Creatio Academy

Objects business logic

Business logic of objects

Version 8.0

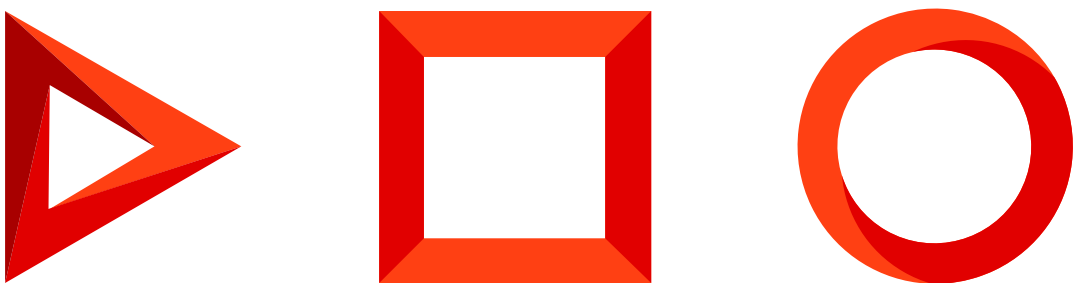

Table of Contents

Business logic of objects

Advanced

You can configure object events (saving, editing, deleting, etc.) in the following **ways**:

- In the Creatio UI using event subprocesses of the Object Designer.
- In the Creatio back-end using development tools.

Mechanism of the `Entity` event layer

The **purpose** of the `Entity` event layer is to configure object event handlers in the Creatio back-end using development tools. Creatio supports only those handlers that are defined in the main configuration and file content assemblies. Handlers from external assemblies are not supported.

Attention. Creatio triggers the mechanism of the `Entity` event layer after executing the object event subprocesses.

The event layer can handle the following **object events**:

- `OnDeleted` after a record is deleted.
- `OnInserted` after a record is added.
- `OnInserting` before a record is added.
- `OnDeleting` before a record is deleted.
- `OnSaved` after a record is saved.
- `OnSaving` before a record is saved.
- `OnUpdated` after a record is updated.
- `OnUpdating` before a record is updated.

The following **components** implement the mechanism of the `Entity` event layer:

- `BaseEntityEventListener`. Provides the handler methods of entity events.
- `EntityAfterEventArgs`. Provides the properties with the arguments of the handler method that is executed after the event triggers.
- The `EntityBeforeEventArgs` class. Provides the properties with the arguments of the handler method that is executed before the event fires.
- The `EntityEventListener` attribute. Registers the listener.

`BaseEntityEventListener` class

The purpose of the `Terrasoft.Core.Entities.Events.BaseEntityEventListener` class is to provide the handler methods of various entity events. View the handler methods in the table below.

Handler methods of event entities

Handler method	Method description
`OnDeleted(object sender, EntityAfterEventArgs e)`	Handles events after a record is deleted.
`OnDeleting(object sender, EntityBeforeEventArgs e)`	Handles events before a record is deleted.
`OnInserted(object sender, EntityAfterEventArgs e)`	Handles events after a record is added.
`OnInserting(object sender, EntityBeforeEventArgs e)`	Handles events before a record is added.
`OnSaved(object sender, EntityAfterEventArgs e)`	Handles events after a record is saved.
`OnSaving(object sender, EntityBeforeEventArgs e)`	Handles events before a record is saved.
`OnUpdated(object sender, EntityAfterEventArgs e)`	Handles events after a record is updated.
`OnUpdating(object sender, EntityBeforeEventArgs e)`	Handles events before a record is updated.

The methods of the `BaseEntityEventListener` class have the following **parameters**:

- `sender`. The link to the object instance that generates the event.
- `e`. The event arguments. Can take on the `EntityAfterEventArgs` (after the event) or `EntityBeforeEventArgs` (before the event) value.

View the call sequence of the event handler methods in the table below.

Call sequence of the event handler methods

Create an object	Edit an object	Delete an object
`OnSaving()`	`OnSaving()`	`OnDeleting()`
`OnInserting()`	`OnUpdating()`	`OnDeleted()`
`OnInserted()`	`OnUpdated()`	
`OnSaved()`	`OnSaved()`	

Event handlers retrieve the instance of `UserConnection` from the `sender` parameter. View an example that retrieves `UserConnection` below.

Example that retrieves `UserConnection`

```
[EntityEventListener(SchemaName = "Activity")]
public class ActivityEntityEventListener : BaseEntityEventListener
{
    public override void OnSaved(object sender, EntityAfterEventArgs e) {
        base.OnSaved(sender, e);
        var entity = (Entity) sender;
        var userConnection = entity.UserConnection;
    }
}
```

`EntityAfterEventArgs` class

The **purpose** of the `Terrasoft.Core.Entities.EntityAfterEventArgs` class is to provide properties with the arguments of the handler method that is executed after the event fires.

The `EntityAfterEventArgs` class has the following **properties**:

- `ModifiedColumnValues`. A collection of the modified columns.
- `PrimaryColumnValue`. The record ID.

`EntityBeforeEventArgs` class

The **purpose** of the `Terrasoft.Core.Entities.EntityBeforeEventArgs` class is to provide properties with the arguments of the handler method that is executed before the event triggers.

The `EntityBeforeEventArgs` class has the following **properties**:

- `KeyValue`. The record ID.
- `IsCanceled`. Enables canceling the further event execution.
- `AdditionalCondition`. Enables providing additional description of the entity filter conditions before the action.

`EntityEventListener` attribute

The **purpose** of the `EntityEventListener` attribute is to register a listener. The listener can be linked to all objects (`IsGlobal = true`) or to a specific object (for example, `SchemaName = "Contact"`). You can tag one listener class with many attributes to define a custom set of listened entities.

Set up the object event handler

To **set up** the event handler for an object that inherits from the `Entity` class:

1. Create a class that inherits from the `BaseEntityEventListener` class.
2. Decorate the class with the `[EntityEventListener]` attribute and specify the name of the entity whose event subscription to execute.
3. Override the event handler method.

Example that overrides the event handler method

```
/* Event listener of the "Activity" entity. */
public class ActivityEntityEventListener : BaseEntityEventListener
{
[EntityEventListener(SchemaName = "Activity")]
    /* Override the handler of the entity save event. */
    public override void OnSaved(object sender, EntityAfterEventArgs e) {
        /* Call the parent implementation. */
        base.OnSaved(sender, e);
        /* Additional actions.
        ... */
    }
}
```

Asynchronous behavior in the `Entity` event layer

The additional business logic of an object is time-consuming and executed sequentially. This hurts Creatio's front-end performance, for example, when an entity is saved or edited. The **mechanism of asynchronous operation execution** based on the `Entity` event layer solves this problem.

The following **components** implement asynchronous behavior in the `Entity` event layer:

- The `IEntityEventAsyncExecutor` interface. Declares the method that executes operations asynchronously.
- The `IEntityEventAsyncOperation` interface. Declares the method that launches an asynchronous operation.
- The `EntityEventAsyncOperationArgs` class. Instances of the class serve as arguments to pass to an asynchronous operation.

`IEntityEventAsyncExecutor` interface

The **purpose** of the `Terrasoft.Core.Entities.AsyncOperations.Interfaces.IEntityEventAsyncExecutor` interface is to declare the method that executes operations asynchronously. `ExecuteAsync<TOperation>(object parameters)` is a typed method that launches an operation with parameters. `TOperation` is a configuration class that implements the `IEntityEventAsyncOperation` interface.

`IEntityEventAsyncOperation` interface

The **purpose** of the `Terrasoft.Core.Entities.AsyncOperations.Interfaces.IEntityEventAsyncOperation` interface is to declare the method that launches an asynchronous operation.
`Execute(UserConnection userConnection, EntityEventAsyncOperationArgs arguments)` is the launch method.

Attention. We do not recommend describing the change logic of the primary entity in the class that implements the `IEntityEventAsyncOperation` interface. This can lead to incorrect data creation. We also do not recommend executing lightweight operations (for example, calculating a field value) since creating a

separate thread can take more time than executing the operation itself.

`EntityEventAsyncOperationArgs` class

The **purpose** of the `Terrasoft.Core.Entities.AsyncOperations.EntityEventAsyncOperationArgs` class is to provide its instances as arguments to pass to an asynchronous operation.

The `EntityEventAsyncOperationArgs` class includes the following **properties**:

- `EntityId`. The record ID.
- `EntitySchemaName`. The name of the schema.
- `EntityColumnValues`. The glossary of current column values of an entity.
- `OldEntityColumnValues`. The glossary of old column values of an entity.

Implement asynchronous behavior in the event layer

To **implement asynchronous behavior in the event layer**:

1. Create a class that implements the `IEntityEventAsyncOperation` interface. Implement additional logic that can be run asynchronously in the class.

   View an example of a class that implements additional logic below.

   Example of a class that implements additional logic

   ```
   /* The class that implements asynchronous operation calls. */
   public class DoSomethingActivityAsyncOperation: IEntityEventAsyncOperation
   {
       /* The start method of the class. */
       public void Execute(UserConnection userConnection, EntityEventAsyncOperationArgs argument
           /* ... */
       }
   }
   ```

2. Use a class factory in the handler method of the activity object listener.

   The class factory serves the following **purposes**:

   - Retrieve the instance of the class that implements the `IEntityEventAsyncExecutor` interface.
   - Prepare the parameters.
   - Pass the class that implements additional logic for execution.

   View an example that calls an asynchronous operation in the event layer below.

   Example that calls an asynchronous operation in the event layer

   ```
   [EntityEventListener(SchemaName = "Activity")]
   ```

```
public class ActivityEntityEventListener : BaseEntityEventListener
{
    /* The handler method of the event after the entity is saved. */
    public override void OnSaved(object sender, EntityAfterEventArgs e) {
        base.OnSaved(sender, e);
        /* The instance of the class to execute asynchronously. */
        var asyncExecutor = ClassFactory.Get<IEntityEventAsyncExecutor>(
        new ConstructorArgument("userConnection", ((Entity)sender).UserConnection));
        /* The parameters to execute asynchronously. */
        var operationArgs = new EntityEventAsyncOperationArgs((Entity)sender, e);
        /* Asynchronous execution. */
        asyncExecutor.ExecuteAsync<DoSomethingActivityAsyncOperation>(operationArgs);
    }
}
```