

# **ORDA Events in 4D: Controlling Data Operations with Event-Driven Logic**

By Abir HSAINI, Technical Services Engineer, 4D Inc.

Technical Note 26-03

## Table of Contents

Table of Contents.....	2
Abstract .....	3
Introduction.....	3
Key Concepts .....	3
What Are ORDA Events?.....	3
Key Characteristics.....	3
Why Use ORDA Events .....	4
No Table Locking.....	4
Parallel Execution .....	4
One Place for All Data Rules.....	4
Seven Distinct Event Types .....	4
Types of ORDA Events.....	4
ORDA provides seven distinct event types that cover the complete entity lifecycle. Understanding when each event executes and what it can do is crucial for implementing effective business logic.....	4
Event Execution Summary.....	5
The touched Event - In-Memory Modifications .....	5
The validateSave Event - Pre-Save Validation .....	6
The Saving Event - During Save Operation.....	7
The afterSave Event - Post-Save Actions .....	7
The validateDrop Event - Pre-Delete Validation .....	8
The dropping Event - During Delete Operation .....	8
The afterDrop Event - Post-Delete Actions.....	9
ORDA Events vs Classic 4D Triggers .....	10
Definition and Activation.....	10
Database Events Available.....	10
Error Handling: .....	11
Example Classic Trigger .....	11
The New Way - ORDA Events.....	11
Definition .....	12
Key Advantages .....	12
Key Differences.....	12
Trigger Compatibility.....	13
How to Update from Triggers to ORDA Events .....	13
Step 1: Analyze Existing Triggers .....	14
Step 2: Create Entity Classes.....	15
Step 3: Migrate Trigger Logic to Events.....	15
1. Client_Trigger:.....	15
2. Product_Trigger.....	16
3. Order_Trigger:.....	17
Automatic Stock Management & Price Calculation.....	19
Conclusion .....	28

## Abstract

4D 21 introduces a new way to handle database operations using ORDA events. This approach replaces traditional triggers and offers better control over data operations. ORDA events run automatically when data is created, modified, saved, or deleted, allowing developers to add validation, error handling, and business logic directly in their code.

Unlike old triggers that lock entire tables, ORDA events work on individual records and can run in parallel, improving performance in multi-user environments. This document explains how to use ORDA events effectively and provides guidance for migrating from classic 4D triggers to this modern approach.

## Introduction

With **4D 21**, ORDA events provide a modern event-driven solution for handling database operations in an object-oriented manner, serving as a replacement for classic triggers. These events are implemented as data class functions and are automatically executed when records are created, updated, saved, or deleted, enabling precise control over data lifecycle actions.

Unlike classic triggers, which operate at the table level and can lock entire tables during execution, ORDA events work at the record level. This design enables multiple events to run in parallel on different records, improving multi-user performance. In addition, ORDA events centralize business logic within data classes, making validation rules and processing easier to maintain and more consistent.

ORDA automatically invokes these event functions in response to user actions or code operations, and they cannot be triggered manually. This guarantees consistent business logic across all access channels, including the user interface, REST APIs, and application code. With seven event types covering the full data lifecycle, ORDA events provide fine-grained control over save and drop operations and offer a robust, modern alternative to classic triggers.

## Key Concepts

### What Are ORDA Events?

ORDA events are special functions defined in Entity classes that execute automatically when specific operations occur on data. They represent a fundamental shift in how 4D handles data operations (moving from table-level triggers to entity-level events).

### Key Characteristics

- **Entity-Based Definition:** Events are always defined in Entity classes (e.g., ProductEntity, CustomerEntity).
- **Two Levels of Control :** Events can be defined at the entity level, applying to all attributes of an entity, or at the attribute level, applying only to a specific attribute (including computed attributes); when both are present, the attribute-level event is executed first, followed by the entity-level event.
- **Automatic Execution:** ORDA event functions cannot be triggered manually and are automatically executed by ORDA in response to user actions, code-based operations on entities, and CRUD operations (Create, Read, Update, Delete).

### Why Use ORDA Events

ORDA events introduce a new way of managing data in 4D applications, providing clear advantages over traditional mechanisms (Triggers).

#### No Table Locking

- Traditional triggers lock the entire underlying table during execution
- ORDA events operate at the entity (record) level only
- Multiple events can run in parallel as long as they involve distinct entities

#### Parallel Execution

- Several events can execute simultaneously on different records
- No waiting for table locks to release

#### One Place for All Data Rules

- All validation, transformation, and business rules reside in Entity classes
- Easier to find, test, and maintain code

#### Seven Distinct Event Types

- Precise control over **when** code executes in the entity lifecycle
- Different events for validation, execution, and post-processing

### Types of ORDA Events

ORDA provides seven distinct event types that cover the complete entity lifecycle. Understanding when each event executes and what it can do is crucial for implementing effective business logic.

## Event Execution Summary

Event	Triggered When	Level	Execution Location	Can Stop Action	Primary Use Case
touched	Value modified in memory	Entity/Attribute	Client/Server*	No	Format data, update related attributes
validateSave	Before save begins	Entity/Attribute	Server	Yes	Data validation, business rule checks
saving	During save operation	Entity/Attribute	Server	Yes	File creation, API calls, external operations
afterSave	After save completes	Entity only	Server	No	Notifications post-save cleanup
validateDrop	Before deletion begins	Entity/Attribute	Server	Yes	Permission checks, deletion validation
dropping	During deletion	Entity/Attribute	Server	Yes	Delete associated files, cleanup resources
afterDrop	After deletion completes	Entity only	Server	No	Audit notifications

### The touched Event - In-Memory Modifications

The touched event fires when an attribute value is modified **in memory**, before any saving operation. This is unique to ORDA and has no equivalent in classic 4D.

#### **Triggers include:**

- User typing in a 4D form field
- Code assignment using := operator (even self-assignment: \$entity.attr := \$entity.attr)
- Entity received on server from client
- REST API requests modifying data

#### **Syntax**

```
// Entity-level: triggered for any attribute change
Function event touched($event : Object)
    // code here

// Attribute-level: triggered only for specific attribute
Function event touched <attributeName>($event : Object)
    // code here
```

The touched event can execute on **client or server** depending on the local keyword:

### **With local keyword (Client-side execution)**

```
local Function event touched firstName($event : Object)
    // Executes on the client This.firstName := Uppercase(This.firstName)
```

### **Without local keyword (Server-side execution)**

```
Function event touched firstName($event : Object)
    // Executes on the server
```

**Important:** In REST, Qodly, or remote datastore scenarios, touched *always executes server-side*.

### **Characteristics**

- **Cannot stop action:** Returns no value, cannot prevent the modification
- **Useful for:** Data formatting, real-time validation, updating derived attributes
- **Also triggered by:** constructor() event and Data Explorer edits

### **The validateSave Event - Pre-Save Validation**

Before the entity is actually saved to disk. This is the opportunity to validate data consistency and enforce business rules.

### **Triggered by**

- entity.save()
- dataClass.fromCollection()

### **Syntax**

```
// Entity-level: validates entire entity
Function event validateSave($event : Object) : Object
    // Return error object to stop save
// Attribute-level: validates specific attribute (only if touched)
Function event validateSave <attributeName>($event : Object) : Object
    // Return error object to stop save
```

### Characteristics

- **Executes:** Server-side only
- **Can stop action:** Yes, by returning an error object
- **Attribute-level behavior:** NOT executed if the attribute wasn't touched
- **Execution order:** Attribute event first, then entity event.

### The Saving Event - During Save Operation

During the actual save operation, **while** the entity is being written to disk. This event runs after validateSave (if no error was raised).

#### Triggered by

- entity.save()
- dataClass.fromCollection()

#### Syntax

```
// Entity-level: runs for any save
Function event saving($event : Object) : Object
// Executes even if no attributes were touched

// Attribute-level: runs only if attribute was touched
Function event saving <attributeName>($event : Object) : Object
// Return error object to stop save
```

### Characteristics

- **Executes:** Server-side only
- **Can stop action:** Yes, by returning an error object
- **Use for:** Time-consuming operations, file creation, API calls
- **Error handling:** Should catch network errors, disk space issues, etc.

### The afterSave Event - Post-Save Actions

#### Triggered by

- entity.save()
- dataClass.fromCollection()

#### Syntax

```
// Entity-level only (no attribute-level version)
Function event afterSave($event : Object)
// Cannot return error object
// Cannot stop the action
```

### Characteristics

- **Executes:** Server-side only
- **Cannot stop action:** Save has already completed
- **Level:** Entity-level only
- **Not executed:** If no attributes were touched
- **Restriction:** Calling save() on This will raise an error (prevents infinite loops)

### When to Use

- Send confirmation emails after successful save
- Propagate changes to external systems.

### The validateDrop Event - Pre-Delete Validation

Before the entity is actually deleted from disk. This is chance to prevent unauthorized or inappropriate deletions.

#### Triggered by:

- entity.drop()
- entitySelection.drop()
- Database deletion control rules

### Syntax

```
// Entity-level: validates entire deletion
Function event validateDrop($event : Object) : Object
    // Return error object to prevent deletion

// Attribute-level: validates specific attribute
Function event validateDrop <attributeName>($event : Object) : Object
    // Return error object to prevent deletion
```

### Characteristics

- **Executes:** Server-side only
- **Can stop action:** Yes, by returning an error object
- **Use for:** Permission checks, status validation, business rule enforcement

### The dropping Event - During Delete Operation

During the actual deletion, **while** the entity is being removed from disk. Runs after validateDrop (if no error was raised).

#### Triggered by:

- entity.drop()
- entitySelection.drop()



- Database deletion control rules

### **Syntax**

```
// Entity-level
Function event dropping($event : Object) : Object
    // Return error object to stop deletion

// Attribute-level
Function event dropping <attributeName>($event : Object) : Object
    // Return error object to stop deletion
```

### **Characteristics**

- **Executes:** Server-side only
- **Can stop action:** Yes, by returning an error object
- **Use for:** Deleting associated files, cleaning up external resources, removing related data

### **The afterDrop Event - Post-Delete Actions**

Just after the entity is successfully deleted from disk.

#### **Triggered by:**

- entity.drop()
- entitySelection.drop()
- Database deletion control rules

### **Syntax**

```
// Entity-level only (no attribute-level version)
Function event afterDrop($event : Object)
    // Cannot return error object
    // Cannot stop the action
```

### **Characteristics**

- **Executes:** Server-side only
- **Cannot stop action:** Deletion has already completed
- **Level:** Entity-level only
- **Entity reference:** The dropped entity still exists in memory via This
- **Restriction:** Calling drop() on This will raise an error (prevents infinite loops)

### **When to Use**

- Send deletion notifications
- Log deletion to audit trail

- Trigger cleanup workflows
- Update related records in other tables
- Mark status for manual review if deletion failed

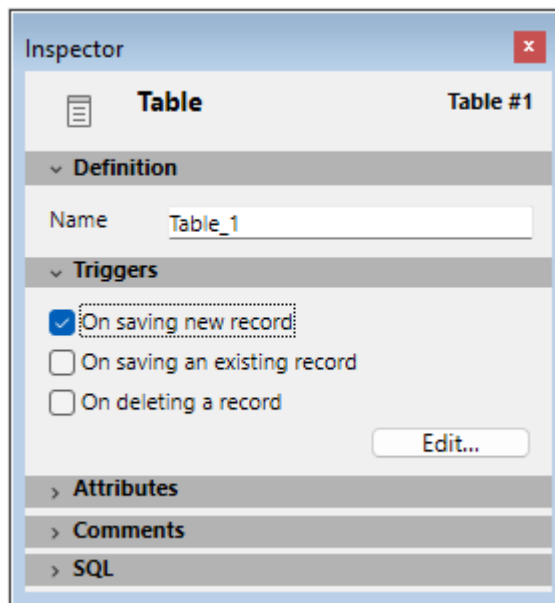
## ORDA Events vs Classic 4D Triggers

### The Classic Way - Database Triggers

Classic 4D triggers are methods attached to tables that execute automatically when database operations occur (create, update, delete). They have been the traditional way to enforce business rules and maintain data integrity in 4D applications.

#### Definition and Activation

- Triggers are created in the Structure editor for each table
- Must be explicitly activated in the table's Inspector window



- One trigger method per table handles all database events

#### Database Events Available

Case of

```
:(Database event = On Saving New Record Event)
// Handle new record save
```

```
:(Database event = On Saving Existing Record Event)
// Handle existing record update
```

```

:(Database event = On Deleting Record Event)
  // Handle record deletion
End case

```

Triggers execute at the database level (lowest level). They lock the entire table during execution. Only one trigger can run at a time per table.

### Error Handling:

```

// Return 0 if operation is allowed
// Return negative error code to prevent operation
$0 := -32001 // Custom error code

```

**Important:** Use custom error codes between **-32000** and **-15000**. Codes above **-15000** are reserved by the 4D database engine.

### Example Classic Trigger

```

// [Products] Table Trigger
C_LONGINT($0)
var $errorCode : Integer

$errorCode := 0

Case of
  :(Database event = On Saving New Record Event)
  :(Database event = On Saving Existing Record Event)
    // Validate margin
    If ([Products]margin < 50)
      $errorCode := -32001
      ALERT("Product margin must be at least 50%")
    End if

  :(Database event = On Deleting Record Event)
    // Check if product can be deleted
    If ([Products]status # "TO DELETE")
      $errorCode := -32002
      ALERT("Product must be marked TO DELETE")
    End if
End case

$0 := $errorCode

```

### The New Way - ORDA Events

ORDA events represent a modern, object-oriented approach to data control, operating at the **datastore level** (business logic layer) rather than the database level.

## Definition

- Events are functions defined in Entity classes
- Each event type has its own dedicated function
- Can be defined at entity level or attribute level
- Part of the ORDA abstraction layer

## Key Advantages

- **Entity-level locking:** Only the specific record is locked
- **Parallel execution:** Multiple events can run simultaneously
- **Granular control:** Seven distinct events for different stages
- **Object-oriented:** Business logic lives in Entity classes
- **Sophisticated error handling:** Return error objects with details

```
// ProductsEntity class

// Validation (attribute level)
Function event validateSave margin($event : Object) : Object
  If (This.margin < 50)
    return {
      errCode: 1001;
      message: "Margin below minimum 50%";
      seriousError: False
    }
  End if

// Deletion validation
Function event validateDrop($event : Object) : Object
  If (This.status # "TO DELETE")
    return {
      errCode: 3001;
      message: "Must be marked TO DELETE"
    }
  End if
```

## Key Differences

Aspect	Classic Triggers	ORDA Events
Operating Level	Database level (lowest level) - direct interaction with physical database	Datastore level (business logic layer) - part of ORDA abstraction

Aspect	Classic Triggers	ORDA Events
<b>Execution Model</b>	Lock entire table during execution Only one trigger per table at a time Sequential processing Poor multi-user performance	Lock only the specific entity (record) Multiple events run in parallel on different entities Concurrent processing Excellent multi-user performance
<b>Code Organization</b>	One method per table for all events Difficult to maintain All events in one Case of structure	Dedicated function for each event type Integrated within Entity classes Clean, modular organization Easy to locate specific logic
<b>Error Handling</b>	Return integer error code only Limited error information Standard alert dialogs	Return rich error objects Include code, message, and extra details Control error severity Better user experience
<b>Granularity</b>	Only 3 events: save new, save existing, delete Cannot target specific attributes Same logic for all attributes	7 events covering complete lifecycle Entity-level (all attributes) Attribute-level (specific attributes) Computed attributes included

### Trigger Compatibility

Classic 4D commands do not trigger ORDA events. They only execute classic triggers.

For example, when using SAVE RECORD, only the classic trigger is executed — ORDA events are not triggered.

Similarly, DELETE RECORD triggers only the classic trigger and bypasses ORDA events.

On the other hand, ORDA methods trigger classic triggers when such triggers exist.

For instance, entity.save() triggers ORDA events and also executes classic triggers (if they are defined).

Likewise, entity.drop() triggers ORDA events and classic triggers as well (if defined).

### How to Update from Triggers to ORDA Events

The migration from traditional 4D triggers to modern ORDA Entity Events is demonstrated through a real-world order management system. This example highlights common challenges developers face when using triggers and shows how ORDA Events effectively address them.

The guide uses a practical **Order Management System** as the foundation for all examples and comparisons. The system includes:

- **Client Management:** Customer data with automatic name formatting
- **Product Catalog:** Inventory with stock level monitoring
- **Order Processing:** Order creation with automatic pricing
- **Stock Management:** Real-time inventory updates with each transaction

### Prerequisites :

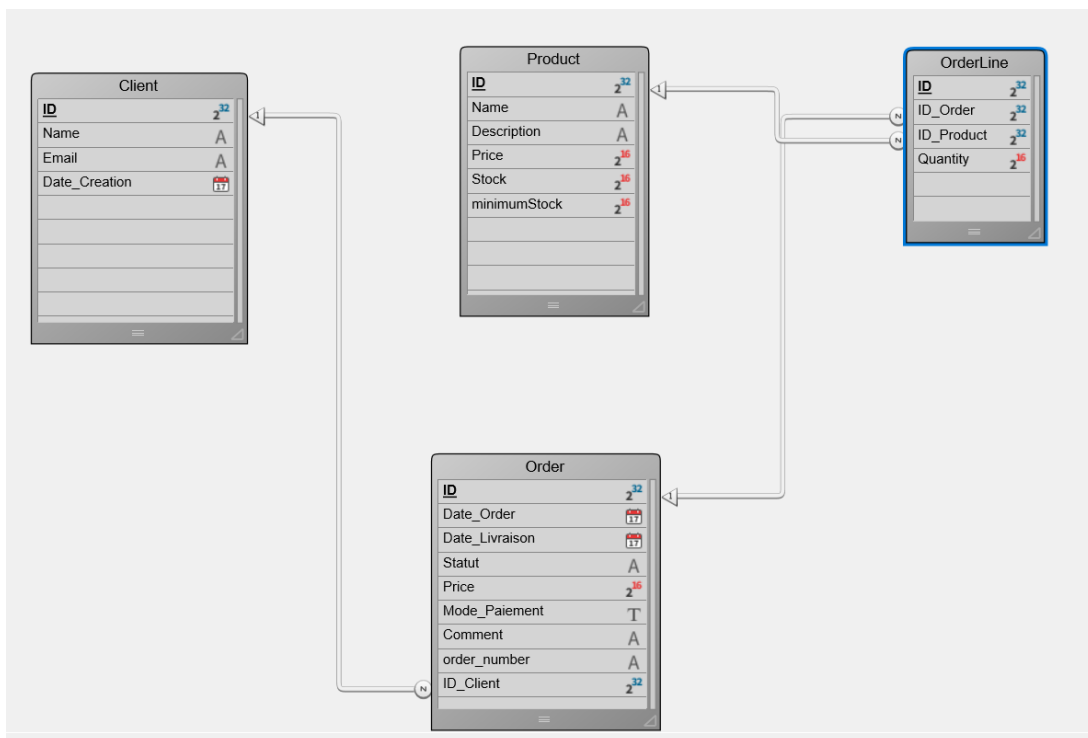
Before starting the migration, ensure :

- **4D Version:** V21 or higher
- **Project Mode:** For using classes
- **Backup:** Complete backup of the database and structure
- **Test Environment:** Separate environment for migration testing

### Step 1: Analyze Existing Triggers

Before writing any new code, thoroughly understand what the current triggers do. This analysis phase is critical for successful migration.

Below the structure:



Create a spreadsheet documenting each trigger in the database:

Table	Trigger Name	Events Used	Purpose
Client	Client_Trigger	On Saving New/Existing	Uppercase name
Product	Product_Trigger	On Saving New/Existing	Check stock Check_negative_Price
ORDER	ORDER_Trigger	On Saving Existing	Validate some attribute

## Step 2: Create Entity Classes

After analyzing the triggers, the next step is to create entity classes for each table that has business logic in triggers.

Each table in the database can have a corresponding entity class. The naming convention is: TableNameEntity. For example:

- Table [Client] → Class ClientEntity
- Table [Product] → Class ProductEntity
- Table [ORDER] → Class ORDEREntity
- Table [OrderLine] → Class OrderLineEntity

## Step 3: Migrate Trigger Logic to Events

Now that the entity classes are created, it's time to move the actual business logic from triggers into the appropriate ORDA Events

### 1. Client\_Trigger:

#### Original Trigger:

Case of

: (Trigger event=On Saving New Record Event) | (Trigger event=On Saving Existing Record Event)

[Client]Name:=**Uppercase**([Client]Name)

End case

The trigger **automatically converts client names to UPPERCASE** before saving.( When creating a new client or modifying an existing one)

#### Migrated to ORDA:

```
// ClientEntity
```

```
Class extends Entity
```

```
Function event touched Name($event : Object)
```

```
This.Name:=Uppercase(This.Name)
```

## 2. Product\_Trigger

### Original Trigger:

The trigger validates stock levels on save: it alerts when stock is low, prevents negative stock, and rejects the save operation with error code -16000.

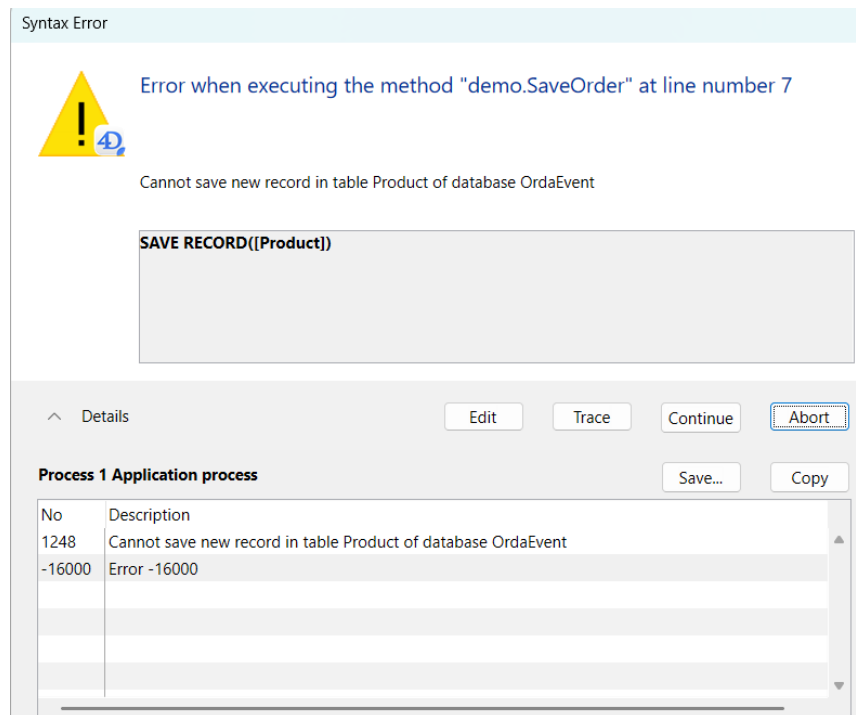
```
#DECLARE-> $result : Integer
```

Case of

```
: (Trigger event=On Saving New Record Event) | (Trigger event=On Saving Existing Record Event)
```

```
  If ([Product]Stock<=[Product]minimumStock)
    ALERT("Warning: The product "+[Product]Name+"is low in stock")
  End if
  If ([Product]Stock<0)
    $result:=-16000
    ALERT("⚠ Stock cannot be negative!")
  End if
```

End case



### Migrated to ORDA:

```
// =====
// CLASS 2: ProductEntity
// File: Project/Sources/Classes/ProductEntity.4dm
// =====
```

Class extends Entity



```

// Constructor: Initialization when creating new product
Function constructor()
    // Event: When Stock is modified
Function event touched Stock($event : Object)
    // 1. Prevent negative stock
    // 2. Check minimum stock level
    If (This.Stock<=This.minimumStock)
        var $message : Text
        If (This.Stock=0)
            // Out of stock - Critical
            $message:=" 🚫 OUT OF STOCK!\r\r"
            $message:=$message+"Product: "+This.Name+"\r"
            $message:=$message+"Stock: 0"
            BEEP
        Else
            // Low stock - Warning
            $message:=" ⚠️ LOW STOCK "
            $message:=$message+"Product: "+This.Name+" "
            $message:=$message+"Current stock: "+String(This.Stock)+" "
            $message:=$message+"Minimum stock: "+String(This.minimumStock)
        End if
        // Display alert (or create entry in Alerts table)
        ALERT($message)
    End if
Function event validateSave($event : Object)
    If (This.Stock<0)
        return {errorCode: 1001; MESSAGE: "The stock can not be negative"; extraDescription: {info: "The stock can not be negative"}}; seriousError: True}
    End if

```

### 3. Order\_Trigger:

#### Original Trigger:

The trigger enforces business rules and data integrity for orders on save:

- **Data normalization:** Uppercase order numbers
- **Default values:** Auto-fills status and date for new orders
- **Validation:** Prevents negative prices, warns about zero-price completed orders
- **Business logic:** Ensures orders marked as "Validate" or "Delivered" should have a price.

```

Case of
: (Trigger event=On Saving New Record Event)
  [ORDER]order_number:=Uppercase([ORDER]order_number)

  If ([ORDER]Statut="")
    [ORDER]Statut:="In progress"
  End if

```

```

If ([ORDER]Date_Order!=!00-00-00!)
    [ORDER]Date_Order:=Current date
End if

If ([ORDER]Price<0)
    $result:=-16000
    ALERT("⚠ The price can not be negatif !" + Char(Carriage return) + "Price has been reset to 0.")
End if
// Lors de la modification d'une commande existante
: (Trigger event=On Saving Existing Record Event)
// 1. Mettre le numéro en MAJUSCULES
[ORDER]order_number:=Uppercase([ORDER]order_number)

// 2. Empêcher prix négatif
If ([ORDER]Price<0)
    $result:=-16000
    ALERT("⚠ The price can not be negatif !" + Char(Carriage return) + "Price has been reset to 0.")
End if

// 3. Alerte si commande validée sans prix
If (([ORDER]Statut="Validate") | ([ORDER]Statut="Delivered"))
    If ([ORDER]Price=0)
        $result:=-17000
        ALERT("⚠ ATTENTION" + Char(Carriage return) + Char(Carriage
return) + "Order" + [ORDER]order_number + " Validate/Delivered" + Char(Carriage return) + "but the price is 0 €")
    End if
End if

End case

```

### Migrated to ORDA:

With this new feature, `_generateOrderNumber()` has also been added to handle cases where the order number is empty.

#### Class extends Entity

```

// Constructor: Initialize new order
Function constructor()

    // Event: When order_number is modified
Function event touched order_number($event : Object)
    // Convert to UPPERCASE
    This.order_number:=Uppercase(This.order_number)
    // Event: When Price is modified
Function event touched Price($event : Object)
    // Prevent negative price
    If (This.Price<0)
        This.Price:=0
    End if

```

```

// Event: Validation before save
Function event validateSave($event : Object) : Object
    var $status : Object
    // 1. Generate order number if empty
    If (This.order_number=Null) | (This.order_number="")
        This.order_number:=This._generateOrderNumber()
    End if
    // 2. Default status
    If (This.Statut=Null) | (This.Statut="")
        This.Statut:="In progress"
    End if
    // 3. Current date
    If (This.Date_Order=Null)
        This.Date_Order:=Current date
    End if
    // Check if order is validated/delivered with price = 0
    If ((This.Statut="Validate") | (This.Statut="Delivered"))
        If (This.Price=0)
            var $message : Text
            $message:="⚠ WARNING "
            $message:=$message+"Order "+This.order_number+" validated/delivered "
            $message:=$message+"but price is 0 €"
            $status:={errorCode: 1002; MESSAGE: "⚠ WARNING: Order "+This.order_number+"
validated/delivered\r but price is 0 €"; extraDescription: {info: "⚠ WARNING: Order "+This.order_number+"
validated/delivered\r but price is 0 €"}; seriousError: True}
        End if
    End if
    return $status

// Private method: Generate unique order number
Function _generateOrderNumber() : Text

    var $count : Integer
    var $orderNumber : Text

    // Count existing orders
    $count:=ds.Order.a//().length+1

    // Format: CMD-YYYY-XXXX
    $orderNumber:="CMD-"+String(Year of(Current date))+ "-" +String($count; "0000")

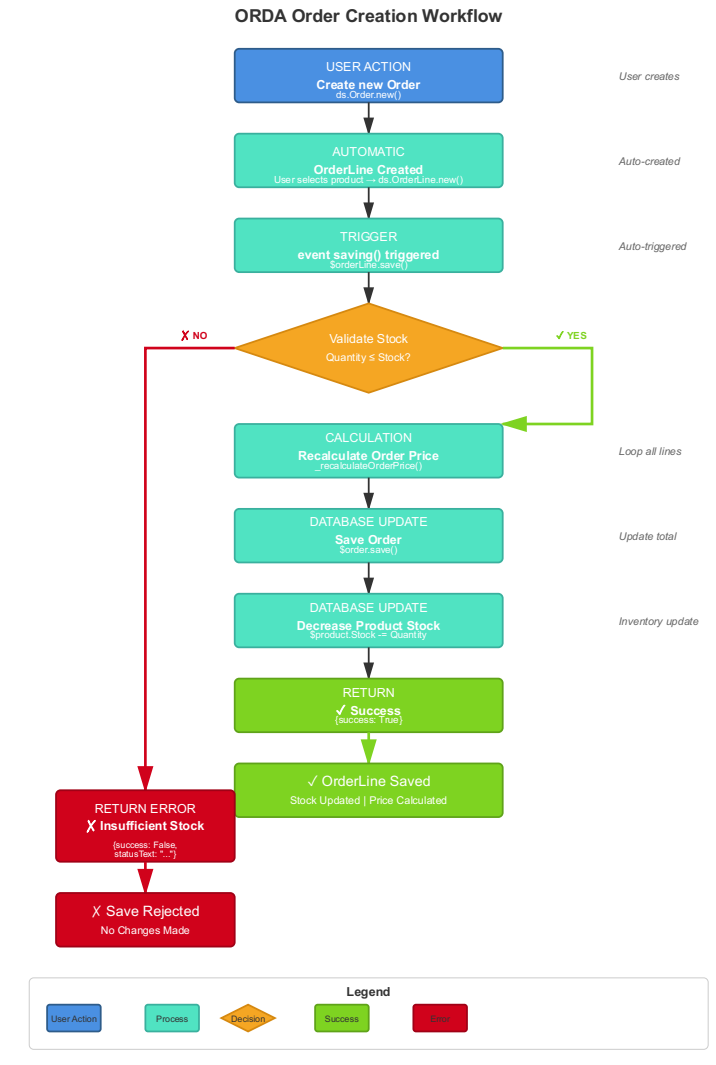
    return $orderNumber

```

## Automatic Stock Management & Price Calculation

With these ORDA events, stock management can be easily integrated and price calculation automated. The following approach is used to achieve this.

## Workflow:



## Code:

Class extends Entity

Function event saving(\$event : Object) : Object

```

var $status : Object
$status:=New object("success"; True)
If (This.Quantity>This.product.Stock)
    $result:={errCode: 1; message: "Insufficient stock "; \
    extraDescription: {info: "The product "+This.product.Name+" is
lower("+String(This.Quantity)+")"}; seriousError: False}
    return $result
End if
// Recalculate order total price
  
```

```
This._updateOrderPrice()  
return $status
```

```
// Private method: Update order price  
Function _updateOrderPrice()
```

```
    var $order : cs.OrderEntity  
    // If no automatic relation, do:  
    $order:=Form.subForm.currentItem  
  
    If ($order#Null)  
        This._recalculateOrderPrice($order)  
    End if
```

```
// Private method: Recalculate total price  
Function _recalculateOrderPrice($order : cs.OrderEntity)
```

```
    var $total : Real  
    var $orderLines : cs.OrderLineSelection  
    var $line : cs.OrderLineEntity  
    var $product : cs.ProductEntity  
  
    $total:=This.Quantity*This.product.Price  
  
    // Get all lines for this order  
    $orderLines:=ds.OrderLine.query("ID_Order = :1"; $order.ID)  
  
    // Calculate total  
    For each ($line; $orderLines)  
  
        // Get product  
        $product:=ds.Product.get($line.ID_Product)  
  
        If ($product#Null)  
            $total:=$total+($product.Price*$line.Quantity)  
        End if  
  
    End for each  
  
    // Update order price  
    $order.Price:=$total  
    $product:=ds.Product.get(This.ID_Product)  
    $product.Stock:=$product.Stock-This.Quantity  
    $product.save()  
    $order.save()
```

## Test:

Orders						Product			
Order Number	Date Order	Date Livraison	Client name	Price	Statut	Name	Minimum Stock	Price	Stock
CMD-2026-0001	3/11/2026	3/18/2026	TECH SOLUTIONS I...	\$ 9,680	In progress	dell xps 15 laptop	5	1299	20
CMD-2026-0002	3/8/2026	3/15/2026	SOPHIE MARTIN	\$ 3,046	Validate	hp probook 450	5	699	8
CMD-2026-0003	3/11/2026	3/25/2026	GLOBAL INDUSTRIE...	\$ 14,223	In progress	lenovo ideapad 3	10	449	30
CMD-2026-0004	3/6/2026	3/13/2026	INNOVATECH STAR...	\$ 4,819	Validate	hp z2 tower workstation	2	2499	8
CMD-2026-0005	3/11/2026	3/18/2026	SOPHIE MARTIN	\$ 587	In progress	apple macbook pro 14	3	1999	11

New Product Order Delete Order

Status

The interface is an order and stock management application, divided into two main panels:

- Left panel:  
A table listing orders with the following columns:
  - **Order Number:** order identifier (e.g. CMD-2026-0001)
  - **Date Order:** order date
  - **Date Livraison:** expected delivery date
  - **Client Name:** customer name
  - **Price:** total amount
  - **Statut:** order status, displayed with a color code

Two action buttons at the bottom:

- **New Product Order:** create a new order
- **Delete Order:** delete a selected order
- Right panel:  
A stock management table with the following columns:
  - **Name:** product name (laptops, workstations, etc.)
  - **Minimum Stock:** minimum stock threshold
  - **Price:** unit price
  - **Stock:** quantity available

An empty text area at the bottom right, likely intended to display messages or event logs related to actions performed in the app.

**Orders**

Order Number	Date Order	Date Livraison	Client name	Price	Statut
CMD-2026-0001	3/11/2026	3/18/2026	TECH SOLUTIONS L...	\$ 9,680	In progress
CMD-2026-0002	3/8/2026	3/15/2026	SOPHIE MARTIN	\$ 3,046	Validate
CMD-2026-0003	3/11/2026	3/25/2026	GLOBAL INDUSTRIE...	\$ 14,223	In progress
CMD-2026-0004	3/6/2026	3/13/2026	INNOVATECH STAR...	\$ 4,819	Validate
CMD-2026-0005	3/11/2026	3/18/2026	SOPHIE MARTIN	\$ 587	In progress

New Product Order Delete Order

Order number: Delivery date: 3/4/2026 Client: JOHN'S WORKSHOP Description: gaming

Date Order: Payment method: PayPal Status: Delivered

Order Items

Product Name	Unit Price	QTY	Total
lenovo ideapad 3	449	1	\$ 449
hp z2 tower works...	2499	1	\$ 2,499
apple macbook pr...	1999	1	\$ 1,999
			\$ 4,947

Add new product

Save

**Product**

Name	Minimum Stock	Price	Stock
dell xps 15 laptop	5	1299	20
hp probook 450	5	699	8
lenovo ideapad 3	10	449	30
hp z2 tower workstation	2	2499	8
apple macbook pro 14	3	1999	11

**Status**

When clicking **New Product Order**, a creation form appears below the table with the following fields:

- **Order Number:** automatically generated by the system, if not entered manually
- **Date Order:** also automatically generated, if not entered by the user
- **Delivery Date:** manually selected (e.g. 3/4/2026)
- **Client:** dropdown selector (e.g. JOHN'S WORKSHOP)
- **Description:** free text field (e.g. "gaming")
- **Payment Method:** dropdown (e.g. PayPal)
- **Status:** dropdown (e.g. Delivered)

A sub-table listing the products in the order:

- **Product Name, Unit Price, QTY, Total**
- A running **total** is calculated automatically at the bottom (e.g. \$4,947)
- An **"Add new product"** button allows adding items — **all products must be selected at once** in a single selection, rather than being added one by one.

### Orders

Order Number	Date Order	Date Livraison	Client name	Price	Status
CMD-2026-0002	3/8/2026	3/15/2026	SOPHIE MARTIN	\$ 3,046	Validate
CMD-2026-0003	3/11/2026	3/25/2026	GLOBAL INDUSTRIE...	\$ 14,223	In progress
CMD-2026-0004	3/6/2026	3/13/2026	INNOVATECH STAR...	\$ 4,819	Validate
CMD-2026-0005	3/11/2026	3/18/2026	SOPHIE MARTIN	\$ 587	In progress
CMD-2026-0006	3/11/2026	3/4/2026	JOHN'S WORKSHOP	\$ 4,947	Delivered

New Product Order
Delete Order

### Product

Name	Minimum Stock	Price	Stock
dell xps 15 laptop	5	1299	20
hp probook 450	5	699	8
lenovo ideapad 3	10	449	29
hp z2 tower workstation	2	2499	7
apple macbook pro 14	3	1999	10

### Status

```

{
  "success": true
}

```

After clicking **Save**, the interface returns to the main view with the following updates:

- The new order **CMD-2026-0006** (JOHN'S WORKSHOP, \$4,947, Delivered) is now added to the **Orders table**, confirming the order was successfully created.
- The **Stock levels** in the Product panel are automatically updated to reflect the purchased quantities — for example, lenovo ideapad 3 went from 30 → 29, hp z2 tower workstation from 8 → 7, and apple macbook pro 14 from 11 → 10.
- The **Status** section displays a JSON response `{ "success": true }`, confirming that the save operation was processed successfully by the system.









These four tests demonstrate the effectiveness of **ORDA events** in enforcing business rules at the data layer:

- **Test 1:** A valid order is saved successfully, with stock automatically updated.
- **Test 2:** An **error** blocks the save when the requested quantity exceeds available stock.
- **Test 3:** The save proceeds but a **warning** is raised when stock drops below the minimum threshold.
- **Test 4:** An **error** blocks the save when attempting to validate an empty order (total = \$0).

### Conclusion

ORDA events represent a significant evolution in how 4D developers handle database operations. By moving from table-level triggers to entity-level events, 4D has introduced a modern, object-oriented approach that addresses the limitations of classic triggers while providing powerful new capabilities.