Guidewire training materials contain Guidewire proprietary information that is subject to confidentiality and non-disclosure agreements. You agree to use the information in this manual solely for the purpose of training to implement Guidewire software solutions. You also agree not to disclose the information in this manual to third parties or copy this manual without prior written consent from Guidewire. Guidewire training may be given only by Guidewire employees or certified Guidewire partners under the appropriate agreement with Guidewire.

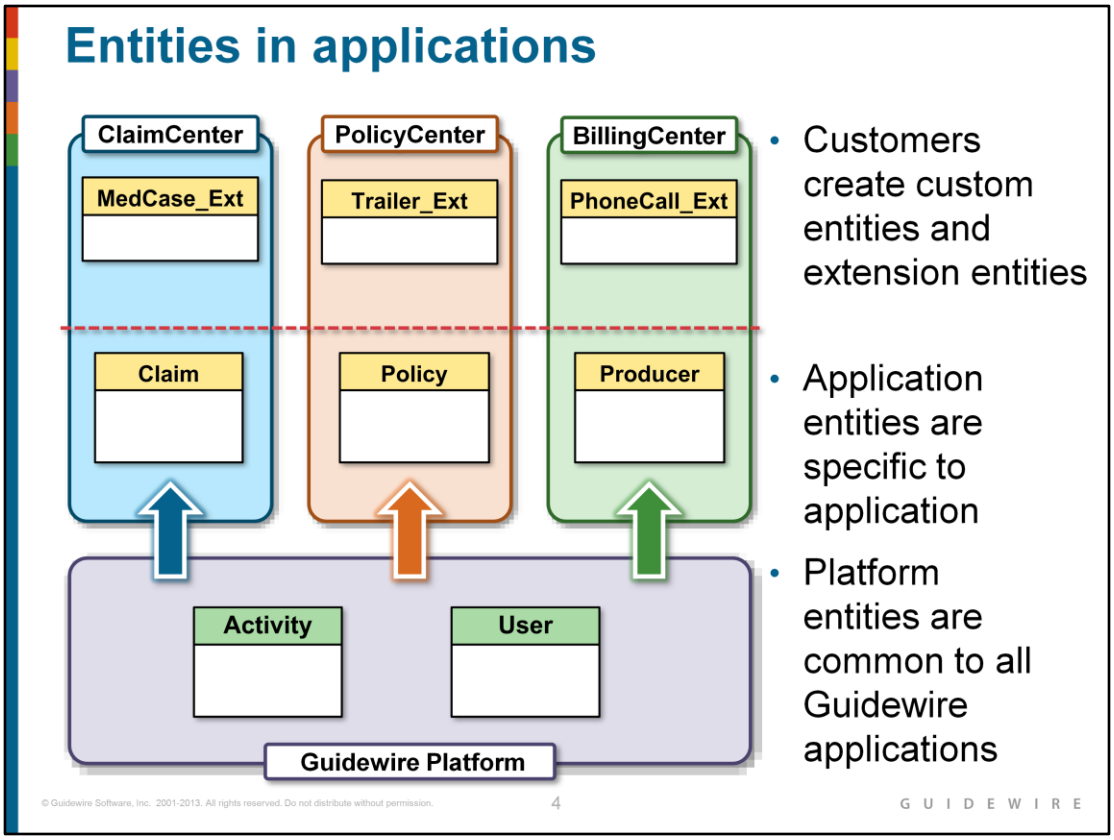
Lesson objectives

- By the end of this lesson, you should be able to:
  - Describe what is a custom entity and how to create a custom entity
  - Create entity elements and subelements that define fields, foreign keys, and arrays on entities
  - Identify entity related data elements associated with data model design

This lesson uses the notes section for additional explanation and information.
To view the notes in PowerPoint, select View → Normal or View → Notes Page.
When printing notes, select Note Pages and Print hidden slides.

Lesson outline

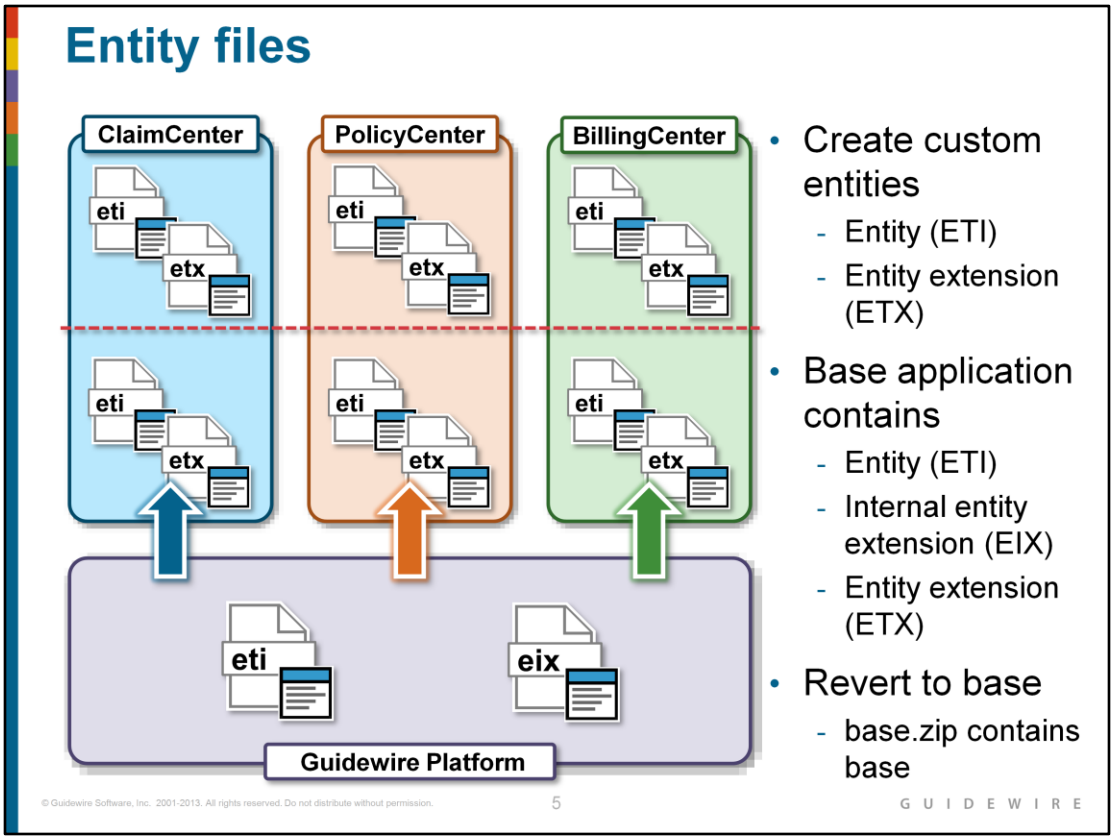
- Custom entities
- Creating and defining an entity
- Related data model elements



The example above has three theoretical custom entities that could be added to each application, including:

- A ClaimCenter MedCase_Ext entity, which stores custom medical information about an injury for which there is a legal dispute.
- A PolicyCenter Trailer_Ext entity, which stores information about trailers that can be covered on Personal Auto policies.
- A BillingCenter PhoneCall_Ext entity, which stores information about phone conversations that occur with payers on delinquent accounts.

Keep in mind that all three products have robust base applications that meet the needs that are common to most carriers. Custom entities are typically created to meet business needs that are not common to the majority of insurance carriers.



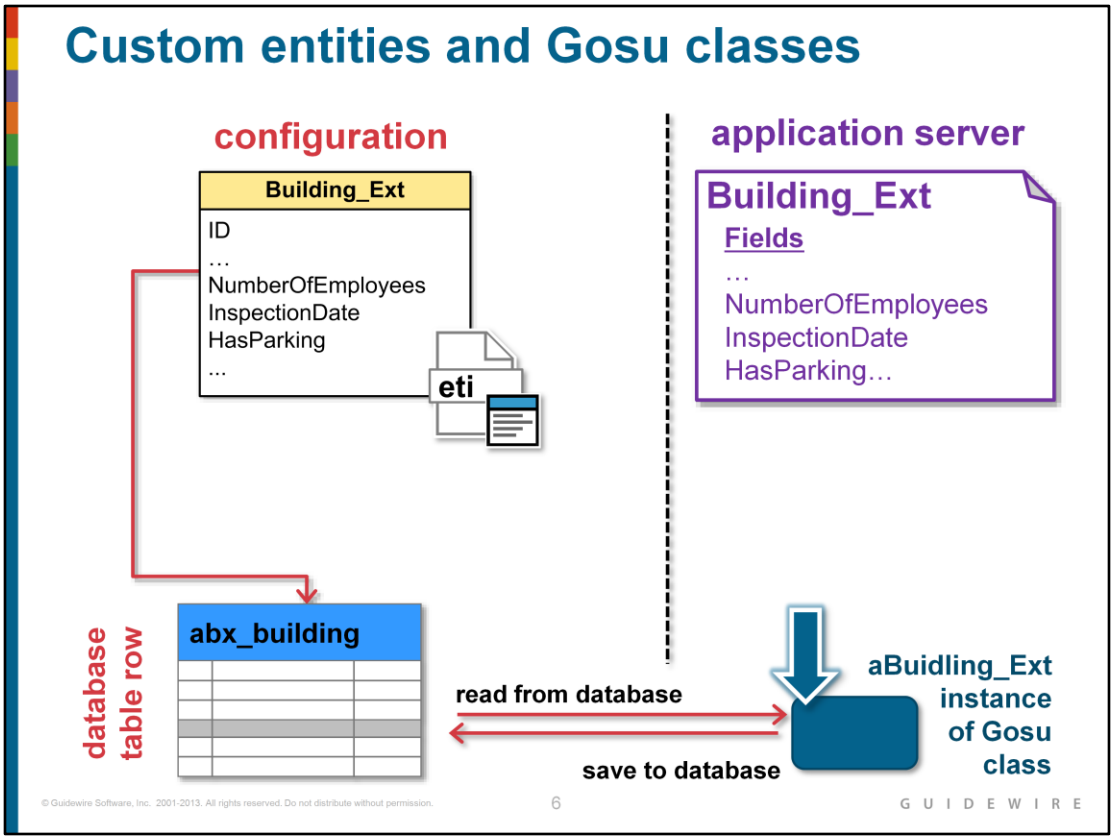
Platform entities are always found in the …\modules\configuration\config\metadata\entity\ folder. All ETI and EIX files in the \metadata\entity\ are read-only. You cannot edit these files directly in Guidewire Studio and should not edit these files in any other application.

Many platform entities have the platform="true" attribute defined in the <entity /> element. Ignore the deprecated base="true" or base="false" attribute in the <entity /> element.

Some application specific entities can also be found in the \metadata\entity\ folder. Many application entities are extendable in the sense that they can be both subtyped and/or an entity extension can be created. Entities with the final="true" attribute defined in the <entity /> element cannot be subtyped. Entities with the extendable="false" attribute defined in the <entity /> element cannot be extended in the sense that new elements cannot be added to the entity.

You can always determine to base configuration for a Guidewire application by looking at the <installDirectory>\modules\base.zip file.

This compressed zip file contains the base configuration files for both platform and the application. The base.zip file is not exposed in the Guidewire Studio project directly. However, for many base application files in Guidewire Studio, you can revert the file back to base file contained in base.zip. Rule folders and files cannot be reverted in Studio back to base.



For a customer data model entity, the ETI file defines the fields. Custom entities are a single ETI file that customers can edit as needed. All of the fields in the custom entity's ETI file become a part of the internal Gosu class and a database table.

For custom entities, Guidewire recommends that the entity name should end with _Ext (or start with Ext), for example, Building_Ext as the name of the entity file. The names of the fields in the custom entity do not need to start or end with any prefix. Only use Ext in field names when extending an existing base application entity or creating new elements in an entity extension of a base application entity.

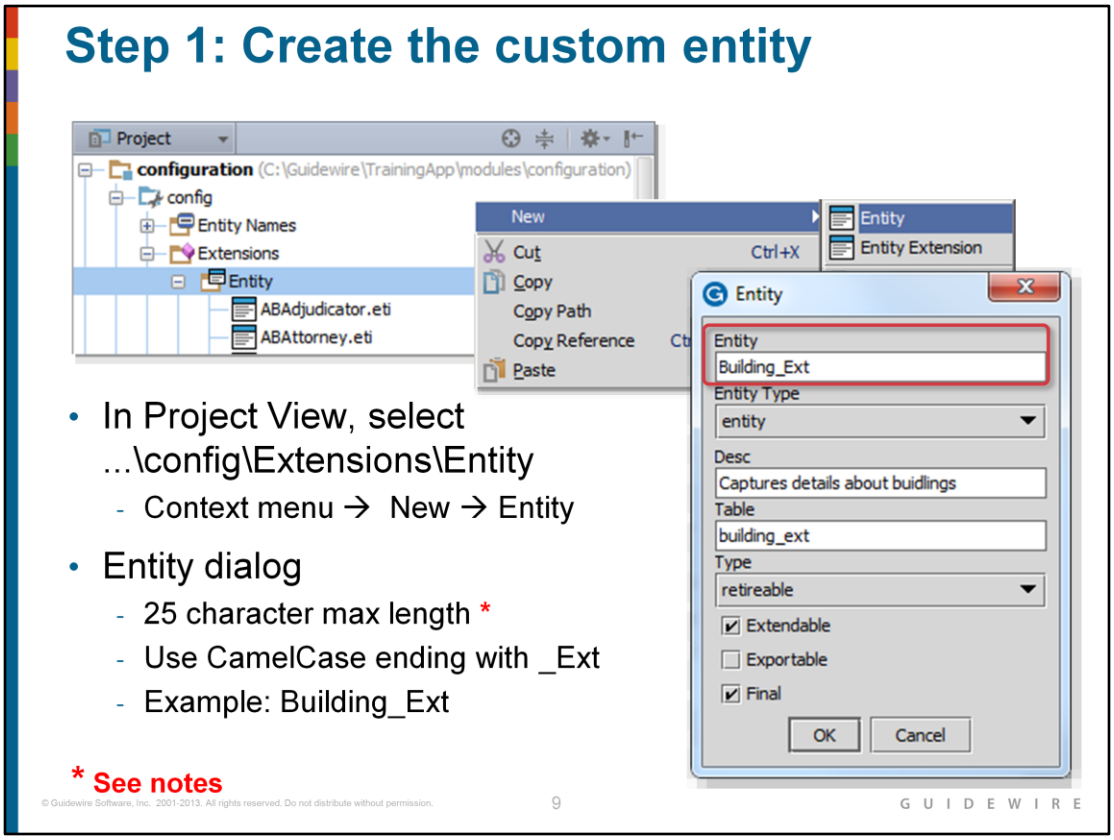
Students coming from an Object-Oriented Programming (OOP) background should be aware that the term "extend" gets used in OOP differently than it does in Guidewire. In OOP, the term "extend" is used to refer to creating a new subclass that extends some parent superclass. In Guidewire data model configuration, the term "extend" is sometimes used to refer to adding new fields to an existing base application entity. In this sense, no new subclass or subtype entity is getting created. One is simply adding additional fields to an existing base application entity.

Lesson outline

- Custom entities
- Creating and defining an entity
- Related data model elements

Steps to create a custom entity

1. Create the entity file
2. Add elements (and subelements) and specify attribute values
3. Optionally regenerate the dictionary
4. Deploy the custom entity



Guidewire recommends the entity name should be in CamelCase. CamelCase is a style of capitalization in which multiple words are joined together with the first character of each word capitalized. Examples include ClaimCenter and JavaScript. CamelCase is the naming convention recommendation for entities. Guidewire recommends ending the name of an Entity with _Ext to distinguish the entity from base application entities.

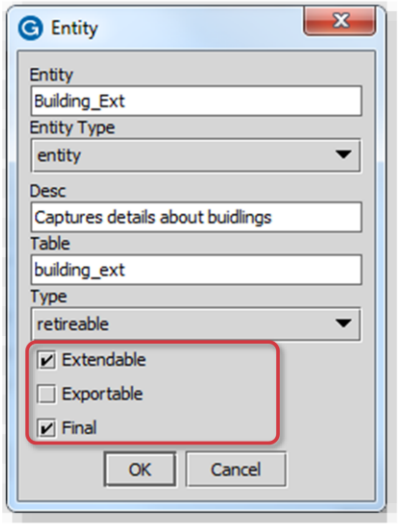
In the entity dialog, you can also define the table name. In the majority of cases, you can keep the dialog defaults. However, by convention we remove the _Et from table name since naming collision is avoided by the automatic prefixing <app>x as in the example: abx_building.

For loadable tables (ones that can be loaded through staging tables), the table name must be 25 characters or less. For non-loadable tables (ones that cannot be loaded through staging tables), the table name must be 26 characters or less. Some entities are supertypes to subtypes. Subtype entity names have a limit of 22 characters.

A retireable entity is an extension of the editable entity type, and is the most common type of entity. Most, but not all, base entities are of this type. Retireable means that entities of this type are never deleted. Entities of the type versionable can be deleted. Editable is not a recommended type. Instead, use versionable. Versionable entities have an a version and ID field. At application server start up, the Guidewire application loads all XML definitions of the data entities into the application database.

Optional <entity /> attributes

- Extendable
  - Internal usage
  - Not needed for custom entities
- Exportable
  - Deprecated support
  - Default = false
  - True allows for deprecated support to serialize entity for SOAP API and RPC-E web services
- Final
  - Default = true
  - True means entity cannot be supertype of a subtype entity



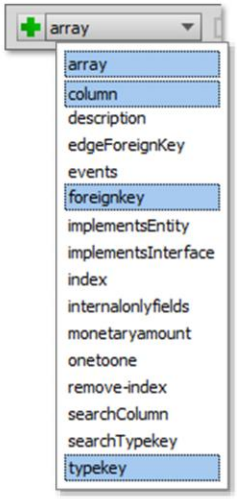
Extendable allows an entity to be extended and is for internal entities. If true, it is possible to extend this entity, which is something that is unnecessary to do for custom entities. Simply edit the custom entity.

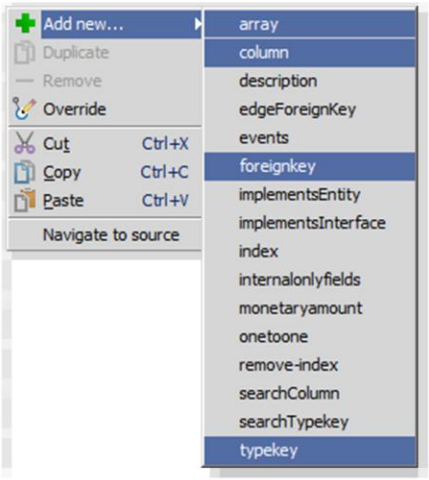
If an entity is exportable, then all of its arrays must be exportable. All of the entity's arrays must point to other exportable entities. The application will not start if there is an exportable entity with one or more arrays pointing to non-exportable entities. Exportable entities support entity serialization which is required for deprecated RPC-E web services and the SOAP API.

The final attribute defaults to true. If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype.

Step 2: Add elements and specify attributes

- Toolbar
  - Select option in dropdown
  - Click **+** to add more of same
- Context menu
  - Add new…
  - Select option in menu





GUIDEWIRE

When creating elements, the Entity Editor refers to an Extensible Schema Definition (XSD) file, datamodel.xsd. The location of the file is <InstallRoot>\modules\configuration\xsd\metadata\datamodel.xsd. Not all elements define fields in a database, for example, <events />. Other elements are specific to database performance such as <index />.

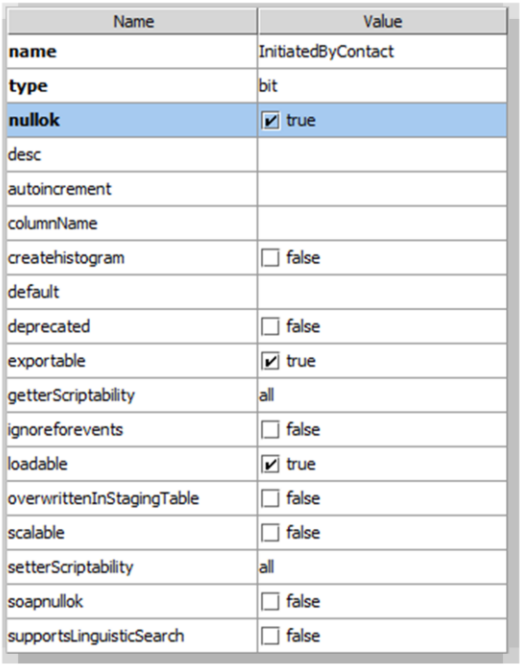
The four primary elements listed above are common to entity extension (<extension>), entity declaration (<entity>), subtype extension (<extension>), and subtype declaration (<subtype>).

Common elements to add

- Common elements (fields) for:
  - Entity extension, entity, subtype, subtype extension
- <array …/>
  - Define array entity and field
- <column …/>
  - Define data field with defined data type
  - Bit, datetime, integer, varchar
- <foreignkey …/>
  - Define foreign key field and entity
- <typekey …/>
  - Define typekey and related typelist
- Do **NOT** create custom entity field names ending with _Ext

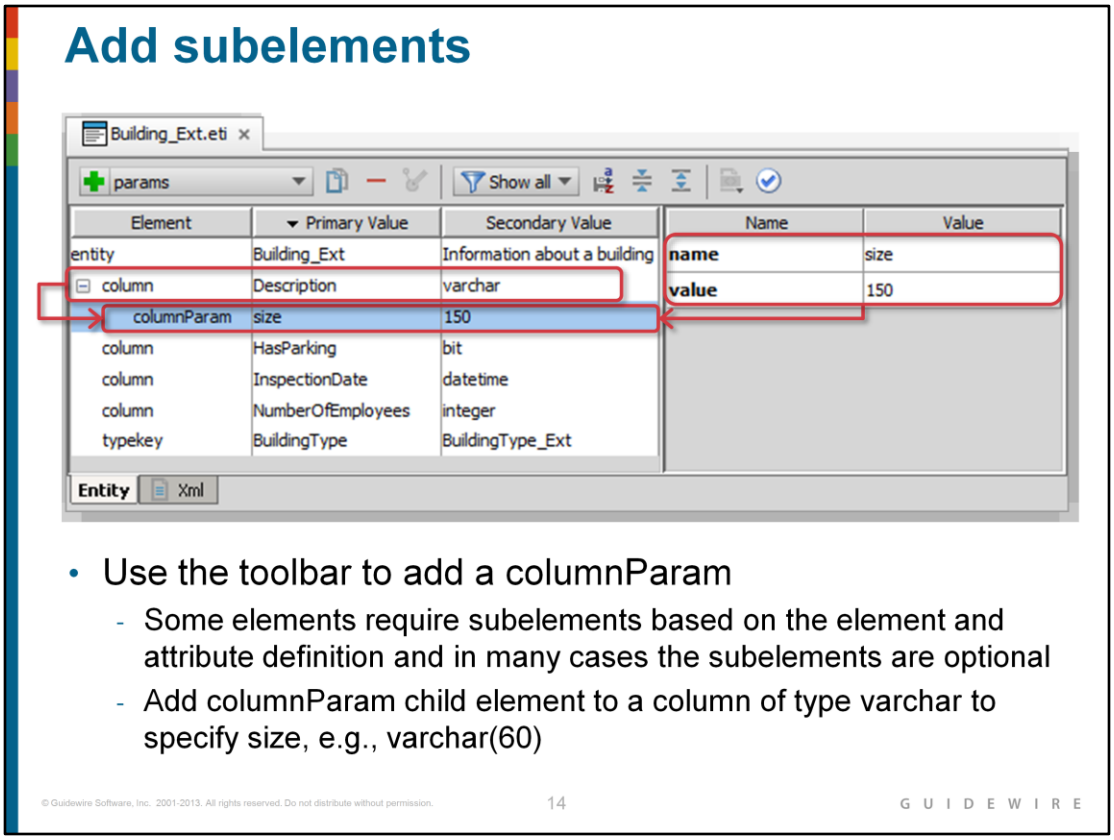
The four primary elements listed above are common to entity extension (<extension>), entity declaration (<entity>), subtype extension (<extension>), and subtype declaration (<subtype>).

Entity editor: Attribute pane



Name	Value
name	InitiatedByContact
type	bit
nullok	☑ true
desc	
autoincrement	
columnName	
createhistogram	☐ false
default	
deprecated	☐ false
exportable	☑ true
getterScriptability	all
ignoreforevents	☐ false
loadable	☑ true
overwrittenInStagingTable	☐ false
scalable	☐ false
setterScriptability	all
soapnullok	☐ false
supportsLinguisticSearch	☐ false

- Name Value columns
  - For selected element, define attributes
- Schema aware values
  - Boolean controls
  - Dropdown lists
- Attribute styling
  - Bold for required
  - Black for editable
  - Grayed-out for non-editable
    - Overridden, Inherited, Internal, Default
- Nullok defaults to false!
  - Set to true in most cases



You use the <columnParam> element to set parameters that a column type requires. The type attribute of a column determines which parameters you can set or modify by using the <columnParam> subelement.

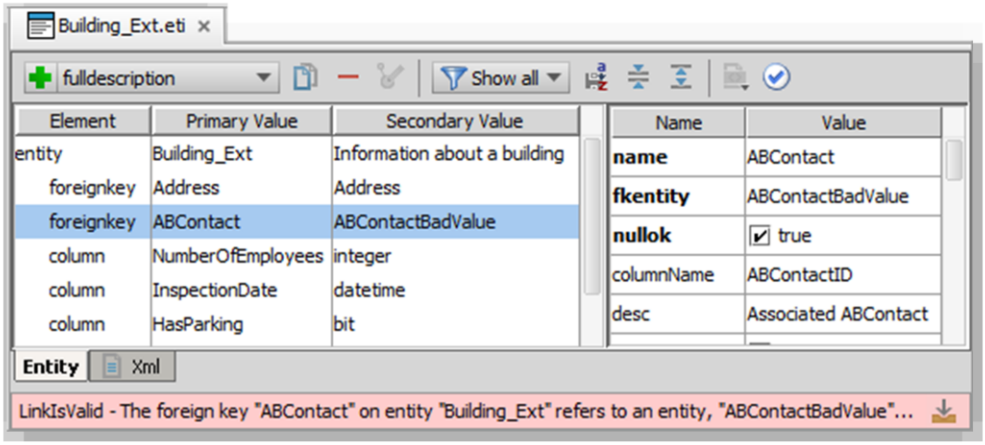
You can determine the list of parameters that a column type supports by looking up the type definition in its .dti file. You can find the datatypes dti files in the datatypes folder in …\modules\configuration\config\datatypes\. For example, the varchar.dti files defines the possible parameters for the varchar column type.

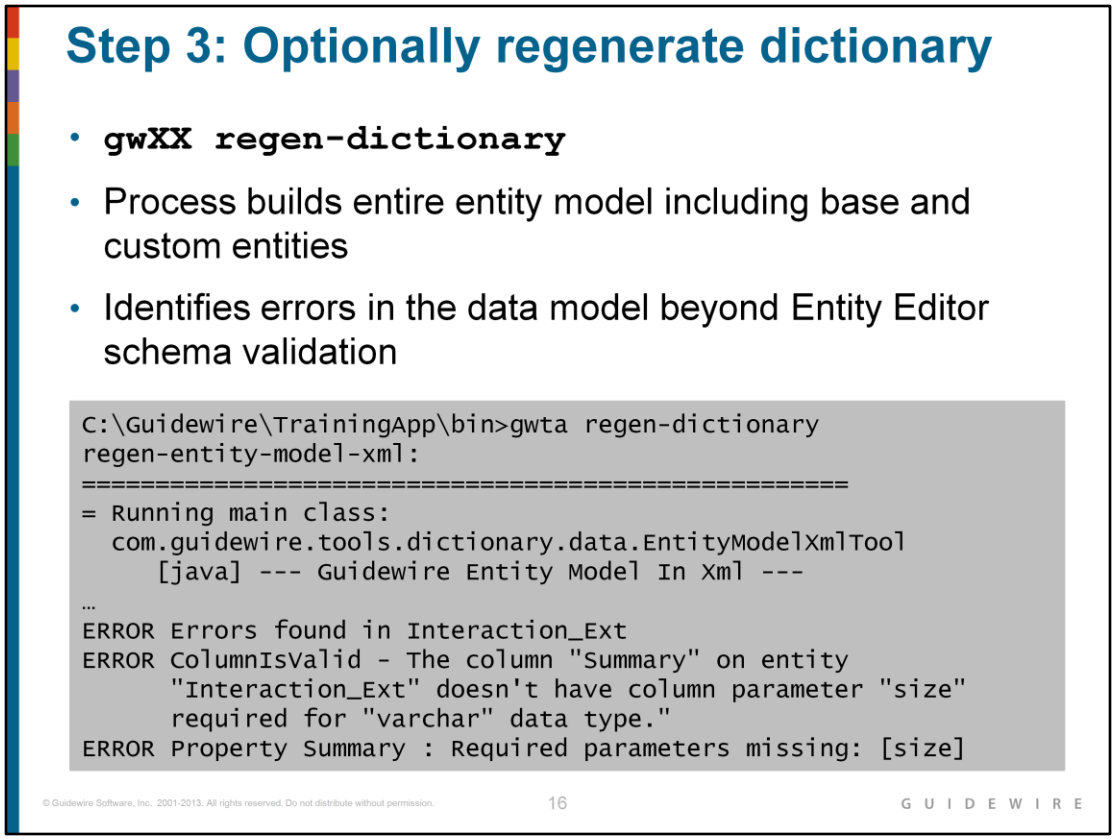
The Entity Editor is schema aware and knows displays the available options for you when you select the type.

Validate the schema

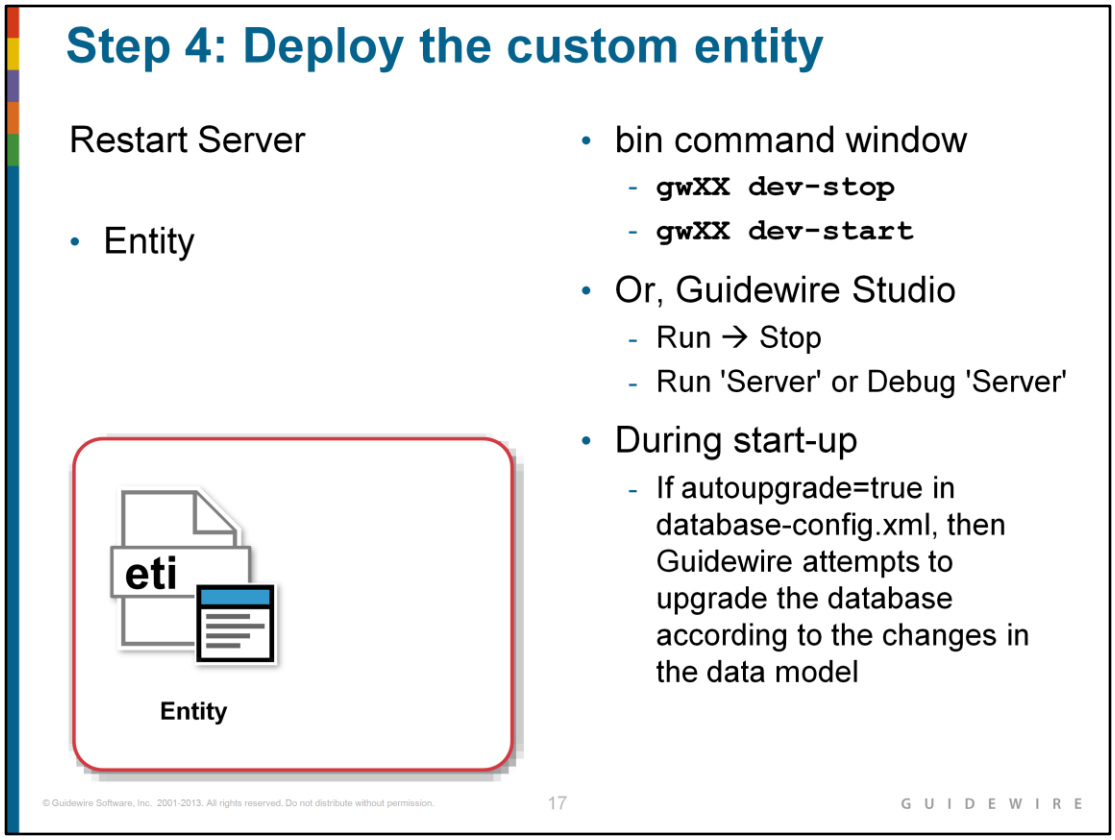
Click Validate in the toolbar

- Red highlight indicates schema violation warning
- Schema violations explained in pane below editor





You can also optionally regenerate the data dictionary to add the new entity to the data dictionary and to check for problems in the data model. Regenerating the data dictionary is not required, but doing so can identify flawed XML in the data model that go beyond schema validation such as certain types of referential integrity.



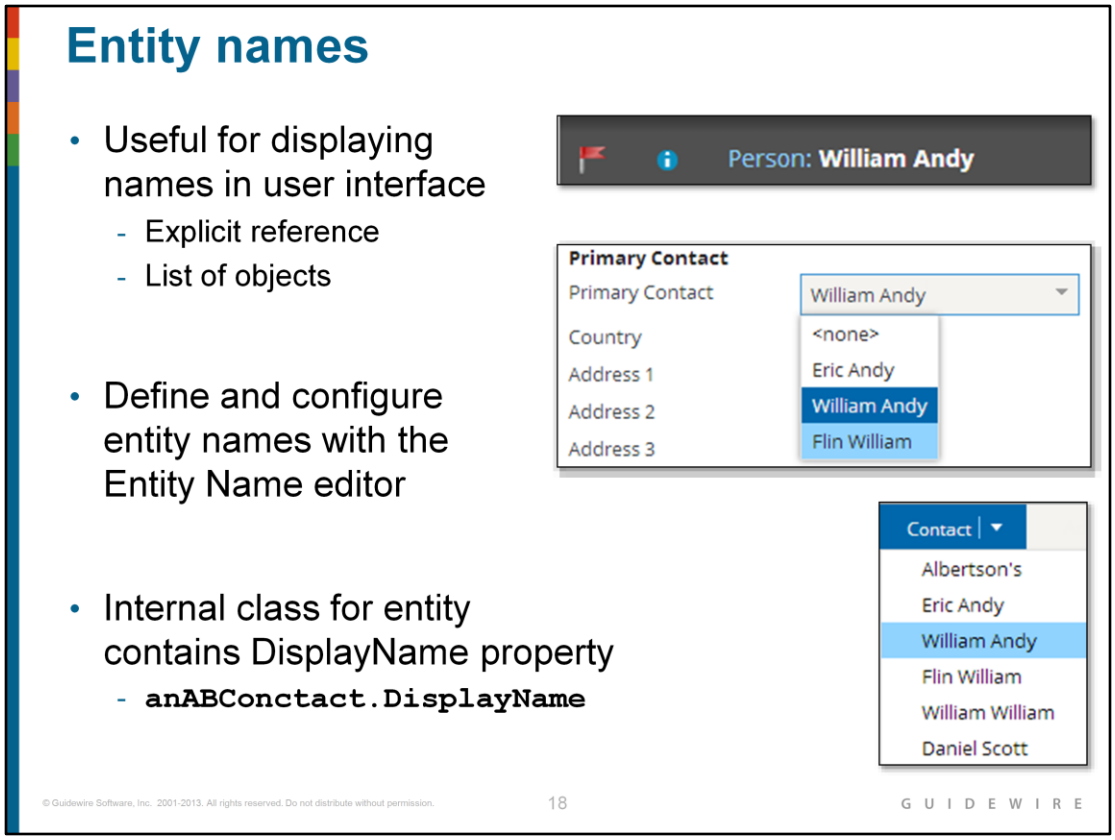
In Studio, the newly created entity is available to reference immediately. However, the entity is not available to the application server until the server is restarted and the database upgraded.

The automatic database upgrade process occurs only if the database autoupgrade=true attribute is defined in the database-config.xml.

TrainingApp uses an H2 database. Here is the XML definition found in database-config.xml:

```
<database name="TrainingAppDatabase"  dbtype="h2" autoupgrade="true">
   <dbcp-connection-pool
     jdbc-url="jdbc:h2:file:/Guidewire/TrainingApp/db/ta;auto_server=true"/>
</database>
```

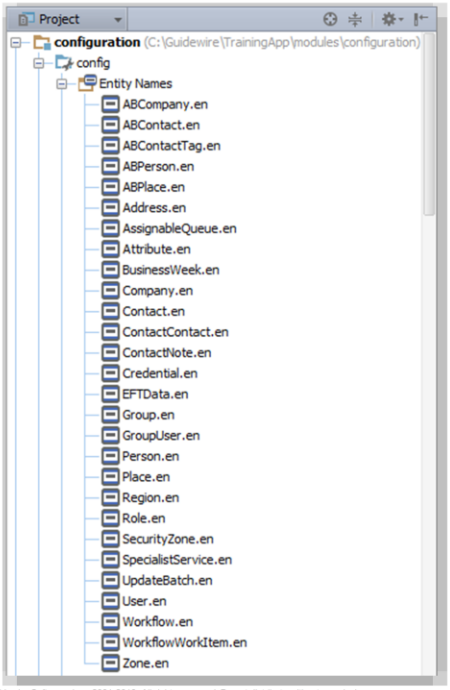
For SQL Server and Oracle, this option is set to false by default.



Entity names are used when…
A) an object as a whole is displayed for example in dropdown that lists ABContacts.
B) an object's DisplayName is explicitly referenced for example, in an info bar widget with its value property set to ABContact.DisplayName.

The DisplayName field does not appear in the Data Dictionary. It is available on every entity, however.

Entity Names



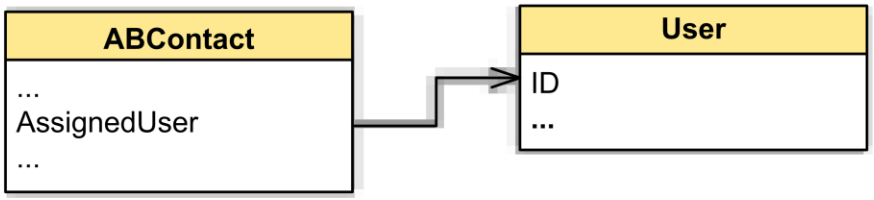
- **`\configuration\ config\Entity Names`**
  - Physical folder is displaynames
- Defined with Gosu code
- Entity names exist for many base and application entities
- For new entities displayed in the application user interface (UI), always create a default entity name
- Edit with Entity Name Editor

Lesson outline

- Custom entities
- Creating and defining an entity
- Related data model elements

Foreign key

- Pointer to single instance of some other entity
- Maintained by foreign key column in database table
- Example: each Contact can have one assigned user



- Recommended naming convention
  - Include columnName attribute and set it to field name + ID
  - For extension, name field with suffix _Ext and column name _ExtID

Name	Value
name	AssignedUser
fkentity	User
nullok	☑ true
columnName	AssignedUserID
desc	Assigned user

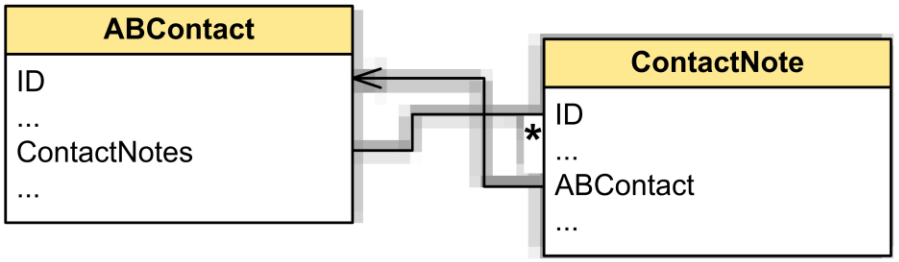
The columnName attribute is an optional attribute that specifies the actual name of the column in table created for this entity. This is useful when you want the database column name to differ from the field as referenced by Guidewire.

If a foreign key field is added to a base application entity, then the suffix should be _ExtID.

For a complete reference of the <foreignkey> syntax, refer to the application's Configuration Guide.

Array

- Collection of pointers to instances of some other entity that is maintained by code during runtime
- Requires reverse foreign key
  - If entity A has array of entity B, then entity B must have foreign key pointing to entity A
- Example: each ABContact can have zero to many buildings



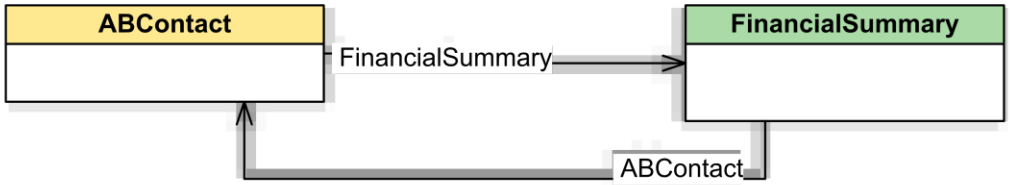
- Recommended naming convention
  - Using plural form of name; For extension, name field with suffix _Ext

Keep in mind that arrays are maintained in code. The code assembles the array by executing queries against the database. In order to do this, the application must be able to query for all members of a given array. It can do this only if each member has a foreign key referencing its parent.

Note that the value of the arrayentity attribute must match exactly the name of the entity it references.

One-to-one relationships

- <onetoone> element defines a single-valued association to another entity that has a one-to-one cardinality
- Provides a reverse pointer to an entity that is pointing at the <onetoone> entity through a foreign key



- Often splits logical entity across multiple physical entities
- Example:
  - ABContact defines Financial Summary as one-to-one
  - FinancialSummary defines a foreign key to ABContact

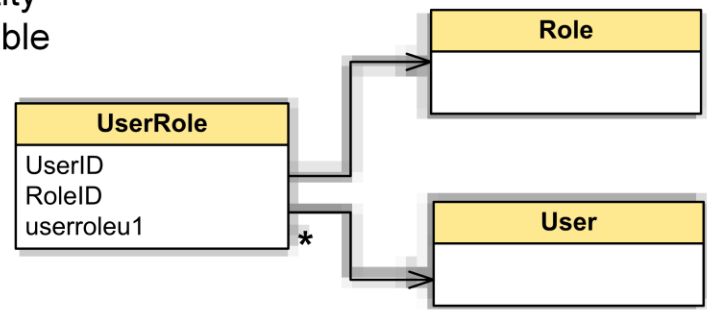
A one-to-one relationship often splits a logical entity across multiple physical entities. For example, when an ABContact is a ABPolicyPerson or ABPolicyCompany, the data model must capture financial summary information.

Storing the financial summary details into the ABContact table would result in a table with numerous rows with null columns. Instead, it is better to have a separate entity manage the financial summary details. In this manner, every ABContact has (at most) one FinancialSummary, and every financial summary applies to (at most) one ABContact.

For more information on one-to-one relationships, refer to documentation.

Many-to-many relationships

- Requires an entity of type versionable
  - Two or more foreign keys
  - Unique index for both foreign keys

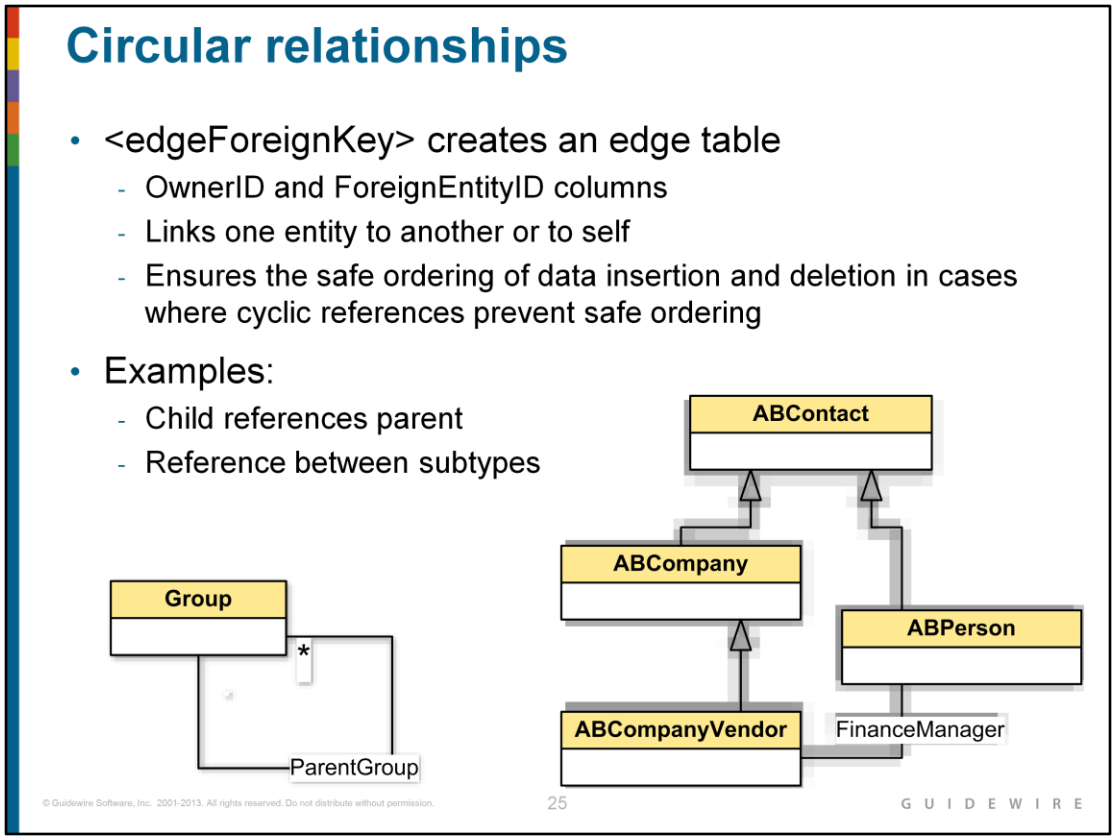


- To access the relationship, add an array to one or both ends of the relationship
- Example:
  - UserRole defines the foreign keys to Role and User
  - UserRole defines a unique index so a role appears once in user's list of roles
  - User defines a UserRole array entity named Roles

Many-to-many relationships require a separate entity that contains the required foreign keys and a unique index for the foreign key values so that duplicate rows are not added to the table. The entity type must be versionable. Joinarray is an internal type and Guidewire does not recommend using this entity type.

In order to access the many-to-many relationship values, one of the foreign key tables needs to specify an array entity.

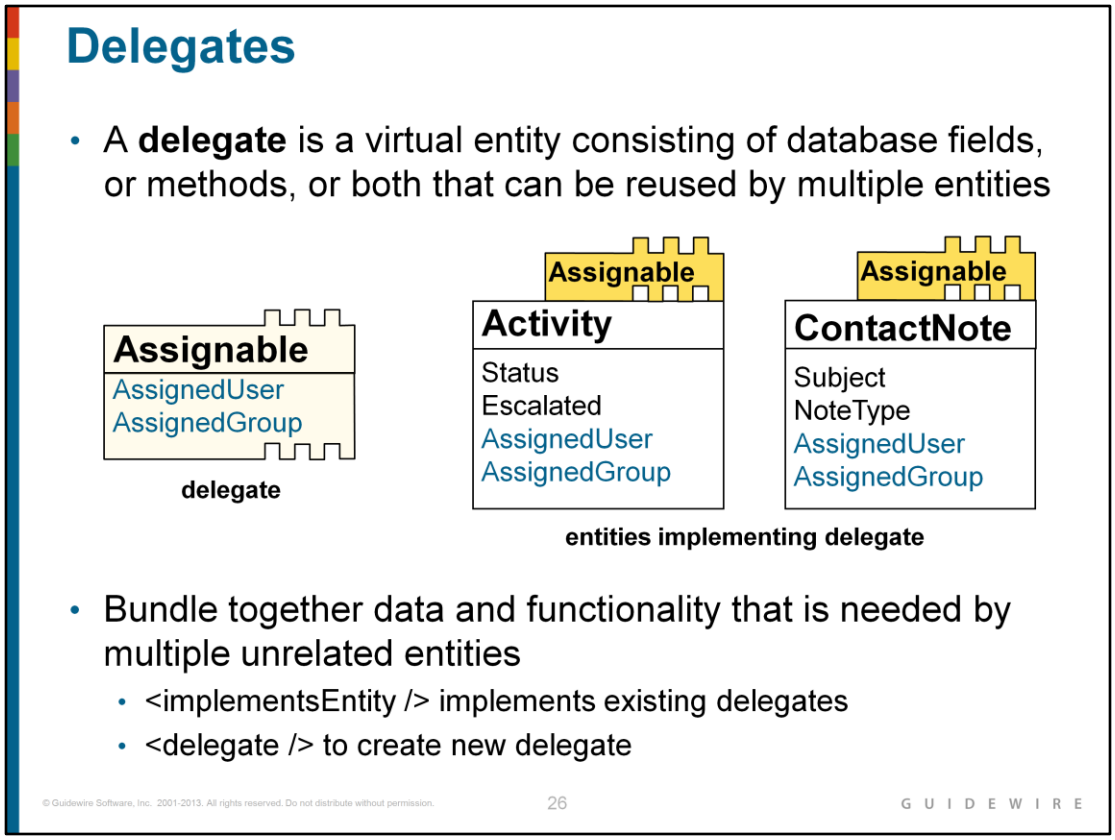
For more information on many-to-many relationships and entities of type joinarray, refer to documentation.



In some cases, an entity needs to refer to itself. This could occur both for a non-subtyped entity (every Group has a parent group) or a subtyped entity when one subtype refers to another (every ABCompany—which is an ABContact—can have a finance manager who is an ABPerson, which is also an ABContact). The Guidewire data model does not let an entity reference itself, or two or more entities reference one another in a cyclical manner, using only foreign keys. This is because Guidewire cannot easily determine the order in which rows can safely be written to the database when a data model cycle exists.

To solve this kind of circular dependency between foreign keys, Guidewire recommends that you use an edge foreign key (<edgeForeignKey>). An edge foreign key from A to B introduces a new, hidden entity that has a foreign key to A and a foreign key to B. However, it does not create any direct foreign key from A to B. As such, ClaimCenter can safely commit the A objects first, then the B objects, and finally, the hidden A/B edge foreign key entities. This ensures that the relationship information does not reference non-existent rows. The same principal is true for two or more entities that reference one another in a cyclical fashion. The cycle can be implemented using an edge foreign key, which allows all the individual rows to be written first, and the relationships among the rows to be written afterwards.

For more information on edge foreign keys and circular references, refer to documentation.



Guidewire offers at least three different features for sharing functionality across the data model: delegates, subtypes, and interfaces. A delegate is designed for situations where multiple entities need common functionality but are distinctly different.

Delegates are often used over subtypes because subtyping can be impractical. The entities in question share only a small set of fields in common and it would be impractical to store them in a single table. You might also need to give one entity delegate A, another delegate B, and a third delegates A and B. Subtyping would require all these entities to be in a single table. Delegates are often used over interfaces because interfaces do not define database fields.

You can add a delegate to an entity by using the implementsEntity element. Specify the delegate for the name value attribute.

For more information on delegates, refer to documentation.

Lesson objectives review

- You should now be able to:
  - Describe what is a custom entity and how to create a custom entity
  - Create entity elements and subelements that define fields, foreign keys, and arrays on entities
  - Identify entity related data elements associated with data model design

Review questions

1. What is the difference between an ETX file and an ETI file?
2. Does the base application have ETI files? Can developers create new ETI files? Where and how?
3. If the ABLawyer entity has an array of Cases, what type of field is required on the Case entity?
4. To deploy data model changes, are you required to:
   a) Regenerate the Data Dictionary?
   b) Restart the application server?
   c) Restart Studio?
5. How can you configure the name of all entity displayed in the User Interface?

Answers

1) An ETX file contains an <extension> element and extends a base application entity. An ETI file contains an <entity> element and defines a new entity.

2) Yes, ETI files are in the base application. Developers can create new ETI files in the …\extensions\entity\ folder using Guidewire Studio and the Entity Editor.

3) The Case entity requires a foreign key element (field) in the ETI definition that specifies the fkentity attribute as the ABLawyer entity.

4a) You are not required to regenerate the data dictionary. Regenerating the data dictionary will identify any errors in the data model. If successful, the process will add new and modified entities to the data dictionary.

4b) Yes, you must restart the application server to deploy the new and modified entities. The process creates new database tables and or upgraded the database. Internal Gosu classes are also created and modified.

4c) Studio is immediately aware of the data model changes. It is not necessary to restart Guidewire Studio.

5) You can use the Entity Editor to configure the displayname property of all instances of a specific entity.

Notices