

Custom processing

It is possible to implement a custom process to supplement the standard processes that Guidewire provides in the base configuration. In creating a custom process, consider the following factors in deciding between developing a custom work queue and developing a custom batch process:

- The volume of items in a typical batch
- The duration of the batch processing window

For business oriented batch processing, such as invoicing or aging, Guidewire recommends that you develop a custom work queue. Work queues process items in parallel tasks distributed across all servers in a PolicyCenter cluster that have the workqueue server role.

IMPORTANT: Regardless of the volume of items or the duration of the processing window, Guidewire recommends that you implement all custom batch processing as a custom work queue.

Manually executing PolicyCenter processes

PolicyCenter provides the following Server Tools screens for use by administrators in manually executing the different application processes:

- **Batch Process Info**
- **Work Queue Info**

Batch processing typecodes

PolicyCenter identifies and manages application batch processing by typecodes in the `BatchProcessType` typelist. Each type of batch processing, whether implemented as a batch process or a work queue, has a unique typecode in the typelist. Whenever you develop a type of custom batch processing, begin by defining a typecode for it the `BatchProcessType` typelist.

You use the typecode for your type of custom batch processing in the following ways:

- Arguments to some of the methods in your custom work queue or batch process class
- An argument to the maintenance tools command and web service that enable you to start a batch processing run
- Categorizing how your custom batch process can be run, from the administrative user interface, on a schedule, or from the maintenance tools command prompt or web service
- A case clause in the Batch Processing Completion plugin for your type of batch processing

Regardless the mode of batch processing you implement, first define a new typecode in the BatchProcessType typelist for your type of batch processing..

About scheduling PolicyCenter processes

It is possible to schedule many, if not most, of the PolicyCenter processes for execution at periodic intervals. Guidewire recommends that you take the following information into account in setting up a processing schedule.

Night-time batch processing

Processes business transactions accumulated at the end of specific business periods, such as business days, months, quarters, or years.

Day-time batch processing

Defers to periodic asynchronous background processing complex transactional processing triggered by user actions.

Depending on your custom process, certain implementation details can vary.

The scheduler configuration file

File `scheduler-config.xml` defines the default schedule that PolicyCenter uses to launch many of the processes, including writer processes for work queues. You can find this file in the following location in the Studio **Project** window:

```
configuration > config > scheduler
```

It is possible to define your own schedule for a process.

Night-time processing

Night-time processing typically processes relatively large batches of work, such as processing premium payments that accumulate during the business day. Each type of night-time processing runs once during the night, when users are not active in the application.

Because most kinds of night-time processing typically operate on large batches and have rigid processing windows, develop your night-time process as a custom work queue. Night-time processing often requires processing items in parallel to achieve sufficient throughput. Large workloads of nightly processing are typically too severe for the servers in a PolicyCenter cluster that have the batch server role.

For night-time batch processing, you typically configure a custom work queue table to segregate the work items of different work queues from each other. Each type of night-time processing typically has many worker tasks simultaneously accessing the queue for available work items. Using the same work queue table for all types of nightly work queues can degrade processing throughput due to table contention. In addition, segregating work items from different work queues into separate tables can ease recovery of failed work items before the nightly processing window closes.

Night-time processing frequently requires chaining so that completion of one type of batch processing starts another type of follow-on processing. Regardless of implementation mode – work queue or batch process – you most likely must develop custom process completion logic for your type of night-time processing.

Daytime processing

Typically, daytime processes are relatively small batches of work, such as reassigning activities escalated by users. Daytime processes run frequently during the day, while users are active in the application. You might schedule different types of daytime processes to run every hour or even every few minutes.

Even though daytime processing typically operates on small batches and lacks rigid processing windows, develop your type of daytime batch processing as a custom work queue. Although daytime processing often achieves sufficient throughput by processing items sequentially, its workload on the servers with the batch server role can slow overall performance of the application. As a work queue, worker tasks on multiple servers (with the workqueue server role), can perform the work in parallel.

If you develop your daytime processing as a custom work queue, you typically can use the standard work queue table for your work items. Different types of daytime processing run intermittently and in different bursts of relatively short work. So, table contention between various types of daytime batch processing often is minimal.

Daytime processes seldom requires chaining. Completion of one type of daytime process seldom starts another type of follow-on processing. Regardless of implementation mode, you most likely do not need to develop custom process completion logic for your type of daytime processing.

Custom work queues

A work queue is code that runs without human intervention as a background process on multiple servers to process the units of work for a batch in parallel. Develop a custom work queue if you require the processing of units of work in parallel to achieve an acceptable throughput rate.

About custom work queues

A custom work queues comprises the following components.

Writer

The `findTargets` method on a Gosu class that extends the `WorkQueueBase` class that selects the units of work for a batch and writes work items for them in the work queue table.

Work queue

An entity type that implements the `WorkItem` delegate, such as the `StandardWorkItem` entity, to establish the database table for the work queue and its work items.

Worker

The `processWorkItem` method on the same Gosu class that extends the `WorkQueueBase` base class that processes the units of work identified on the work queue by work items

Starting the writer initiates a run of the type of batch processing that a work queue performs. The batch is complete when the workers exhaust the queue of all work items in the batch, except those they fail to process successfully.

About custom work queue classes

You implement the code for your writer and your workers as methods on a single Gosu class. You must derive your custom work queue class from the `WorkQueueBase` class. This base class and the work queue framework provide most of the logic for managing the work items in your work queue. You typically need override only few methods in the base class to implement the code for your writer and your workers.

Work queue writer

You implement the writer for your work queue by overriding the `findTargets` method inherited from the base class. Your code selects the units of work for a batch and returns an iterator for the result set. The work queue framework then uses the iterator to create and insert work items into your work queue. Each work items holds the instance ID of a unit of work in your result set.

For example, your custom work queue sends email about overdue activities to their assignees. In this example, instances of the `Activity` entity type are the units of work for the work queue. In your `findTargets` methods, you query the database for activity instances in which the last viewed date of an open activities exceeds 30 days. You return an iterator to the result set, and then the framework creates a work item for each element in the result.

Work queue workers

You implement the workers of your work queue by overriding the `processWorkItem` method inherited from the base class. The work queue framework calls the method with a work item as the sole parameter. Your code accesses the unit of work identified in the work item and processes it to completion. Upon completion, your code returns from the method, and the work queue framework deletes the work item from the work queue.

For example, the units of work for your work queue are open activities that have not been viewed in the past 30 days. In your `processWorkItem` method, you access the activity instance identified by the work item. Then, you generate

and send an email message to the assignee of that activity. After your code sends the email, it returns from the method. The framework then deletes the work item and updates the count of successfully processed work items in the process history for the batch.

Work queues and work item entity types

A work item is an instance of entity type that implements the `WorkItem` delegate, such as `StandardWorkItem`. The entity type provides the database table in which the work items for your custom work queue persist. Work items in the work queue table are accessible from all servers in a PolicyCenter cluster. Thus, workers can access the work items asynchronously, in parallel, and distribute the work items to servers with the `workqueue` server role.

Work items have many properties that the work queue framework uses to manage work items and the process histories of work queue batches. If you use `StandardWorkItem` for your work queue, the only field on a work item that your custom code uses is the `Target` field. The `Target` field holds the ID of a unit of work that your writer selected. The code for your writer and your workers can safely ignore the other properties on work item instances. However, you can define a custom work item type with additional fields for your custom code to use.

Custom work item types

Guidewire recommends that you define your custom work item entities as `keyable`, rather than `retireable`. If you define a custom work item as `retireable`, then you also need to create a custom batch process to purge the work items after PolicyCenter processes them and they become old and stale. This batch process also needs to clean up failed work items as well. Any delay in purging failed or stale work items can potentially prevent further operations.

Work queues that use `StandardWorkItem`

Guidewire supports creating multiple work queues that use the same work item type only if that work item is the `StandardWorkItem` type.

In the PolicyCenter data model, the `StandardWorkItem` entity type implements the `WorkItem` delegate. The `StandardWorkItem` entity type thus inherits the following from the `WorkItem` delegate:

- Methods to manipulate the work item
- Entity fields that store such items as `Status`, `Priority`, `Attempts`, and other similar types of information

Guidewire marks the `StandardWorkItem` entity type as `final`. Thus, it is not possible to subtype this entity type.

The `StandardWorkItem` entity type contains a `QueueType` typekey, which obtains its value from the `BatchProcessType` typelist. It is the use of this typekey that makes it possible for multiple work queues to use the `StandardWorkItem` type simultaneously.

For example, to query for all `StandardWorkItem` work items of a certain type, use a query that is similar to the following:

```
var target = Query.make(StandardWorkItem).compare(StandardWorkItem#QueueType, Equals, BatchProcessType.TC_CUSTOMWORKITEM)
```

This query returns all work items for the process that match the filter criteria. The query includes failed work items and work items in progress, not just the work items available for selection.

Error: Two work queues cannot use the same work item type

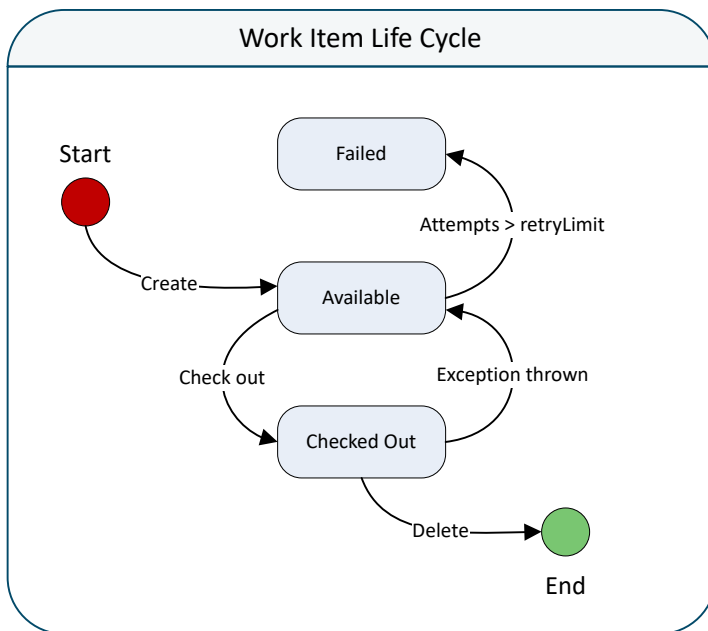
Guidewire does not support creating multiple work queues that use the same work item type, except for the `StandardWorkItem` type. If you attempt to do so using a work item type other than `StandardWorkItem`, PolicyCenter throws the following error:

```
Two work queues cannot use the same work item type at runtime: workItemType.Name
```

It is not possible for workers from different work queues to process work items from the same work item database table. In this circumstance, it is not possible for the different sets of workers to determine which work items to process. In any case, it is very likely that there can be database contention issues along with race conditions between the workers in picking up the work items from the queue.

Lifecycle of a work item

The **Status** field of a work item records the current state of its lifecycle. The work queue framework manages this field and the lifecycle of work items for your writer and workers. The following diagram illustrates the lifecycle.



A work item begins life after the writer selects the units of work for a batch and returns an iterator for the collection. The framework then creates work items that reference the units of work for a batch. The initial state of a work item is `available`. At the time the framework checks out a quota of work items for a worker, the state of the work items becomes `checkedout`. The framework then hands the quota of work items to the worker one at a time. After a worker finishes processing a work item successfully, the framework deletes it from the work queue and updates the statistics in the process history for the batch.

Returning work items to the work queue

Sometimes a worker cannot finish processing a work item successfully. For example, a network resource may be temporarily unavailable. Or, a concurrent data change exception (CDCE) can occur if multiple workers simultaneously update common entity data in the domain graphs of two separate units of work. Whenever errors occur, the worker throws an exception to return the work item to the work queue as `available` again.

Whenever a worker returns a work item to the work queue, PolicyCenter increments the `Attempts` property on the work item. If the value of `Attempts` remains below the retry limit for the worker, the state of the work item remains `available`. The work item is subsequently checked out in a quota for the same or another worker. If the temporary error condition clears, the next worker completes it successfully and PolicyCenter removes the work item from the work queue.

Work items that fail

Whenever a worker returns a checked out work item and the `Attempts` property exceeds the work item retry limit, the state of the work item becomes `failed`. Failed work items end their lifecycle in this state. Workers ignore items with a status of `failed` and no longer attempt to process them. Someone must determine the reason for the failure of a work item and take corrective action. They can then remove the failed work item from the work queue.

Considerations for developing a custom work queue

Before you develop the Gosu code for your custom work queue, you need to decide which of the following you want to use for your work queue table:

- A `StandardWorkItem` entity type
- A custom work item type

The following reasons outline why you would want to define a custom work item type:

- Include custom fields on work items not available on the `StandardWorkItem` entity type
- Avoid table contention with other work queues that typically process large batches at the same time

Whenever you define a custom work item type, you must implement the `createWorkItem` method that your custom work queue inherits from `WorkQueueBase`.

If you create a custom work queue, your writer can pass more fields to the workers than a target field. On `StandardWorkItem`, the `Target` field is an object reference to an entity instance, which represents the unit of work for the workers. In a custom work item, you can define as many fields as you want to pass to the workers. Your custom work item itself can be the unit of work.

Guidewire recommends that you use custom work queue types for work queues with typically large batches that potentially run at the same time as other work queues with large batches. Especially if developing a custom work queue for night-time processing, consider using a custom work queue subtype instead of using `StandardWorkItem`. If you create a custom work item subtype to avoid table contention, you often define a single field for the writer to set, much like the `Target` field on `StandardWorkItem`. By convention, you name the single field with the same name as the entity type, not the generic name `Target`.

IMPORTANT: Do not create multiple work queues that use the same work item type other than the `StandardWorkItem` type. If you attempt to do so, PolicyCenter throws an error.

Linking custom work items to target entities

In creating a custom work item, you need to create a link from the custom work item table to the entity table that is the target of the work item. To create this link, add a column element to your custom work item and use the following parameters.

Name	Value
name	Target
type	softentityreference
nullokk	false

For `Target`, enter the name of the entity on which the work items acts. Use `softentityreference` (a soft link) rather than `foreignkey` (a hard link). A `softentityreference` is a foreign key for which PolicyCenter does not enforce integrity constraints in the database. The use of a hard foreign key in this context would do the following:

- Require that you delete the work item row from the database before you delete the linked entity row.
- Can force PolicyCenter to include the custom work item table in the main domain graph.

Keyable work item entities


Guidewire recommends that you define your custom work item entities as `keyable`, rather than `retireable`. If you define a custom work item as `retireable`, then you also need to create a custom batch process to purge the work items after PolicyCenter processes them and they become old and stale. This batch process also needs to clean up failed work items as well. Any delay in purging failed or stale work items can potentially prevent further operations.

Define a typecode for a custom work queue

About this task

Before you begin developing the Gosu code for your custom work queue, you need to define the custom work queue type. For your custom type, you need to add a typecode for it in the `BatchProcessType` typelist. The constructor of your custom work queue class requires the typecode as an argument.

Procedure

1. Open Guidewire Studio
2. In the Studio **Project** window, expand **configuration** > **config** > **Extensions** > **Typelist**.
3. Open `BatchProcessType.ttx`.
4. Click the green plus button, , to add a typecode.
5. In the panel of fields on the right, specify appropriate values for the following fields:

code	Specify a code value that uniquely identifies your work queue. This code value identifies the work queue in configuration files.
name	Specify a human readable identifier for your work queue. This name appears for the work queue writer on the Server Tools Batch Process Info screen.
desc	Provide a short description of what your custom process accomplishes. This description appears for your work queue on the Server Tools Batch Process Info screen.



Define a custom work item type

Procedure

1. Open Guidewire Studio.
2. Add a new entity for your custom work item type:
 - a) In the Studio **Project** window, expand **configuration** > **config** > **Extensions** > **Entity**.
 - b) Right-click **Entity**, and then select **New** > **Entity**.
 - c) In the Entity dialog box, enter the appropriate values.
For example, if creating a new work item entity called `MyWorkItem`, enter the following values:

Field	Example value
Entity	<code>MyWorkItem</code>
Entity Type entity	
Desc	Custom work item
Table	<code>myworkitem</code>
Type	<code>keyable</code>

Accept all the other default values in the dialog box.

- d) Click **OK**.
3. Set the entity to implement the correct work item delegate:
 - a) In the toolbar, in the drop-down list next to the green plus button, , select **implementsEntity**.
 - b) In the panel of fields on the right, select `WorkItem` from the **name** drop-down list.
 4. Add a soft reference to the unit of work for your custom work queue:
 - a) Select the **column** field type in the drop-down list next to the green plus button, . Studio inserts a new **column** element in the element table.
 - b) In the panel of fields on the right, enter the entity name for the unit of work for the **name** value.
For example, if the unit of work is an instance of an `Activity` object, add the following values at the panel at the right:

Field	Example value
name	<code>Activity</code>

Field	Example value
type	Activity
nullokk	false

- (Optional) Add other fields to your custom entity as meets your business needs.

What to do next

Whenever you define a custom work item for a work queue, your custom work queue class must implement the `createWorkItem` method to write the work item to the queue.

Creating a custom work queue class

After you define a typecode for your custom work queue and possibly create a custom work item type for it, you are ready to create your custom work queue class. This class contains the programming logic for the writer and the workers of your work queue. You must derive your class from `WorkQueueBase`, and you generally must override the following two methods.

<code>findTargets</code>	Logic for the writer, which selects units of work for a batch and returns an iterator for the result set
<code>processWorkItem</code>	Logic for the workers, which operates on a single unit of work selected by the writer

The `WorkQueueBase` provides other methods that you can override such as `shouldProcessItem`. However, you can generally develop a successful custom work queue by overriding the two required methods `findTargets` and `processWorkItem`.

WARNING: Do not implement multi-threaded programming in custom work queues derived from `WorkQueueBase`.

Custom work queue class declaration

In the declaration of your custom work queue class, you must include the `extends` clause to derive your work queue class from `WorkQueueBase`. Because the base class is a template class, your class declaration must specify the entity types for the unit of works and for the work items in your work queue.

The following example code declares a custom work queue type that has `Activity` as the target, or unit of work, type. It also declares that `StandardWorkItem` as the work queue type.

```
class MyWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> { ... }
```

Custom work queue class constructor

You must implement a constructor in your custom work queue class. The constructor that you implement calls the constructor in the `WorkQueueBase` class. Your constructor associates the custom work queue class at runtime with its typecode in the `BatchProcessType` typelist, its work queue item type, and its target type.

The following example code is the constructor for a custom work queue. It associates the class at runtime with its batch process typecode `MyWorkQueue`. The code associates the work queue with the `StandardWorkItem` entity type. So, the work queue shares its work queue table with many other work queues. The code associates the work queue with the `Activity` entity type as its target unit of work type. So, the writer and the workers operate on `Activity` instances.

```
construct () {
    super (typekey.BatchProcessType.TC_MYWORKQUEUE, StandardWorkItem, Activity)
}
```

Do not include any code in the constructor of your custom work queue other than calling the super class constructor.

Developing the writer for your custom work queue

In your custom work queue class, override the `findTargets` method that your custom work queue class inherits from `WorkQueueBase` to provide your specific logic for a writer task.

IMPORTANT: Do not operate on any of the units of work in a batch within your writer logic. Otherwise, process history statistics for the batch will be incorrect.

In the `findTargets` method logic, return a query builder iterator with the results you want as targets for the work items in a batch. PolicyCenter uses the iterator to write the work items to the work queue. The `findTargets` method is a template method for which you specify the target entity type, the same type for which you make a query builder object.

The following example code is a writer for a work queue that operates on `Activity` instances. The query selects activities that have not been viewed for five days or more and returns the iterator.

```
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate, LessThanOrEquals, java.util.Date.Today.addBusinessDays(-5))

    return targetsQuery.select().iterator()
}
```

Returning an empty iterator

Generally, if your writer returns an iterator with items found by the query, PolicyCenter creates a `ProcessHistory` instance for the batch run. Sometimes the query in your writer finds no qualifying items. If your writer returns an empty iterator, PolicyCenter does not create a process history for that batch processing run.

Writing custom work item types

Suppose that you are creating a custom work queue class and decide not to use the `StandardWorkItem` work item and instead define a custom work item type. If you create your own class, you must override the `createWorkItem` method inherited from `WorkQueueBase` to write new work items to your custom work queue table. The `createWorkItem` method has two parameters.

- **target** – An object reference to an entity instance in the iterator returned from the `findTargets` method.
- **safeBundle** – A transaction bundle provided by PolicyCenter to manage updates to the work queue table.

Your method implementation must create a new work item of the custom work item type that you defined. Pass the `safeBundle` parameter in the new work item statement. Then, assign the `target` parameter to the target unit of work field that you defined for your custom work item type. If you defined additional fields, assign values to them, as well.

The return type for the `createWorkItem` method is any entity type that implements the `WorkItem` delegate. Return your new custom work item type.

The following Gosu example code creates a new custom work item that has a single custom field, an `Activity` instance. The implementation gives the target field in the parameter list the name `activity` to clarify the code. The custom work item type is `MyWorkItem`.

```
override function createWorkItem (activity : Activity, safeBundle : Bundle) : MyWorkItem {
    var customWorkItem = new MyWorkItem(safeBundle)
    customWorkItem.Activity = activity
    return customWorkItem
}
```

Developing workers for your custom work queue

In your custom work queue class, override the `processWorkItem` method that your custom work queue class inherits from `WorkQueueBase` to provide the programming logic for a worker task. The workers of a work queue are inherently

single threaded. Do not attempt to improve the processing rate of individual workers by spawning threads from within your `processWorkItem` method.

The `processWorkItem` has a work item as its single parameter. You access the target, or unit of work, for the worker through the `WorkItem.Target` property. The type and target type of the work item are the types that you specified in the constructor of your custom work queue class derived from `WorkQueueBase`. At the same time that PolicyCenter calls the `processWorkItem` method, it sets the `Status` field of the work item to `checkedout`.

Successful work items

In your custom work queue class, return from the `processWorkItem` method after a worker finishes operating on a target instance. PolicyCenter then deletes the work item from the work queue.

The following example code extracts the targeted unit of work, an `Activity` instance, from the work item parameter. Then, the code sends the assigned user an email message.

```
override function processWorkItem (WorkItem : StandardWorkItem): void {
    // Extract the unit of work: an Activity instance
    var activity = extractTarget(WorkItem)

    if (activity.AssignedUser.Contact.EmailAddress1 != null) {
        // Send an email to the user assigned to the Activity.
        mailUtil.sendEmailWithBody(null,
            activity.AssignedUser.Contact,
            null,
            "Activity not viewed for five days",
            "Take a look at activity " + target.Subject + ", due on " + target.TargetDate + ".")
        // To:
        // From:
        // Subject:
        // Body:
    }

    return
}
```

In your override of the base `processWorkItem` method, specify the same work item entity type that you specified in the constructor of your custom work queue class.

Updates to entity data

The abstract `WorkQueueBase.processWorkItem` method does not have a bundle to manage database transactions related to targeted units of work. A bundle does exist at the time PolicyCenter calls your custom `processWorkItem` method, but PolicyCenter uses that bundle for updates to the work item. To modify entity data, you must create a bundle using the `Transaction.runWithNewBundle` API in your custom work queue class.

The following example code uses the `runWithNewBundle` transaction method to update the `EscalationDate` on the `Activity` instance from the work items that is processes.

```
uses gw.transaction.Transaction
...
override function processWorkItem (WorkItem : StandardWorkItem): void {
    // Extract the unit of work: an Activity instance
    var activity = extractTarget(WorkItem)

    // Update the activity escalation date
    Transaction.runWithNewBundle( \ bundle -> {
        activity = bundle.add(activity) // add the activity to the new bundle
        activity.EscalationDate = java.util.Date.Today } ) // update the escalation date

    return
}
```

Managing failed work items

If your custom worker code encounters an error while operating on a target instance, throw an exception. PolicyCenter detects and throws some types of exceptions automatically, such as a concurrent data change exception (CDCE). These types of exceptions generally resolve themselves quickly and automatically. However, it is also possible that your code detects other logic errors that require human intervention to resolve. If so, implement a custom exception and throw it whenever your worker code detects the exceptional situation.

PolicyCenter catches exceptions thrown by code within the scope of your custom `processWorkItem` method. Whenever PolicyCenter catches an exception, it increments the `Attempts` property on the work item. If the value of the `Attempts` property does not exceed the value of the `WorkItemRetryLimit` parameter set in `config.xml`, PolicyCenter sets the `Status` property of the work item to `available`. Otherwise, PolicyCenter sets the `Status` property of the work item to `failed`.

Retrying work items that cause exceptions

PolicyCenter makes work items that trigger exceptions available again on the work queue, because many exceptions resolve themselves quickly. For example, a CDCE exception often occurs whenever two worker tasks attempt to update common data related to their two separate units of work. By the time PolicyCenter gives a work item that encountered an exception to another worker task, the exceptional condition often resolves itself. The second worker then process the work item to completion successfully.

Handling exceptions

In your custom work queue class, provide an implementation of the `handleException` method inherited from `WorkQueueBase` to augment the actions taken whenever code in the `processWorkItem` method throws an exception. For example, it is possible for your worker code to throw an exception whenever it encounters a logic error that cannot resolve itself without human intervention.

Bulk insert work queues

Guidewire recommends that you implement a custom work queue class that extends `BulkInsertWorkQueueBase`, rather than `WorkQueueBase` if it is possible for a query to determine the work items to create.

The use of the `BulkInsertWorkQueueBase` class provides the following performance optimizations:

- The work queue writer does not add duplicate entries to the work queue.
- The use of the bulk insert method eliminates the need to fetch and commit each work item individually.

As a consequence, a work queue that subclasses `BulkInsertWorkQueueBase` performs its work much more quickly than a work queues that subclasses `WorkQueueBase`.

Overview of bulk insert work queues

If your type of process works extremely large work batches, it is possible for the `WorkQueueBase.findTargets` method to return an iterator that exceeds the memory capacity of the work queue server. In some cases, the method returns an iterator for batches that typically number hundreds of thousands of units of work.

Thus, Guidewire recommends that you implement your custom work queue as a class that extends `BulkInsertWorkQueueBase` instead of `WorkQueueBase`. In your class, implement the `BulkInsertWorkQueueBase.buildBulkInsertSelect` method for your query logic, instead of the `WorkQueueBase.findTargets` method.

PolicyCenter calls the `buildBulkInsertSelect` method with a query object that has no restrictions on the target entity type that you specify in the class constructor. However, bulk insert work queues support standard work items only. Thus, do not attempt to use custom fields on work items to pass any data other than target entity instances to the workers of your bulk insert work queue.

Use the query builder APIs in your code to add restrictions, including restrictions based on joins, that select the targets for a batch. After your code returns from `buildBulkInsertSelect`, PolicyCenter submits a `SELECT` statement based on the query object directly to the database. The database then uses its native bulk insert capabilities to insert standard work items for the batch directly into the standard work queue table.

Bulk insert work queues are suitable if you can reduce the selection criteria for the targeted units of work to a single query builder query object. Because PolicyCenter creates and inserts work items at the database level, it is not necessary to implement the following methods:

- `createWorkItem`

- `shouldProcessItem`

The `BulkInsertWorkQueueBase` base class is a subclass of `WorkQueueBase`, so you must also implement method `processWorkItem` for the worker logic of a bulk insert work queue.

Bulk insert work queue declaration and constructor

In the declaration of your bulk insert work queue class, you must include the `extends` clause to derive your work queue class from abstract class `BulkInsertWorkQueueBase`. Your custom class must include a constructor that calls the constructor in the base class using the `super` syntax. Your constructor associates your bulk insert work queue class at runtime with the following items:

- The work queue typecode in the `BatchProcessType` typelist
- The work queue item type
- The user under which the database transaction runs

The following example code declares a bulk insert work queue type and implements a constructor. Unlike a work queue class that you derive from `WorkQueueBase`, the constructor does not specify the entity type of the `Target` fields on the work items for the work queue.

```
class MyBulkInsertWorkQueue extends BulkInsertWorkQueueBase <User, StandardWorkItem> {
    construct () {
        super(BatchProcessType.TC_MYBULKINSERTWORKQUEUE, StandardWorkItem, User)
    }
    ...
}
```

Querying for targets of a bulk insert work queue

You provide the query logic for the writer thread of a bulk insert work queue by overriding the abstract `buildBulkInsertSelect` method that your custom work queue class inherits from `BulkInsertWorkQueueBase`.

The following Gosu example code selects the units of work for a batch run, which are users with a status of On Vacation.

```
override function buildBulkInsertSelect(query : InsertSelectBuilder, args: List<Object>) {
    query.SourceQuery.compare( User#VacationStatus.PropertyInfo.Name, Relop.Equals,
        VacationStatusType.TC_ONVACATION )
}
```

The following Gosu example code selects the units of work for a batch run, which are activities that no one has viewed in five days or more. Notice that this implementation of the method uses the `extractSourceQuery` method to generate the query result.

```
override function buildBulkInsertSelect(builder : Object, args: List<Object>) {
    extractSourceQuery(builder).compare( Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5) )
}
```

Eliminating duplicate items in bulk insert work queue queries

In constructing the query for your custom work queue class, Guidewire recommends that you include the following method, which excludes duplicate `StandardWorkItem` work items from the query:

```
excludeDuplicatesOnStandardWorkItem(java.lang.Object opaqueBuilder, boolean allowFailedDuplicates)
```

The `excludeDuplicatesOnStandardWorkItem` method takes the following parameters:

<code>opaqueBuilder</code>	The query builder to use.
<code>allowFailedDuplicates</code> (Boolean)	Whether to recreate a failed work item.

The `excludeDuplicatesOnStandardWorkItem` helper method excludes `StandardWorkItem` work items only. If you query on another work item type, use custom code similar to the following to exclude the duplicate work items:

```
builder.mapColumn(XXWorkItem#Target.PropertyInfo as IEntityPropertyInfo,
  XX#Id.PropertyInfo as IEntityPropertyInfo)

var subQuery = Queries.createQuery(XXWorkItem)
if allowFailedDuplicates subQuery.compare(XXWorkItem#Status.PropertyInfo.Name, Relop.NotEquals,
  WorkItemStatusType.TC_FAILED)

builder.SourceQuery.subselect(XX#Id.PropertyInfo.Name, CompareNotIn, subQuery,
  XXWorkItem#Target.PropertyInfo.Name)
```

Processing work items in a custom bulk insert work queue

You provide the programming logic for the writer thread of a bulk insert work queue by implementing the abstract `processWorkItem` method that your custom work queue class inherits from `BulkInsertWorkQueueBase`. The `processWorkItem` method takes a single argument, which is the work item type to use as the unit of work.

The following Gosu example code processes the units of work for a batch run, which are activities that no one has viewed in five days or more.

```
override function processWorkItem(workItem: StandardWorkItem) {
  // Extract an object reference to the unit of work: an Activity instance
  var activity = extractTarget(workItem) // Convert the ID of the target to an object reference

  if (activity.AssignedByUser.Contact.EmailAddress1 != null) {
    // Send an email to the user assigned to the activity
    EmailUtil.sendEmailWithBody(null, activity.AssignedUser.Contact,
      null,
      "Activity not viewed for five days",
      "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".")
  }
}
```

Example work queue that bulk inserts its work items

The following Gosu code is an example of a custom work queue that uses bulk insert to write work items for a batch to its work queue. This custom work queue class derives from the base class `BulkInsertWorkQueueBase` instead of from `WorkQueueBase`. The code provides an implementation of the `buildBulkInsertSelect` method instead of `findTargets` for its writer logic. The code provides an implementation of the `processWorkItemMethod` for its worker logic.

Bulk insert work queues like this one use the `StandardWorkItem` entity type for its work queue table. To use a custom work queue, you must implement `createBatchProcess` method to tell PolicyCenter in which table to insert the work items created from the `SELECT` statement.

The unit of work for this work queue is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date. The process updates entity data, so the `processWorkItemMethod` uses the `runWithNewBundle` API.

```
uses gw.api.database.Query
uses gw.api.database.Relop
uses gw.api.email.EmailUtil
uses gw.processes.BulkInsertWorkQueueBase

/**
 * An example of a process implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyOptimizedActivityEmailWorkQueue extends BulkInsertWorkQueueBase <Activity, StandardWorkItem> {

  /**
   * Let the base class register this type of custom batch processing by
   * passing the batch processing type, the entity type the work queue item, and
   * the target type for the units of work.
   */

  construct() {
    super ( BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity )
  }
}
```

```

/**
 * Select the units of work for a batch run: activities that have not been
 * viewed for five days or more.
 */
override function buildBulkInsertSelect(builder : Object, args: List<Object>) {
    excludeDuplicatesOnStandardWorkItem(builder, true); // if using StandardWorkItem
    extractSourceQuery(builder).compare(
        Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5) )
}

/**
 * Process a unit of work: an activity not viewed for five days or more
 */
override function processWorkItem(workItem: StandardWorkItem) {
    // Extract an object referent to the unit of work: an Activity instance
    var activity = extractTarget(workItem) // Convert the ID of the target to an object reference

    if (activity.AssignedByUser.Contact.EmailAddress1 != null) {
        // Send an email to the user assigned to the activity
        EmailUtil.sendEmailWithBody(null, activity.AssignedUser.Contact,
            null, // To:
            "Activity not viewed for five days", // From:
            "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".") // Subject:
        // Body:
    }
}
}

```

Examples of custom work queues

Example work queue that extends WorkQueueBase

The following Gosu code is an example of a simple custom work queue. It uses the `StandardWorkItem` entity type for its work queue table. Its unit of work is an activity that no one has viewed for five days or more. The process simply sends an email to the user assigned to the activity. The process does not update entity data, so the `processWorkItemMethod` does not use the `runWithNewBundle` API.

Note: In general, Guidewire recommends that your custom work queue class extend `BulkInserWorkQueueBase` rather than `WorkQueueBase`. The following example is for illustration only.

```

uses gw.api.database.Query
uses gw.api.database.Relop
uses gw.api.email.EmailUtil
uses gw.processes.WorkQueueBase
uses java.util.Iterator

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyActivityEmailWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom process by
     * passing the process type, the entity type the work queue item, and
     * the target type for the units of work.
     */

    construct() {
        super ( BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity )
    }

    /**
     * Select the units of work for a batch run: activities that have not been
     * viewed for five days or more.
     */

    override function findTargets() :Iterator <Activity> {
        // Query the target type and apply conditions: activities not viewed for 5 days or more
        var targetsQuery = Query.make(Activity)
        targetsQuery.compare( Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
            java.util.Date.Today.addBusinessDays(-5) )
        return targetsQuery.select().iterator()
    }

    /**
     * Process a unit of work: an activity not viewed for five days or more
     */
    override function processWorkItem(WorkItem: StandardWorkItem) {
        // Extract an object referent to the unit of work: an Activity instance
        var activity = extractTarget(WorkItem) // Convert the ID of the target to an object reference
    }
}

```

```

    if (activity.AssignedByUser.Contact.EmailAddress1 != null) {
        // Send an email to the user assigned to the activity
        EmailUtil.sendEmailWithBody(null,
            activity.AssignedUser.Contact,
            null,
            "Activity not viewed for five days",
            "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".")
        //To:
        //From:
        //Subject:
        //Body:
    }
}
}

```

Notice that this work queue implementation uses a `findTargets` method to return an iterator of target objects on which to base the work items.

Example work queue for updating entities

The following Gosu code is an example of a custom work queue that updates entity data as part of processing a unit of work. It uses the `StandardWorkItem` entity type for its work queue table. Its unit of work is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date. The process updates entity data, so the `processWorkItemMethod` uses the `runWithNewBundle` API.

```

uses gw.api.database.Query
uses gw.api.database.Relop
uses gw.api.email.EmailUtil
uses gw.processes.WorkQueueBase
uses gw.transaction.Transaction
uses java.util.Iterator

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyActivityEscalationWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom process by
     * passing the process type, the entity type the work queue item, and
     * the target type for the units of work.
     */
    construct () {
        super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity)
    }

    /**
     * Select the units of work for a batch run: activities that have not been
     * viewed for five days or more.
     */
    override function findTargets (): Iterator <Activity> {
        // Query the target type and apply conditions: activities not viewed for 5 days or more
        var targetsQuery = Query.make(Activity)
        targetsQuery.compare(Activity#LastViewedDate, Relop.LessThanOrEquals,
            java.util.Date.Today.addBusinessDays(-5))
        return targetsQuery.select().iterator()
    }

    /**
     * Process a unit of work: an activity not viewed for five days or more
     */
    override function processWorkItem (WorkItem : StandardWorkItem) {
        // Extract an object referent to the unit of work: an Activity instance
        var activity = extractTarget(WorkItem)

        // Send an email to the user assigned to the activity.
        if (activity.AssignedUser.Contact.EmailAddress1 != null) {
            EmailUtil.sendEmailWithBody(null,
                activity.AssignedUser.Contact,
                null,
                "Activity not viewed for five days",
                "See activity " + activity.Subject + ", due on " + activity.TargetDate + ".")
            // To:
            // From:
            // Subject:
            // Body:
        }

        // Update the escalation date on the assigned activity
        Transaction.runWithNewBundle( \ bundle -> {
            activity = bundle.add(activity) // Add the activity to the new bundle
            activity.EscalationDate = java.util.Date.Today } ) // Update the escalation date
    }
}

```


Example work queue with a custom work item type

The following example creates a custom work queue that uses a custom work item type for its work queue table. The name of the custom work item type is `MyWorkItem`. The code provides an implementation of the `createWorkItem` method in addition to the `findTargets` method to add work items for a batch to the work queue.

The unit of work for the work queue is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date.

Define custom work item `MyWorkItem`

It is necessary to create a custom work item if using a custom work queue.

Procedure

1. Open Guidewire Studio.
2. In the Studio **Project** window, expand **configuration** > **config** > **Extensions** > **Entity**.
3. Right-click **Entity** and select **New** > **Entity**.
4. Enter the following information in the **Entity** dialog and click **OK**.

Entity	MyWorkItem
Entity Type	Entity
Description	Custom work item
Type	keyable

Note: See “Considerations for developing a custom work queue” on page 335 for a discussion as to why Guidewire recommends the use of `softentityreference` instead of `foreignkey` here.

For all other fields, accept that dialog defaults.

5. Using the add functionality at the top of the left-hand column (the plus sign), add the following subelements to the `MyWorkItem` entity.

Element	Element attribute
column	<ul style="list-style-type: none"> • <code>name</code> – Activity • <code>type</code> – <code>softentityreference</code> • <code>nullok</code> – false
implementsEntity	<ul style="list-style-type: none"> • <code>name</code> – Workitem

What to do next

To use this custom work item, you need to implement custom work queue `MyActivitySetEscalationWorkQueue`.

Implement `MyActivitySetEscalationWorkQueue`

To create a custom work queue, implement a class that extends class `WorkQueueBase`.

Before you begin

This example requires that you first create custom work item `MyWorkItem`.

Procedure

1. Open Guidewire Studio

2. In the Studio **Project** window, expand **configuration > gsrc**.
3. Create a new package directory named **workflow**.
4. Within the **workflow** folder, create a new Gosu class named **MyActivitySetEscalationWorkQueue**.
5. Populate this class with the following code.

```
package workflow

uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.Relop
uses java.util.Iterator
uses gw.transaction.Transaction
uses gw.api.email.EmailUtil
uses gw.pl.persistence.core.Bundle

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyActivitySetEscalationWorkQueue extends WorkQueueBase <Activity, MyWorkItem> {

    /**
     * Let the base class register this type of custom process by
     * passing the process type, the entity type the work queue item, and
     * the target type for the units of work.
     */
    construct () {
        super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, MyWorkItem, Activity)
    }

    /**
     * Select the units of work for a batch run: activities that have not been
     * viewed for five days or more.
     */
    override function findTargets (): Iterator <Activity> {
        // Query the target type and apply conditions: activities not viewed for 5 days or more
        var targetsQuery = Query.make(Activity)
        targetsQuery.compare(Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
            java.util.Date.Today.addBusinessDays(-5))
        return targetsQuery.select().iterator()
    }

    /**
     * Write a custom work item
     */
    override function createWorkItem (activity : Activity, safeBundle : Bundle) : MyWorkItem {
        var customWorkItem = new MyWorkItem(safeBundle)
        customWorkItem.Activity = activity
        return customWorkItem
    }

    /**
     * Process a unit of work: an activity not viewed for five days or more
     */
    override function processWorkItem (workItem : MyWorkItem): void {
        // Get an object reference to the activity
        var activity = workItem.Activity

        // Send an email to the user assigned to the activity
        if (activity.AssignedUser.Contact.EmailAddress1 != null) {
            EmailUtil.sendEmailWithBody(null,
                activity.AssignedUser.Contact, // To:
                null, // From:
                "Activity not viewed for five days", // Subject:
                "See activity " + activity.Subject + ", due on " + // Body:
                    activity.TargetDate + ".")

        }

        // Update the escalation date on the assigned activity
        Transaction.runWithNewBundle( \ bundle -> {
            activity = bundle.add(activity) // add the activity to the new bundle
            activity.EscalationDate = java.util.Date.Today } ) // update the escalation date

        return
    }
}
```