiveCycle Data Services. You generate Forms in Flash Builder by selecting a data type (generated from an entity) in the Flash Builder Data/Services panel and choosing the relevant menu and dialog options. For more information on generating model-driven Forms, see “Building model-driven applications” in Using LiveCycle Data Services.

The following example shows a model that contains a Product entity with a style for performing price validation:
The following example shows the model-driven Form generated from the Product entity in Flash Builder. The validator associated with the priceValidation style is initialized near the bottom of the code sample.
<?xml version="1.0" encoding="utf-8"?>
<Form xmlns="http://www.adobe.com/2006/mxml" width="400" height="300"
creationComplete="_creationCompleteHandler()">
  <FormItem label="Description" includeInLayout="{valueObject == null || 
    valueObject._model.isDescriptionAvailable}"
    visible="{valueObject == null || valueObject._model.isDescriptionAvailable}"
    enabled="{valueObject != null}" />
  <TextInput id="_descriptionInput" text="{valueObject.description}" 
    enabled="{valueObject != null}" />
  <Binding source="_descriptionInput.text" destination="valueObject.description"/>

  <FormItem label="Price" includeInLayout="{valueObject == null || 
    valueObject._model.isPriceAvailable}"
    visible="{valueObject == null || valueObject._model.isPriceAvailable}"
    enabled="{valueObject != null}" />
  <NumericStepper id="_priceInput" minimum="-1000000" stepSize="0.1" maximum="10000000"
    color="{valueObject._model.priceIsValid ? 0x000000 : 0xFF0000}" value="{valueObject.price}" 
    enabled="{valueObject != null}" />
  <Binding source="_priceInput.value" destination="valueObject.price"/>

  <FormItem label="Productname" includeInLayout="{valueObject == null || 
    valueObject._model.isProductnameAvailable}"
    visible="{valueObject == null || valueObject._model.isProductnameAvailable}"
    enabled="{valueObject != null}" />
  <TextInput id="_productnameInput" text="{valueObject.productname}" 
    enabled="{valueObject != null}" />
  <Binding source="_productnameInput.text" destination="valueObject.productname"/>

  <Spacer height="10"/>
  <HBox width="100%" horizontalAlign="center">
    <Button id="_btnAdd" label="Add" click="_btnHandler(event)"
      enabled="{!_serviceWrapper.serviceControl.commitRequired}"
      icon="@Embed(source='icons/LC_Form_Add.png')"
      disabledIcon="@Embed(source='icons/LC_Form_Add_Disabled.png')" />
    <Button id="_btnDelete" label="Delete" click="_btnHandler(event)"
      enabled="{valueObject != null && 
        !_serviceWrapper.serviceControl.commitRequired}"
      icon="@Embed(source='icons/LC_Form_Delete.png')"
      disabledIcon="@Embed(source='icons/LC_Form_Delete_Disabled.png')" />
    <Button id="_btnReset" label="Reset" click="_btnHandler(event)"
      enabled="{_serviceWrapper.serviceControl.commitRequired}"
      icon="@Embed(source='icons/LC_Form_Reset.gif')"
      disabledIcon="@Embed(source='icons/LC_Form_Reset_Disabled.gif')" />
    <Button id="_btnSave" label="Save" click="_btnHandler(event)"
      rollOver="_calculateErrorString()"
      enabled="{!_serviceWrapper.serviceControl.commitRequired && 
        !valueObject._model.isValid}"
      borderColor="{valueObject == null || valueObject._model.isValid? 0xaab3b3 : 
        0xff0000}" 
      toolTip="{!valueObject._model.isValid? 'Save is not allowed due to:
        ' + _errorString : ''}" 
      icon="@Embed(source='icons/LC_Form_Save.png')"
      disabledIcon="@Embed(source='icons/LC_Form_Save_Disabled.png')" />
  </HBox>
</Script>
<!CDATA[
  import mx.events.CloseEvent;
  import mx.rpc.AsyncResponder;
  import mx.rpc.AsyncToken;
  import mx.rpc.events.ResultEvent;
]}
import mx.rpc.events.FaultEvent;
import mx.controls.Alert;
// Imports for the entity and its service.
import ModelFormApp.Product;
import ModelFormApp.ProductService;
// Imports for associated entities and their services.
[Bindable]
private var _serviceWrapper:ProductService;
private var _valueObject:Product;
private var _prevValueObject:Product;
[Bindable]
private var _errorString:String;
[Bindable]
public function get valueObject():Product
{
    return _valueObject;
}
public function set valueObject(value:Product):void
{
    _valueObject = value;
    _initValidations();
}
// Button handler.
private function _btnHandler(event:Event):void
{
    if (event.target == _btnAdd)
    {
        // Hold onto the current value object, in case add is reverted later.
        _prevValueObject = valueObject;
        valueObject = new Product();
        _serviceWrapper.createProduct(valueObject);
    }
    else if (event.target == _btnDelete)
    {
        Alert.show("Are you sure you want to delete the item?", "Delete Item", 3, this,
        function alertClickHandler(event:CloseEvent):void
        {
            if (event.detail==Alert.YES)
            {
                var deleteToken:AsyncToken = _serviceWrapper.deleteProduct(valueObject);
                deleteToken.addEventListener("response", new AsyncResponder(
                    function (event:ResultEvent, token:Object=null):void
                    {
                        valueObject = null;
                    },
                    function (event:FaultEvent, token:Object=null):void
                    {
                        Alert.show("Delete failed: " + event.fault.faultString, "Error");
                        _serviceWrapper.serviceControl.revertChanges(valueObject);
                    }));
                _serviceWrapper.serviceControl.commit();
            }
        });
    }
    else if (event.target == _btnSave)
    {
        // Button handler.
    }
}
_serviceWrapper.serviceControl.commit(); // Commit the change.
    if (_prevValueObject != null)
      _prevValueObject = null;
  }
  else if (event.target == _btnReset)
  {
    _serviceWrapper.serviceControl.revertChanges(valueObject);
    if (_prevValueObject != null)
    {
      valueObject = _prevValueObject;
      _prevValueObject = null;
    }
    else
    {
      _initValidations();
    }
  }

  // Creation complete handler.
  private function _creationCompleteHandler():void
  {
    // Initialize service wrapper and make sure autoCommit is set to false.
    _serviceWrapper = new ProductService();
    _serviceWrapper.serviceControl.autoCommit = false;
    // Initialize associated services.
  }

  // Calculates the String for validation and constraint errors.
  private function _calculateErrorString():void
  {
    if (valueObject != null)
    {
      var errors:String = "";
      for (var i:int=0; i < valueObject._model.validationFailureMessages.length; i++)
        errors += "-" + valueObject._model.validationFailureMessages[i] + "\n";
      _errorString = errors;
    }
    else
    {
      _errorString = "";
    }
  }

  // Initializes style validators (if any) to the appropriate fields.
  private function _initValidations():void
  {
    if (valueObject == null)
      return;
    valueObject._model.priceStyle.validator.source = _priceInput;
    valueObject._model.priceStyle.validator.property = "value";
  }

  }}>
</script>
</form>

**Important:** The model-driven Form is a developer productivity aid only. It is not a general solution for RIA form generation and development. The model-driven Form is only intended for use with models deployed to the LiveCycle Data Service server using the Model Assembler feature. Regenerating a model-driven Form overwrites the existing Form.
Value object binding
A generated model-driven Form component has a single publicly settable property named `valueObject`. The type of the `valueObject` property is the value object class generated for the entity corresponding to the Form. When the Form component is used in a Flex user interface, this property must be set to enable the Form for action.

Data types
The following table lists the mapping between entity data types and the Flex FormItem components used to represent them in generated model-driven Forms:

<table>
<thead>
<tr>
<th>Data type</th>
<th>FormItem component</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>mx.controls.TextInput</td>
</tr>
<tr>
<td>integer, float, double, long</td>
<td>mx.controls.NumericStepper</td>
</tr>
<tr>
<td>date</td>
<td>mx.controls.DateField</td>
</tr>
<tr>
<td>boolean</td>
<td>mx.controls.CheckBox</td>
</tr>
<tr>
<td>entity (associations)</td>
<td>mx.controls.ComboBox</td>
</tr>
</tbody>
</table>

Single value associations (*-to-one) are represented by a ComboBox component that contains a list of all instances of the associated entity type determined by a `getAll()` fill invocation to the destination corresponding to that type. The ActionScriptGeneration.DisplayColumn property-level annotation lets you specify the property of the associated entity to be displayed in association ComboBoxes. The property you specify is used whenever a list of instances of the containing entity is displayed in a ComboBox.

Collection properties, whether collections of primitives, or multi-value associations (*-to-many), are ignored and not displayed by the model-driven Form.

Data properties
As soon as a value object is bound to a model-driven Form, FormItems corresponding to data properties are enabled and editable. The form generator uses the two-way binding feature of the Flex SDK to create two-way binding between each FormItem and the corresponding property of the value object associated with the Form. Whenever the value in a FormItem changes, the value object’s property is updated and vice versa. When a model-driven Form displays existing entity instances, all data property FormItems are initialized to the current values of the instance being edited.

Derived properties
FormItems corresponding to derived properties are enabled but not editable. A single-direction binding is created between the value object’s corresponding property and the corresponding FormItem.

Styles
Model-driven Forms use the caption messages of style elements for labels of the FormItems. Full style message capabilities are supported. Model-driven Forms use the description messages of style elements for the toolTip (hover text) of the FormItems. Full style message capabilities are supported. Style validations contribute to the validity state of the value object bound to the Form and are integrated into the Flex validation framework.

Constraints and validations
FormItems are not created for constraints. Constraints and style validations are help define the overall validity state of a value object instance and are not intended for individual display.
The validity state of a value object is indicated in a model-driven Form in the following ways:

- If a property contains style validations, it uses the Flex validation framework. When the property is not valid, its FormItem is outlined in red and has a red hover box that displays the validation error message. When the property's validations evaluate to false, the property is displayed in red instead of black.
- The Save button on the Form is disabled when the value object instance is not valid.
- The set of all validation failure messages, constraints and style validations, is displayed as the hover text for a disabled Save button.

Constraint failures are detectable only in the Save button's hover text and not at the level of the property's Flex validation.

Non-FormItem code
The model-driven Form component instantiates a service wrapper around the LiveCycle Data Services service that backs its corresponding entity. This service wrapper instance allows the Form component to persist and revert changes that made with it. In addition to the FormItems described above, the Form contain the following buttons:

- ADD invokes the createItem operation on the destination corresponding to the Form's entity, and binds the newly created value object instance to the Form. This button is only enabled when commit is not required on the corresponding DataService instance. This safeguard prevents the Form from losing a handle on a value object while the user is making data modifications on the client.
- DELETE invokes delete operation on the destination corresponding the Form's entity for the value object instance currently bound to the Form. This button is only enabled when commit is not required on the corresponding DataService instance.
- RESET invokes the revertChanges operation on the DataService corresponding to the Form's entity. This button is only enabled when commit is required on the corresponding DataService instance.
- SAVE invokes commit on the DataService instance corresponding to the Form's entity. This button is only enabled when the corresponding value object is valid and commit is required (changes have been made).

Customized Forms
The model-driven Form generator uses a default FreeMarker template called ModelFormTemplate.ftl. The template is usually hidden. You can use your own customized version of the template file instead of the default template. For more information on customizing a model-driven Form, see “Building model-driven applications” in Using LiveCycle Data Services.