What Happens If You Change the Data Model?

During server start up, ClaimCenter analyzes the metadata for changes since the last build. If you have made extensions, the application merges this into the working ClaimCenter data model which is the composite of the base entities and your extensions.

After merging the base data model with any extensions, ClaimCenter compares the startup layout to the physical schema in the current database. (Each ClaimCenter database stores schema version numbers and metadata checksums to optimize the analysis and comparison.)

If the application detects changes between the startup layout and the physical database schema, it initiates a database upgrade automatically. This keeps the physical schema synchronized with the schema defined by the XML metadata. By default, ClaimCenter refuses to start until the two are synchronized. By setting the autoupgrade parameter to false (within the database element in config.xml), you can configure ClaimCenter to report the need for an upgrade, but not actually perform it.

###### **Database Upgrade Triggers**

The upgrade utility initiates a database upgrade automatically at application server startup if there are additions, modifications, or extensions to any of the following:

|  |  |
| --- | --- |
| • | Data model version |
| • | Extensions version |

|  |  |
| --- | --- |
| • | Platform version |
| • | ClaimCenter data model |

|  |  |
| --- | --- |
| • | Field encryption |
| • | Typelists |

In addition to these generic changes, the following specific localization changes trigger a database upgrade:

|  |  |
| --- | --- |
| • | In file localization.xml, any change to the <LinguisticSearchCollation> subelement on the <GWLocale> element of the default application locale forces a database upgrade at application server startup. |
| • | In file collations.xml, any change to the source definition of the DBJavaClass definition forces a database upgrade at application server startup. |

### **Dynamic Filters**

A typecode filter uses categories and category lists at the typecode level to restrict or filter a typelist. Typecode filters function in an equivalent manner to dependent filters in that the a parent typecode filters the available values on the child typecode.

You define a typecode filter directly on a typecode. You do this through the Studio Typelist editor, by defining a filter on the Codes tab for a particular typecode. To create this filter, you select a specific typecode and set a filter (category) on that typecode.

There are two types of typecode filters that you can define on the Codes tab:

|  |  |
| --- | --- |
| **Filter type** | **Use to...** |
| [Category](javascript:WWHClickedPopup('config',%20'studio_typelist.25.12.html" \l "3606303', '');" \o "Dynamic Filters) | Associate one or more typecodes on a parent typelist with one or more typecodes on a child typelist. |
| [Category list](javascript:WWHClickedPopup('config',%20'studio_typelist.25.12.html" \l "3620193', '');" \o "Dynamic Filters) | Associate all the typecodes on a parent typelist with one or more typecodes on a child typelist. |

###### Category Typecode Filters

|  |  |
| --- | --- |
| • | You use a category filter to associate one or more typecodes from one or more typelists with a specific typecode on the filtered typelist. |
| • | You define a category filter in the Typelist editor on the Codes tab using the Categories pane. |

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a typecode category filter:

<typecode code=”DependentTypecode” desc=”DescriptionString” typelist=”DependentTypelistName”>

  <category code=”Typecode1” typelist=”Typelist1”/>

  <category code=”Typecode2” typelist=”Typelist1”/>

  <category code=”Typecode3” typelist=”Typelist2”/>

...

</typecode>

###### Category List Typecode Filters

|  |  |
| --- | --- |
| • | You use a category list filter to associate all of the typecodes from one or more typelists with a specific typecode on the filtered typelist. |
| • | You define a category list filter in the Typelists editor on the Codes tab using the Category Lists pane. |

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a typecode category list filter:

<typecode code=”Typecode” desc=”DescriptionString” typelist=”DependentTypelistName”>

  <categorylist typelist=”TypelistName”/>

</typecode>

### **What is the Data Model?**

At its simplest, the Guidewire data model is a set of XML-formatted metadata definitions of entities and typelists.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Entities | An entity defines a set of fields for information. You can add the following kinds of fields to an entity:   |  |  | | --- | --- | | • | Column | | • | Type key |  |  |  | | --- | --- | | • | Array | | • | Foreign key |  |  |  | | --- | --- | | • | Edge foreign key | |
| Typelists | A typelist defines a set code/value pairs, called typecodes, that you can specify as the allowable values for the type key fields of entities. Several levels of restriction control what you can modify in typelists:   |  |  | | --- | --- | | • | Internal typelists – You cannot modify internal typelists because the application depends upon them for internal application logic. | | • | Extendable typelists – You can modify this kind of typelist according to its schema definition. |  |  |  | | --- | --- | | • | Custom typelists – You can also create custom typelists for use on new fields on existing entities or for use with new entities. | |

Guidewire ClaimCenter loads the metadata of the data model on start-up. The loaded metadata instantiates the data model as a collection of tables in the application database. Also, the loaded metadata injects Java and Gosu classes in the application server to provide a programmatic interface to the entities and typelists in the database.

**Edge Foreign Key:**

#### edgeForeignKey>

You use the <edgeForeignKey> element to define a reference to another entity, in a manner similar to the [<foreignkey>](javascript:WWHClickedPopup('config',%20'entities.21.10.html#2637861', '');) element. However, you use an edge foreign key in place of a standard foreign key to break a cycle of foreign keys in the data model. Guidewire defines this element in the data model metadata files as the <edgeForeignKey> XML subelement.

|  |  |
| --- | --- |
|  | **The Data Model and Circular References** |

A chain of foreign keys can form a cycle, also known as a *circular reference*, in the data model. As an example of a circular reference, entity type A has a foreign key to entity type B, and B has a foreign key to A. Circular references can occur with more extensive chains of foreign keys, such as A refers to B, which refers to C, which refers to A. The PolicyCenter data model does not permit circular foreign keys reference, because PolicyCenter cannot determine a safe order for committing the entity instances in a circular reference to the database.

For example, entity types A and B have foreign key references to each other. The foreign keys create a circular reference. Suppose that a bundle contains a new instance of A and a new instance of B. The circular reference would cause a foreign key constraint to fail upon committing the bundle. If PolicyCenter commits A before B is committed and in the database, a constraint failure occurs on the foreign key from A to B. The converse order of committing B before A causes a similar failure.

An edge foreign key in place of a standard foreign key resolves circular references so PolicyCenter can determine a safe order for committing the entity instances within a cycle. An edge foreign key from A to B introduces a new, hidden associative entity with a foreign key to A and a foreign key to B. The edge foreign key associates A and B without establishing foreign keys in the database directly between them. With an edge foreign key, PolicyCenter can safely first commit new object A, then new object B, and finally the edge foreign key instance.

|  |
| --- |
| **Edge Foreign Keys and Associative Database Tables** |

An edge foreign key creates an associative table in the database. An associative table is essentially a table of foreign keys relationships. An associative table associates other database tables with each other but holds no other essential business data itself.

In PolicyCenter, the associative table that implements an edge foreign key has two columns:

|  |  |
| --- | --- |
| • | OwnerID |
| • | ForeignEntityID |

If entity instance A has an edge foreign key to entity type B, PolicyCenter creates a row in the edge foreign key table. The value in the row for OwnerID points to A and the value for ForeignEntityID points to B.

Every time you traverse, or dereference, the edge foreign key, PolicyCenter loads the join array.

|  |  |
| --- | --- |
| • | If the array is of size 0, then the value of the edgeForeignKey is null. |
| • | If the array is of size 1, the PolicyCenter follows the ForeignEntityID on the row |

A chain of ownership relationships can form a cycle known as an *ownership cycle* in the domain graph. Ownership cycles are hard to detect because ownership can flow either to or from an object that has a foreign key to another object.

By default, ownership flows in the same direction as foreign keys. For example, if B has a foreign key to A, B is owned by A. Sometimes it is necessary to invert the flow of ownership, so a foreign key points from the owner to the owned object instead.

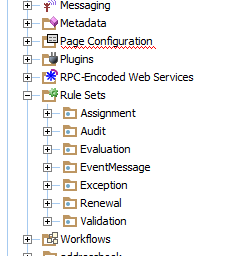
Guidewire strongly recommends against the use edge foreign keys to resolve ownership cycles in the domain graph. Introduce edge foreign keys into the domain graph only to resolve circular foreign key references that require edge foreign keys for safe ordering during bundle commit.

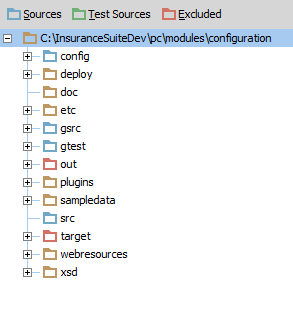
**Folder Structure:**

**Pc/modules/configuration/config**

* **EntityNames (.en files)**
* **Extensions (.etx and typelists)**
* **Localizations**
* **Messaging**
* **Metadata (GW entities and type lists)**
* **PCF**
* **Plugins**
* **Rule Sets**
* **Workflows**
* **Resources(generally system tables will be present)**

**Types of Rules:**





**Creating the System Table:**

1.In Studio, create a file named *entity*.eti.

|  |  |  |  |
| --- | --- | --- | --- |
| 2. | 2.In Studio, create a file named *entity*.xm  3.In Studio, add a new FileDefinition element for your new system table to systables.xml      <FileDefinition Name="*my\_class\_codes*.xml" Priority="0">          <Entity Type="*MyClassCodes*"/>      </FileDefinition>   |  |  | | --- | --- | |  | 4.Use Product Designer to fill in system table values. | |

**Product Model:**

The product model identifies the types of products and policies that your PolicyCenter configuration offers. For each product, the product model specifies the choices about the items that can be covered.

**Product Model Patterns:**

The product model consists of set of templates called patterns. PolicyCenter uses these patterns during policy transactions to generate specific instances of policies and policy objects. The product model provides a large number of patterns. The following patterns are the core patterns you use to start to define a new product.

Product – Creates an instance of a product, which is a policy type available to an applicant in the Submission Manager screen in PolicyCenter. A product is a pattern that creates new policy instances. PolicyCenter lists each product on a separate row of the New Submissions screen. Each product pattern contains at least one PolicyLinePattern.

PolicyLinePattern – Creates an instance of a policy line for a specific line of business, such as businessowners or personal auto. Each policy line pattern contains any number of clause patterns. A clause pattern is a generic term that refers to a coverage pattern, exclusion pattern, or condition pattern.

CoveragePattern – Creates an instance of a coverage, which is a type of loss covered by a policy. A coverage pattern creates an instance of a coverage on a specific policy line.

ExclusionPattern – Creates an instance of an exclusion, which is a type of loss explicitly not covered by a policy line. An exclusion pattern creates an instance of an exclusion on a specific policy.

ConditonPattern – Creates an instance of a condition, which is a contractual obligation that is neither providing nor excluding coverage. A condition pattern creates an instance of a condition on a specific policy.

CoverageTermPattern – Creates an instance of a value that specifies the extent, degree, or attribute of coverage, exclusion, or condition. Coverage terms measure or further define a specific clause pattern. One example of a coverage term is a deductible.

ModifierPattern – Creates an instance of a modifier that affects the calculation of the policy premium

**Coverage Symbol Group:**

Coverage symbol groups apply only to the commercial auto line. A commercial auto policy prints these symbols on the policy declarations page to indicate the types of coverages in effect.

**Question Sets:**

A question set is a collection of questions presented within PolicyCenter that gathers information about an applicant. You use the answers to these questions to evaluate the risk associated with a policy applicant.

**Types of Question Sets:**

PreQual – Questions that determine whether or not the insured meets the basic requirements for coverage by your carrier.

Product Qualification – Questions to assess whether or not an applicant is qualified for a particular product.

|  |  |
| --- | --- |
|  | **Location** – Questions that link to a particular location and used for underwriting. PolicyCenter is configured to display an additional tab when location questions are present. You can further configure PolicyCenter to display these questions when the location fits a particular profile or has certain associated coverages. For example, a question about whether the location serves alcohol could be used in businessowners products where the location has an industry code associated with restaurants. The answer then could be associated with liquor liability coverage. For another example, a question about whether the location has a swimming pool could be use if the location has an industry code associated with apartments or motels. |

Offering Selection – Questions associated with products that have offerings. Such products have an Offerings screen that appears at the beginning of a job wizard. The offerings available in the Offerings Selection list can be determined by answers to these questions.

Supplemental – Questions that collect additional underwriting information needed to assess risk. For example, in a workers’ compensation policy, a question asking if any employees are under age 16 or over age 60 could add a certain number of risk points. Another question about whether any employees travel out of state on business could add a smaller number of risk points.

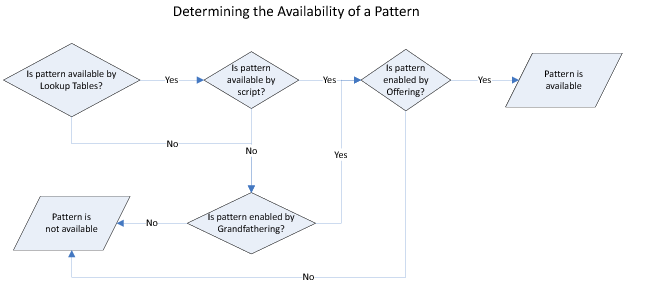
Underwriting – Questions that generally help to determine the quality of a risk. A higher-quality risk (one that is less likely to have a loss) could qualify for automated approval and a favorable rating. A lower-quality risk could be charged more for the policy, or possibly not qualify for the policy at all. Underwriting questions are typically evaluated as a set.

**Availability:**

If a pattern is unavailable, PolicyCenter does not expose that pattern and does not permit you to create instances from the pattern. In some cases, for example in the **Submission Manager**, PolicyCenter applies other criteria as well.

All of the product model patterns except policy line patterns have a configurable availability framework, including all of the core coverage patterns (CoveragePattern, CovTermPattern, CovTermOpt, and CovTermPack), exclusions, and conditions. However, PolicyCenter does not apply availability to policy line patterns, but instead controls policy line availability at the product level. There is never a need for a product pattern to be available and one of its underlying policy line patterns to be unavailable. Define other product patterns to make different policy line patterns available.

The availability calculation is a multiple step process as shown in the following illustration.



PolicyCenter controls availability using following mechanisms:

|  |  |
| --- | --- |
| • | Lookup tables(Availability Tables) |
| • | Availability scripts |

|  |  |
| --- | --- |
| • | Grandfathering |
| • | Offerings |

Not all mechanisms are available for every pattern type. For example, you cannot specify an availability script for a product pattern.

Grandfathering and Offerings

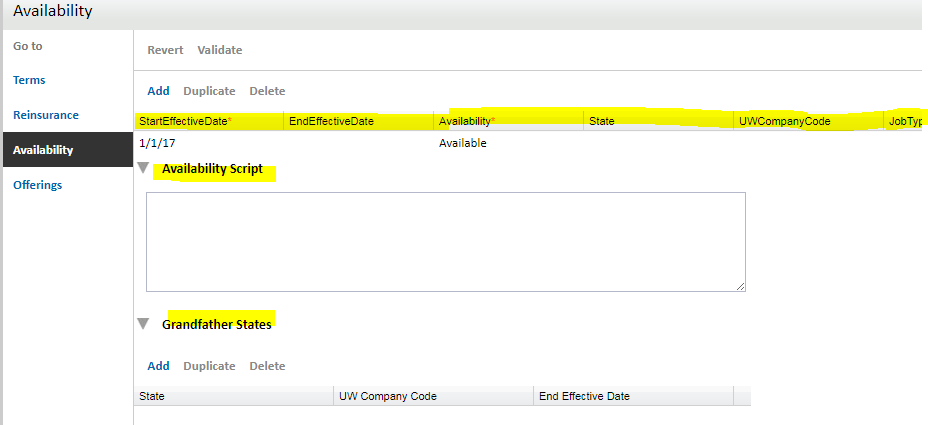
Grandfathering provides ways to continue to make various patterns available even if lookup tables and availability scripts determine that they are unavailable. Typically, grandfathering is used to maintain ongoing policy features to existing customers after those features are no longer offered to new customers. The pattern is unavailable when writing new business and cannot be added to the policy of an existing customer. However, with grandfathering enabled, the pattern is not removed from existing customers. Grandfathering can be applied to the following patterns:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | • | Coverages | | |  |  | | --- | --- | | • | Exclusions | |
| |  |  | | --- | --- | | • | Coverage term options | | |  |  | | --- | --- | | • | Conditions | |
| |  |  | | --- | --- | | • | Coverage term packages | | |  |  | | --- | --- | | • | Modifiers | |
| |  |  | | --- | --- | | • | Offerings | | |  |  | | --- | --- | | • | Modifier rate factors | |

Offerings can be used to tailor a product for a particular use case, such as a business-specific product or tiers of coverage. **PolicyCenter applies offering logic after grandfathering logic, which means that offerings can make a pattern unavailable even if grandfathering makes it available.** Offerings can control the availability of:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | • | Products | | |  |  | | --- | --- | | • | Exclusions | |
| |  |  | | --- | --- | | • | Policy lines, in package policies only | | |  |  | | --- | --- | | • | Conditions | |
| |  |  | | --- | --- | | • | Policy terms | | |  |  | | --- | --- | | • | Modifiers | |
| |  |  | | --- | --- | | • | Coverages | | |  |  | | --- | --- | | • | Question sets | |
| |  |  | | --- | --- | | • | Coverage terms | | |  |  | | --- | --- | | • | Individual questions within a question set | |
| |  |  | | --- | --- | | • | Coverage term options and packages | |  |

**Performance of availability lookup tables is much better than availability scripts. Consequently, Guidewire recommends that you specify availability through the availability tables whenever possible.**



IMPORTANT Every availability lookup table must contain at least one row. Product Designer displays validation errors and refuses to commit your changes if it detects any availability tables that do not have at least one row. If you define an availability table without at least one row by, for example, using Studio to edit the XML file, the PolicyCenter server refuses to start.

|  |
| --- |
| **Defining a Pattern as Available on a Job-by-Job Basis** |

The JobType column specifies the policy transaction type on which to base availability. One use of this field is to make a coverage or other pattern available on a policy transaction basis. For example, you can make a pattern available for new business only.

**Offerings** have their own availability and grandfathering configuration. Availability for offerings is checked last, after all other availability checks for a product model pattern. For this reason, if all other checks enable the pattern to be available, the offering availability check can make it unavailable. However, if any of the other checks makes the pattern unavailable, the offering availability check cannot make it available.

### **Static Filters**

A static typelist filter causes the typelist to display only a subset of the typecodes for that typelist. Therefore, a static filter narrows the list of typecodes to show in the typelist view in the application. Guidewire calls this kind of typelist filter a static typefilter.

You define a static filter at the level of the typelist. You do this through the Studio Typelist editor, by defining a typefilter for that particular typelist.

### **Dynamic Filters**

A typecode filter uses categories and category lists at the typecode level to restrict or filter a typelist. Typecode filters use a parent typecode to restrict the available values on the child typecode.

You define a typecode filter directly on a typecode. You do this through the Studio Typelist editor, by defining a filter for a particular typecode. To create this filter, you select a specific typecode and set a filter, or category, on that typecode.