



QAD Enterprise Applications  
Enterprise Edition

# Training Guide **QAD Customization**

70-3243-2015EE  
QAD 2015 Enterprise Edition  
April 2015

This document contains proprietary information that is protected by copyright and other intellectual property laws. No part of this document may be reproduced, translated, or modified without the prior written consent of QAD Inc. The information contained in this document is subject to change without notice.

QAD Inc. provides this material as is and makes no warranty of any kind, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. QAD Inc. shall not be liable for errors contained herein or for incidental or consequential damages (including lost profits) in connection with the furnishing, performance, or use of this material whether based on warranty, contract, or other legal theory.

QAD and MFG/PRO are registered trademarks of QAD Inc. The QAD logo is a trademark of QAD Inc.

Designations used by other companies to distinguish their products are often claimed as trademarks. In this document, the product names appear in initial capital or all capital letters. Contact the appropriate companies for more information regarding trademarks and registration.

Copyright ©2015 by QAD Inc.

Customization\_TG\_v2015EE.pdf/biw/mdf

**QAD Inc.**

100 Innovation Place  
Santa Barbara, California 93108  
Phone (805) 566-6000  
<http://www.qad.com>

# Contents

<b>QAD Customization Change Summary</b> .....	<b>vii</b>
<b>About This Course</b> .....	<b>1</b>
Course Description .....	2
Virtual Environment Information .....	2
QAD Web Resources .....	2
<b>Chapter 1 Course Overview</b> .....	<b>3</b>
Course Description .....	4
Objectives .....	5
Daily Schedule .....	6
Agenda .....	7
Your Instructor .....	8
Facilities .....	9
Questions .....	10
<b>Chapter 2 Non-intrusive Customization of Enterprise Financials</b> .	<b>11</b>
Non-Intrusive Customization: UI Design .....	12
UI Design Mode .....	13
Importing and Exporting UI .....	19
Exporting UI .....	20
Importing UI .....	21
Hands-on Exercise (1) .....	22
User-Defined Fields .....	23
Hands-on Exercise (2) .....	27
Non-intrusive Customization: Concepts .....	28
Concepts .....	29
Layered Application .....	30
Business Components .....	32
Proxy Implementations .....	33
Predefined Application Patterns .....	34
Business Fields .....	35
Business Activities .....	37

Object Datasets . . . . .	38
(BL) Component Instantiations . . . . .	39
Logical Transactions . . . . .	40
Session Component . . . . .	41
User-Defined Fields . . . . .	42
User-Defined Tables + Components . . . . .	43
Non-intrusive Customization: Architecture . . . . .	44
Introduction to QAD EE Architecture . . . . .	45
Application Layers . . . . .	48
Business Layer Component Structure . . . . .	52
Inheritance Structure . . . . .	53
Business Layer Component Structure - Terminology . . . . .	57
Application Patterns Overview . . . . .	59
Application Patterns Client . . . . .	60
Application Patterns Business Logic Flow . . . . .	65
Application Patterns Report Flow . . . . .	73
Data Handling - Datasets . . . . .	75
Data Handling - Data Flow In Common Patterns . . . . .	84
Data Handling - Data Flow in Patterns . . . . .	86
Data Handling - Data Handling in Object Datasets - New Object . . . . .	87
Data Handling - Data Flow in Patterns . . . . .	88
Data Handling - Data Handling in Object Datasets - Modify Object . . . . .	89
Data Handling - Data Flow in Patterns . . . . .	90
Data Handling - Data Handling in Object Datasets - Delete Objects . . . . .	91
Error Handling . . . . .	92
Non-intrusive Customization: Development . . . . .	96
Requirements for Customization . . . . .	97
BL Customizations . . . . .	102
Customization Diagram . . . . .	106
Customization - Development Steps . . . . .	108
BL Customizations . . . . .	112
Customization Exercise - Set Up . . . . .	114
BL Customizations . . . . .	115
BL Customizations - InitialValues . . . . .	124
Hands-on Exercise (3) . . . . .	126
BL Customizations - Calculate . . . . .	127
Hands-on Exercise (4) . . . . .	129
BL Customizations - ValidateComponent . . . . .	130
Hands-on Exercise (5) . . . . .	132
BL Customizations - Additional Updates . . . . .	133
BL Customizations - PreSave and PostSave . . . . .	135
BL Customizations - PostSave . . . . .	136
Hands-on Exercise (6) . . . . .	137

BL Customizations - GetBusinessFields	138
BL Customizations - Business Fields Mechanism	139
BL Customizations - GetBusinessFields	141
Hands-on Exercise (7)	142
BL Customizations - GetTranslation	143
Translations	145
Hands-on Exercise (8)	149
<b>Chapter 3 Non-intrusive Customization: Supporting Tools</b>	<b>151</b>
BL Customization - Tools	152
Launching HTML Documentation	153
Navigating HTML Documentation	155
Hands-on Exercise (9)	168
BL Customizations - Tools	169
BL Customizations - BL Code Tracing	170
Hands-on Exercise (10)	171
BL Customizations	172
Typical Customization: Case 1	184
Demo Customization: Case 1	185
Demo Customization: Case 1A	192
Hands-on Exercise (11)	193
Typical Customization: Case 2	194
Demo Customization: Case 2	195
Custom Tables	200
Custom Tables 2	201
Custom Tables 3	202
Demo Customization: Case 2 - DefineCustomRelations	203
Demo Customization: Case 2 - DataLoad	204
Demo Customization: Case 2 - PostSave	205
Demo Customization: Case 2 - Compile and Deploy	206
Demo Customization: Case 2 - Add Grid	207
Hands-on Exercise (12)	208
Typical Customization: Case 3	209
Demo Customization: Case 3 - Database Model	210
Demo Customization: Case 3 - Field Mapping	212
Custom Components	213
Demo Customization: Case 3 - Define User-Defined Component	215
Demo Customization: Case 3 - Define User-Defined Fields	216
Demo Customization: Case 3 - Define Menu Items	217
Demo Customization: Case 3 - Define Role Permissions	218
Demo Customization: Case 3 - UI Design Mode	219
Hands-on Exercise (13)	222
Report Customization	223

Report Framework for Financials Reports . . . . .	224
Report Customizations . . . . .	247
Report Customization - User-Defined Fields . . . . .	248
Report Customization - Debugging . . . . .	249
Hands-on Exercise (14 + 15 + 16) . . . . .	250
<b>Chapter 4 Integration with Enterprise Financials: Backend . . . . .</b>	<b>251</b>
Integrating with Financials Backend . . . . .	252
Calling the API . . . . .	253
UI Customization Method . . . . .	270
UI Customization - UI Side . . . . .	271
UI Customization - BLF Side . . . . .	272
UI Customization - UI Side . . . . .	273
<b>Product Information Resources . . . . .</b>	<b>275</b>

# QAD Customization Change Summary

The following table summarizes significant differences between this document and the last published version.

<b>Date/Version</b>	<b>Description</b>	<b>Reference</b>
April 2015/v2015 EE	Rebranded for QAD 2015 EE	---
April 2014/v2014 EE	Numerous editorial changes	---
	Revised Overview chapter	Page 3
	Removed Browse Customizations chapter	---
	Removed Customization Using UI Design Mode chapter	---
November 2013/2013.1 EE-Rev1	Numerous editorial changes	---
	Removed page from “Non-intrusive Customization of Enterprise Financials”	---
	Updated screen shots in “Non-intrusive Customization of Enterprise Financials”	Page 11
	Added Importing and Exporting UI	Page 19, Page 20, Page 21
	Updated screen shots in “Non-intrusive Customization: Supporting Tools”	Page 151
	Added additional Report Framework for Financials Reports page	Page 243
	Added UI Customization Method	Page 270, Page 271, Page 272, Page 273
October 2013/2013.1 EE	Rebranded for QAD 2013.1 EE	---
April 2013/2013 EE	Rebranded for QAD 2013 EE	---
	Numerous editorial changes	---
	Added PowerPoint slide notes to guide	---
October 2012/2012.1EE	Changed PostSave.After to DataSave.After	Page 224
August 2012/2012EE	Numerous editorial changes	---
November 2012/2010EE	Initial version. No changes.	---



# **About This Course**

## Course Description

This course is designed to teach Enterprise Edition customization concepts and practice. It consists of the following modules:

- Customization Using UI Design Mode
- Non-intrusive Customization of Enterprise Financials
- Non-intrusive Customization: Supporting Tools
- Integration with Enterprise Financials: Backend

### Course Objectives

By the end of this class, students will:

- Understand non-intrusive development
- Understand the impact on development
- Know how to perform customizations using QAD application features and functions
- Know how to customize component-based QAD applications

### Audience

This training is for developers/designers with Progress 4GL experience who want to learn non-intrusive customization techniques for component-based programs and functions (typically the new Financials functions).

### Prerequisites

Students are expected to have an intermediate understanding of Progress (OpenEdge 10) and a basic understanding of QAD Enterprise Edition.

### Virtual Environment Information

Use the hands-on exercises in this book with the QAD Enterprise Edition training environment that your instructor specifies.

### QAD Web Resources

From QAD's main site, you can access QAD's Learning or Support sites.

<http://www.qad.com/>

Chapter 1

# **Course Overview**

## Course Description

### Course Description

Welcome to QAD Enterprise Edition Customization training

- Course consists of 4.5 days of instruction combined with hands-on exercises
- Areas covered include:
  - Customization using UI Design Mode
  - Non-intrusive Customization of Enterprise Financials
  - Non-intrusive Customization: Supporting Tools
  - Integration with Enterprise Financials: Backend



CUST\_OVER\_030

## Objectives

### Objectives

After completion of this course, you should:

- Have an understanding of:
  - Non-intrusive development
  - The impact on development
- Be able to:
  - Perform customizations using QAD application features and functions
  - Customize component-based QAD applications



CUST\_OVER\_040

## Daily Schedule

### Daily Schedule

- Class begins 9:00
- Break 10:00
- Lunch 12:00 - 1:00
- Break 2:20
- Break 3:50
- Class ends 5:00 (last day ends earlier)



CUST\_OVER\_050

## Agenda

### Agenda

- Day 1:
  - Customizations using UI Design Mode
  - Non-intrusive Customization of Enterprise Financials - Introduction & Architecture
- Day 2:
  - Non-intrusive Customization of Enterprise Financials - Development
  - Non-intrusive Customization: Supporting Tools
- Day 3:
  - Non-intrusive Customization of Enterprise Financials – Development and Typical Customization Cases
- Day 4:
  - Non-intrusive Customization of Enterprise Financials – Typical Customization Cases
  - Report Customization
- Day 5:
  - Integration with Enterprise Financials: Backend
  - UI Customizations



CUST\_OVER\_060

## Your Instructor

### Your Instructor

- Experience before QAD
- QAD experience



CUST\_OVER\_070

## Facilities

### Facilities

- Restrooms
- Smoking areas
- Set cell phones to vibrate
- Other



CUST\_OVER\_080

## Questions

### Questions

Feel free to ask questions at any time



CUST\_OVER\_090

Chapter 2

# **Non-intrusive Customization of Enterprise Financials**


## Non-Intrusive Customization: UI Design



## UI Design Mode

### UI Design Mode

- Available in all Financials UIs
- Lets you change certain control properties:
  - Controls can be anything on the form, such as fields, buttons, labels, or tab controls
  - Properties can be visible, enabled, label, initial value, co-ordinates on the form
- Different levels of Customization: User / Role / System  
 Secured via Secured Items not on Menu (under `BControlProperty` in the tree view)


CUST\_UI\_050

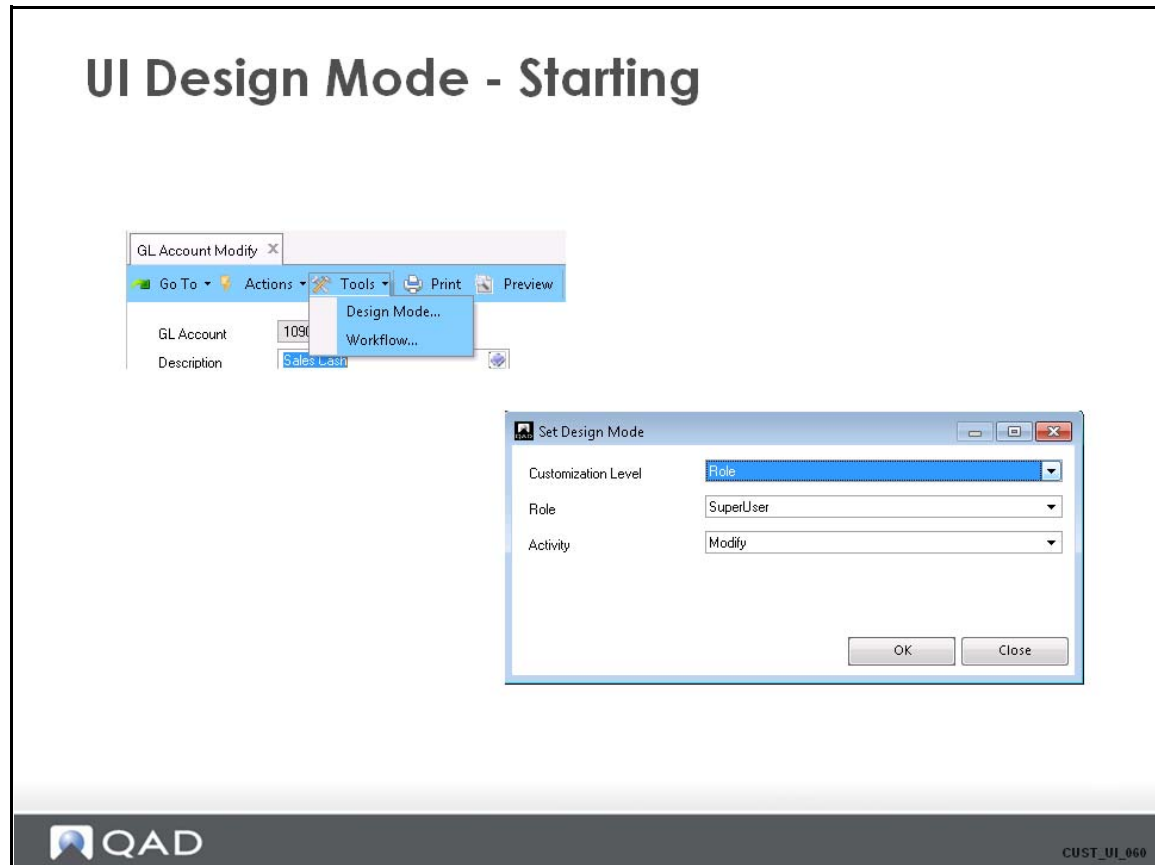
- QAD UI developers design all Financials forms. The forms are included in the application UI libraries.
- The system lets you apply changes to the standard layout/design. The system stores the applied changes (changed properties) at user/role/system level.
- UI Design Mode is a secured feature. The system administrator can enable/disable it for certain roles.
- The UI design changes are at the activity level. Even if the standard application used the same object form for all activities, the end user can change the form layout for each of the activities.

UI Design mode is typically used to:

- Add user-defined fields to the form
- Change the position of fields on the form
- Change labels
- Hide fields
- Assign initial values to fields

You can disable Design Mode in System Settings. If the customer does not intend to do any UI customization, you can disable the function for performance reasons.

## UI Design Mode

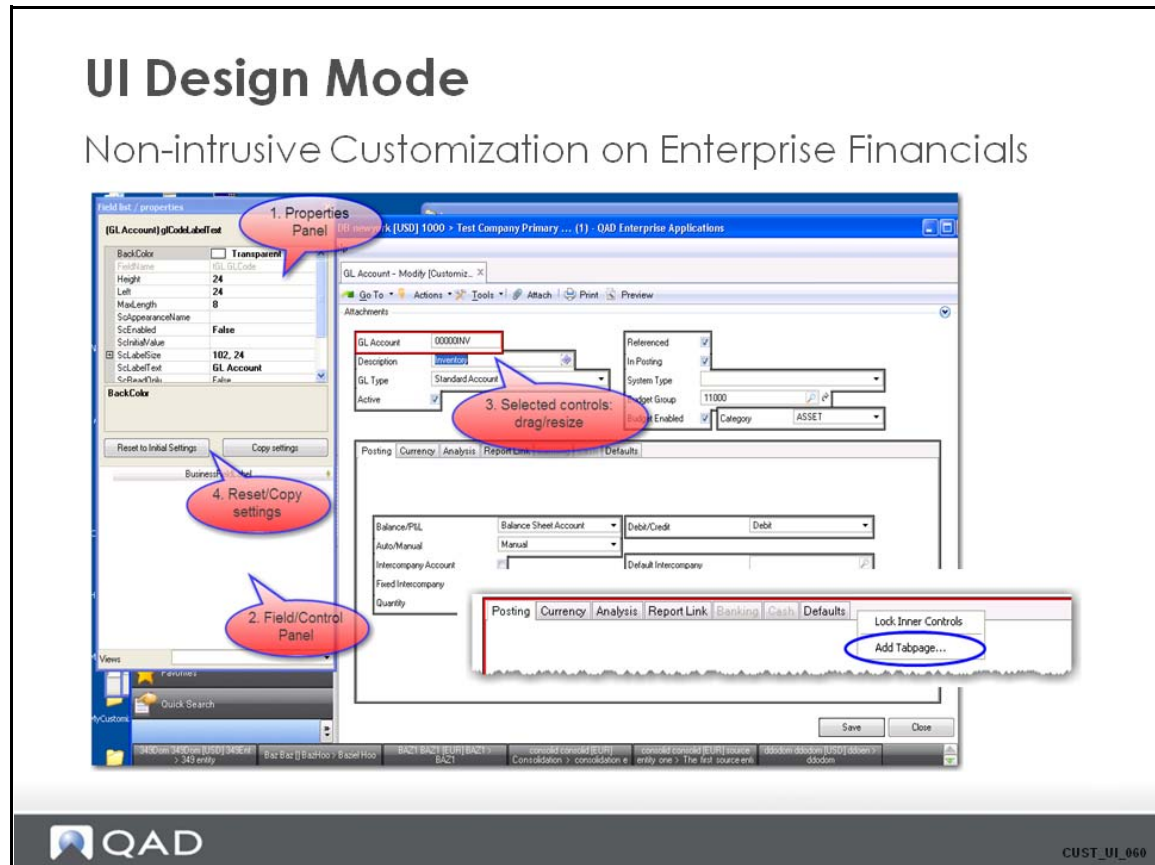


The UI Design Mode is the main way to change the applications UI. Typically, customization developers would remove fields, or make them read-only. They can also totally rearrange the form.

You can also use UI customization to put new fields, tab controls, and grids on the form. For example, you could add a custom table to an existing form.

The UI Design Mode is available on almost all forms in the fin application. Start it using “Tools->Design Mode...”

## UI Design Mode



1. You can use the Properties panel to manually change the available properties of the selected control (the control that is selected with a red rectangle on the screen). The name of the business field represented on the object form is displayed in the Field List, Properties window (as the FieldName property). In most cases, this value is auto-completed. If the field is not directly mapped to a field in the object dataset, the field name is not displayed. Also, the related business field name is retrieved using other methods or by consulting the HTML documentation.

2. The Field/Control panel contains the business fields and/or controls that are available, but not visible on the form. The user can drag the fields from the panel straight on the form in Design Mode.

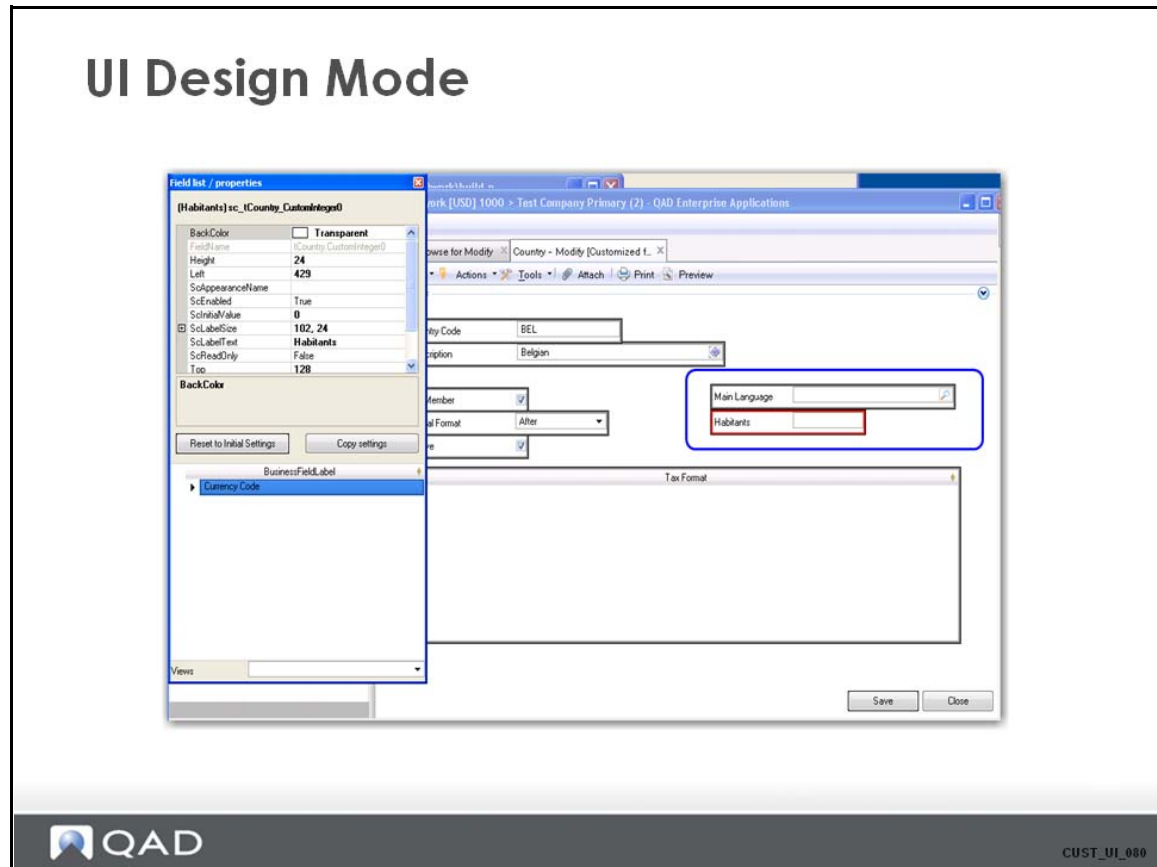
If the user wants to add a custom table on the form, the user can drag it to the form, which adds a grid to the form.

3. You can use the form in Design Mode to select a field or control. You can drag a control or resize it.

On the form design, the user can use the context menu to add a tabcontrol (max. 1 per form), and in an existing tabcontrol add tabpages. One can also add a text control this way.

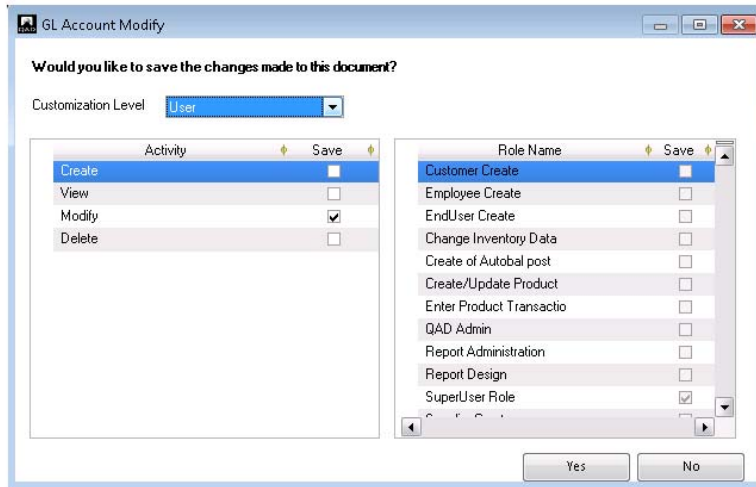
4. You can always go back to the factory defaults by using the "Reset to initial settings" button. The "Copy settings" button gives the user the ability to copy an existing UI customization (from some other user/role) for personal use. When this option is used, the selected customization overwrites the previous customization.

## UI Design Mode



UI Design Mode

# UI Design Mode - Closing



CUST\_UI\_080



## UI Design Mode

## UI Design Mode - Maintenance

- Definition and update via UI Design Mode
- Deletion via Customization Delete

Customization Delete x

Component Code

Component Label

Form

Level

Value

Delete	Layer	Business Component	Activity	Form
Nothing	USER (Plant Manager)	Business Relation	Create	BusinessRelationForm
Nothing	USER ()	Business Relation	Create	BusinessRelationForm
Nothing	USER (Demo User)	Business Relation	Create	BusinessRelationForm
Nothing	USER ()	Business Relation	Create	BusinessRelationForm
Nothing	USER (Plant Manager)	Business Relation	Excel Integration	BusinessRelationExcelForm
Nothing	USER (Plant Manager)	Business Relation	Modify	BusinessRelationForm
Nothing	USER (Demo User)	Business Relation	Modify	BusinessRelationForm
Nothing	USER (Robin Kronk)	Business Relation	Modify	BusinessRelationForm
Nothing	USER (Plant Manager)	Business Relation	View	BusinessRelationForm
Nothing	USER ()	Business Relation	View	BusinessRelationForm
Nothing	USER (Demo User)	Business Relation	View	BusinessRelationForm
Nothing	SYSTEM	Business Relation	View	BusinessRelationForm



CUST\_UI\_100

## Importing and Exporting UI

### Importing and Exporting UI

- Export in xml format
  - BControlProperty.xml
  - BReportVariant.xml
  - BCustomField.xml
  - BStoredSearch.xml



## Exporting UI

### Exporting UI

- Export UI customization option

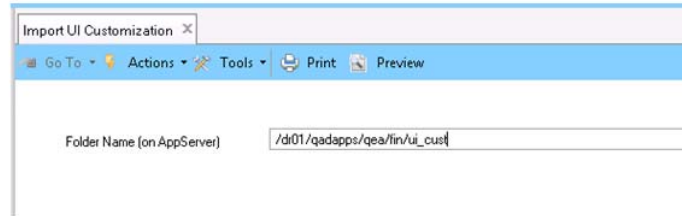
- API  
`ApiExportCustomization(...)`



## Importing UI

### Importing UI

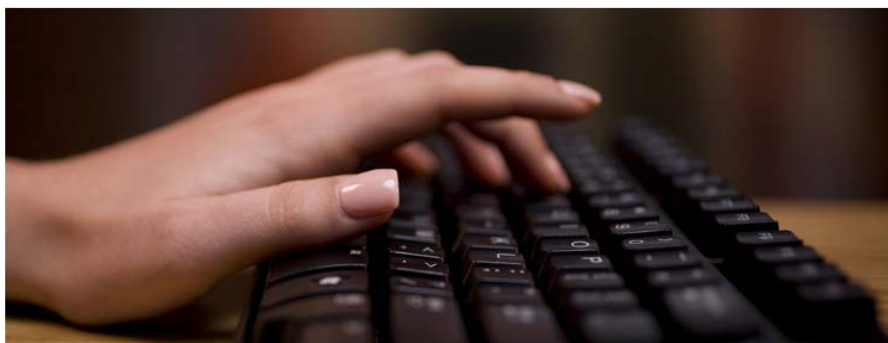
- Import UI customization option



- API  
`ApiImportCustomizations(icImportFile)`

## Hands-on Exercise (1)

# Hands-on Exercise (1)



UI Design

## User-Defined Fields

Many of the UI customization activities include the use of user-defined fields. The data available on the BL backend and the UI front end are different. Therefore, the datasets are the only way the data is passed between the layers. We try to use UDFs in many cases just to pass data.

Every component's object dataset can contain business fields to store customer-specific values (user-defined fields). They can be defined on one or more data tables in the object dataset. The user-defined fields are fields that map to physical fields in the database. The fields are known in the internal data flow within the application and in the application database itself. For the internal use of these user-defined fields, all normal patterns are followed. User-defined fields are passed between database, application server, and client in the same way as other object data.

Use this activity to specify a user-defined field. A user-defined field means, in effect, that the end user gives the field a meaning.

Specify the following information:

- **Business Component:** This is the label of the business component. For example "GL account" maps to the physical component name "BGI". Use the activity Business Component View to detect this mapping.
- **Field Name:** Select a type of physical field. Typically, the following types of custom fields are available: CustomShort0..9 (character type fields, for normal fields with a limited display length); CustomLong0..1 (character type fields, with a longer display length); CustomCombo0..9 (character type fields, with a possible values list attached);

CustomDecimal0..4 (decimal type fields); CustomInteger0..4 (Integer type fields); CustomDate0..4 (date type fields); CustomNote (character type field especially for fields represented as an editor)

- Description: Enter a logical name for the field to use in the application.
- Display: Modify the display length and decimal precision to change the display format.
- Lookup reference / Stored search / Return Field: Use these values to attach a lookup to the user-defined field.
- The Value List tab is only enabled for user-defined fields of type “CustomCombo”. This tab lets the user specify a distinct list of possible values.

### User-Defined Fields

The screenshot displays the 'User-Defined Field Create' window. At the top, there is a title bar and a menu bar with options: Go To, Actions, Tools, Print, Preview, and Attach. Below the menu, the 'Business Component' is set to 'Country', the 'Field Name' is 'tCountry.CustomShort0', and the 'Description' is 'Main language'. The 'General' tab is active, showing configuration for a 'Value List' field. The 'Side Label' is 'Main Language', the 'Column Label' is 'Lang', and the 'Display Format' is 'x(20)'. The 'Display Length' is set to 20, and the 'Decimal Precision' is 0. The 'Mandatory' checkbox is unchecked. The 'Lookup' type is 'Stored Search'. The 'Lookup Reference' is '/BLanguage.SelectLanguage', the 'Stored Search' is 'FACTORYDEFAULT', and the 'Return Field' is 'tLng.LngCode'. The QAD logo is in the bottom left, and 'CUST\_UI\_120' is in the bottom right.

## User-Defined Fields

### User-Defined Fields

Start using user-defined fields:

- On the UI:
  - Use UI Design Mode to add fields to the forms
  - Use the selection of columns in the browse grid to show it on the result
- On the BL: Use the physical field representing the user-defined field (like `tCreditor.CustomShort1`) in the NI Customization code
- On reports: Change the NI Customization code for the reporting component to give access to the UDF (see later in the training)



CUST\_UI\_140

## Hands-on Exercise (2)

### Hands-on Exercise (2)



**Define a User-defined Field for the  
Customer Application Object**

## Non-intrusive Customization: Concepts

Non-intrusive Customization: Concepts



## Concepts

### Concepts

- Layered application
- Business components (inheritance, encapsulation)
- Proxy implementations
- Predefined application patterns
- Business fields
- Business activities
- Object datasets
- Component instances
- Logical transaction
- Session component
- User defined fields – tables – components
- Related information / objects

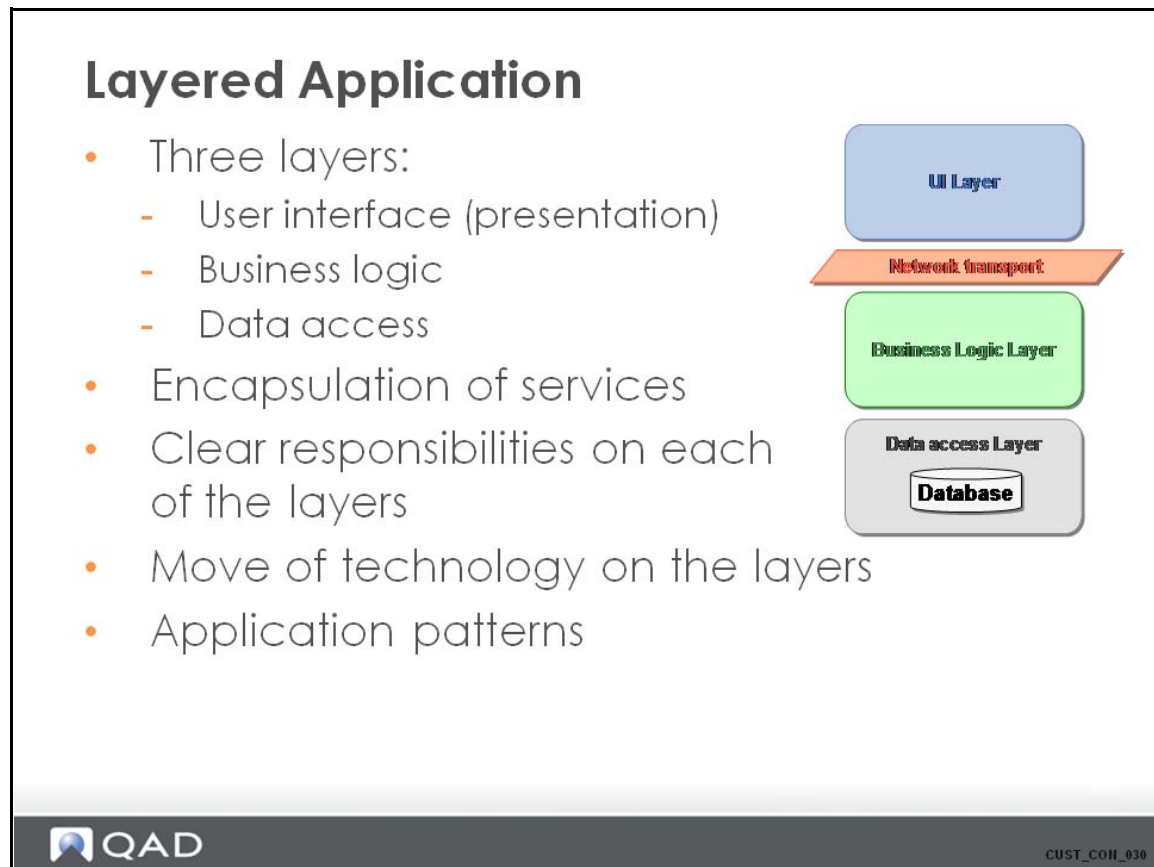


CUST\_COH\_020

This presentation covers the most important concepts for application development of the new Financials.

Most of the concepts are important to fully understand the application's built-in, non-intrusive customization capabilities.

## Layered Application



Encapsulation of services. Each layer offers a number of clearly defined services. These services have interfaces that describe how the services can be used.

- For example, on the UI layer, the AppShell offers a plugin architecture that allows for the calling/combining of services from other plugins through a predefined interface.
- For example, on the BL layer, specific components provide central functionality for the backend, like Session (for session management), Transaction (for logical transactions), Translations, and so on. These components have clearly defined interfaces that any external consumer can call (whether it is the UI, or another party that integrates with the application core services).
- For example, on the data access layer, the connection to the database, the possible queries, and updates are implemented in a component. This component has a clear interface. All business components in the system use it to get to the data. This brings a powerful level of abstraction to the application.

Responsibility of each layer:

- UI: The UI layer provides user interaction, user access to the data (shows the data on the screen), and system navigation. It also provides the application menu, integration of the application on the client/UI level, and so on.
- BL: Access to the data, calculation of data, validation of data, updates in related objects, query capabilities, meta information about objects, and so on.
- Data access: Data retrieval and data update.

Layering an application allows components to handle evolving technologies more easily and independently than a non-layered application. UI technology is moving much faster than business logic technology. This is why it is very important to use layers and for layers to always access services through predefined interfaces.

Besides separating services in layers with clear interfaces, it is important that the application functions follow a limited number of application patterns. This allows for easier customization and extension of standard functionality at a later stage.

## Business Components

# Business Components


Abstraction

- Encapsulation
- Inheritance

**Abstract**




Sphere




House plan

---

**Specific**




CUST\_COH\_040

Business components offer the ability to provide a level of abstraction in the business application. This enables you to develop the application using infrastructure code that supports some standard application patterns out of the box. Through this generic code, the application gets some typical characteristics as stability, easier maintenance of overall functionality, the ability to be non-intrusively customized, and so on. It also ensures that application services can be developed within the context of components.


Two mechanisms are mostly used to obtain the required level of abstraction. Through encapsulation, specific implementations and data are hidden from the callers. This simplifies the design of functionality. Inheritance gives you the ability to bring generalized interfaces to the application infrastructure. This allows different programs to use the same code.

The business components consist of methods, queries, and data items. All of these have a scope level. This way functionality and data can be hidden away for other components or applications.

## Proxy Implementations

### Proxy Implementations

- Surrogates for business components that provide access to it on the client (consumer) side.  
Client deployment of stubs representing business functions
- Proxies hide the complexity of connection infrastructure, session handling, and state handling for the client
- Proxies are generated and can be used in different types of technologies


CUST\_COH\_050

All business services are available on a central host in the system. These services are available through from a client through the network layer. In order to use the central business service, it is typically necessary to know:

- The interface of the business service.
- The connection string to the central host.
- The mechanism for connecting to the central host.
- Other information, including the implementation of data types on the client, such as datasets, XML representations, and so on.

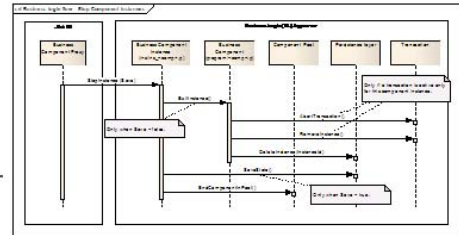
Proxies allow you to centrally develop a business service in one technology, and yet expose the interface for the business service in another technology (C#, Progress, XML, Java, ...). If you generate the proxies, and the connection method to the central business service layer is generic, you can hide the complexity of the connection for the client.

## Predefined Application Patterns

### Predefined Application Patterns

Provide predefined program flows on different levels:

- Examples high-level patterns:
  - Creation of an object
  - Modification of an object
  - Deletion of an object
  - Maintenance of object set
  - Selection of an object
- Examples of lower level patterns:
  - Instantiation of components
  - Error processing



CUST\_COH\_060

Patterns are crucial to new types of layered, component-based applications. The patterns drive the requirements for the generic coding in the infrastructure and give the guidelines for the design of the specific application components or classes.

Patterns also provide more consistency in the program code for the application, which makes maintenance easier because program code written by other developers is easier to understand.

The defined patterns in the application infrastructure determine a developer's ability to implement standard functions in the application.

## Business Fields

### Business Fields

- Abstraction of the information that is available for every object on the business layer:
  - Properties and information defined for the object, which typically can be used on the UI
  - Business field information provides information required by an external party representing the object
- Kinds of business fields:
  - Business fields representing physical database fields
  - Business fields representing calculated values which logically extend the object information
  - Related business fields representing fields that belong to other objects, which are related to the current object
- Values for business fields found in object dataset



CUST\_COH\_070

The typical properties of a business field are:

- Label
- Format
- Linked lookup
- Link with related object (goto)
- List of possible values (combo-box)
- Representation style

## Business Fields

**Business Fields**

Business fields on the UI representing information for the "customer" object

**Main object**  
Customer  
Accounting Info  
Bank Info

**Related object**  
Business Relation  
Ship-to address  
Head Office address  
Contact

Information from the business object "customer" is linked to controls on the UI via a generic mechanism

QAD CUST\_COI\_080

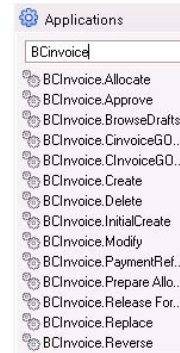
Typically, the client UI uses business field information to show the business information on the screen.

The business field information is language-independent. The current UI is written in C#.NET, but any other UI might use the business field information to format the screens.

## Business Activities

### Business Activities

- Represent the business functions of the application
- Defined on the level of the business component (for example, "Approve a supplier invoice") (*BCInvoice.Approve*)
- Mostly accessible from the application menu
- Used to secure the business functions in the application



CUST\_COH\_099

The business activities are defined on a higher level than the methods and the queries on components. Typically, the activities are exposed directly to end users. This is not the case for normal methods (even if they are public).

A business activity typically represents a certain program flow, and mostly implements one of the high-level application patterns (like "Journal Entry Create").


The list of business activities can be retrieved in a generic way for each component. The generic infrastructure code uses the business activity code to verify access for the user/consumer of the service against the role-based security in the application.

## Object Datasets

### Object Datasets

- The data associated with an object is passed between the different layers using object datasets
- Object datasets represented by:
  - OpenEdge ProDatasets on the BL side
  - .NET datasets on the UI side
- Object datasets are not only used by the UI client, but all types of consumers of the business services (integration, load functionality, event publishing...)

The diagram illustrates the flow of object data between three layers: the UI Layer (represented by a blue square), the Business Layer (represented by a green square), and the Data layer (represented by a blue cylinder). A central green arrow labeled 'OBJECT DATASET' points from the Business Layer to the Data layer. A green truck icon is positioned on this arrow. A second green arrow points from the UI Layer to the Business Layer, also featuring a green truck icon. This visualizes the role of object datasets as a 'transportation vehicle' for data between these layers.


CUST\_COH\_100

It is important to have a uniform way to pass object data between layers. As soon a business component is instantiated, and an object from the database is “loaded”, all relevant information is stored in the object dataset. The object dataset is the only way of passing the object state / information between the different layers.


Besides being used as transportation vehicle, the dataset also lends its structure to external parties wanting to integrate with the business application. For example, when the business application exposes objects through configured events, the data that is published has always the same format. The XML schema of the object dataset for the specific component defines the format. Also, when data is loaded, the business application backend requires the data to be in the official object dataset format.


For more information about object datasets, refer to the HTML documentation delivered in the installation package.

**(BL) Component Instantiations**

## (BL) Component Instantiations

- Instances of business components:
  - Alike instances of objects
  - Contain state (properties, data)
  - Instances are typically used for situations in which the method calls take part in a bigger logical transaction controlled by the external client
- Most methods need an instance to be started (or revived) before being called. Some methods do not need it, and encapsulate the start of the component instance and the required transaction within the method itself




CUST\_COH\_110

In a component-based application, the principle of “instantiation” must be very clear. Each business service consumer must instantiate a business component before being able to call a method on the business component. The consumer can be internal in the business layer, or it can be an external client (like the UI).

The instantiation process ensures that the necessary properties (data items) are initialized or loaded for the component. This allows the logic in the methods to execute properly.

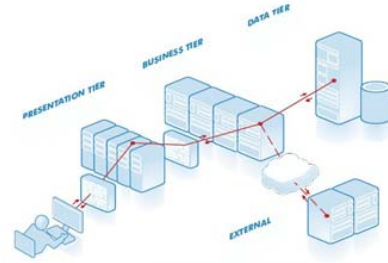
An instance ID identifies every component instance. This ID is a number generated at the moment of initial instantiation that can be used to identify the instance every time a consumer wants to communicate with the instance again.

For example, for the UI to create a supplier invoice, it first must start an instance for the supplier invoice before it can send data to it, validate it, and save it.

## Logical Transactions

### Logical Transactions

- Physical transactions do not work well with stateless applications. We must try to eliminate the risk of record locking by optimistic locking
- The transaction control can be encapsulated in the business logic level, but can also be on the level of the UI or an external consumer
- Component instances must be aware of logical transactions



CUST\_COH\_120

Physical transactions are de facto linked to one process, and require a stateful application model to guarantee that the same process executes the logic within a function. If you work with a multi-layer model, in which the business logic is used in a stateless way, the logical transaction is used to logically link different updates in business components. The logical transaction is living on the level of the database and is available every time the client uses a different process on the server.

Every time a component instance is created, the instance must know if it is “standalone” or if it takes part in a bigger transaction.

If instance is “standalone”, the logic itself is responsible for initiating and committing the transaction at the correct time. If the instance is part of a bigger transaction, it is assumed that the consumer (client) initiates the transaction and also decides when to commit the transaction.

For example, assume that the creation of a supplier invoice is started from the UI screen, but includes besides the supplier invoice a Financials posting. The instance for the supplier invoice and the posting belong to the same logical transaction.

## Session Component

### Session Component

- Each client needs a valid session to call functions on the business layer
- Central place to store session-specific data
- Session is always available when executing application logic
- A session is clearly linked to the application's authentication system



CUST\_COH\_130

A session component is instantiated at the moment a user logs on to the system. The instance stays available throughout the user session until the user logs off or until a time-out occurs. As long as the session stays active, some data (specific for the user or specific to the context in which the user is working) stays available as properties of the session instance.

A session instance is only valid if the user is authenticated and granted access to the application.

Because a valid session is required for all business logic execution, external consumers must provide the required authentication information to execute business functions.

## User-Defined Fields

### User-Defined Fields

- Easy extension of object datasets
- Predefined, with fixed name fields in the object datasets and in the application database
- User can give meaning to the fields using application functions or Customization code
- The system recognizes these fields and retrieves the extra properties for the fields when required
- Do not require any coding



CUST\_COH\_140

## User-Defined Tables + Components

### User-Defined Tables + Components

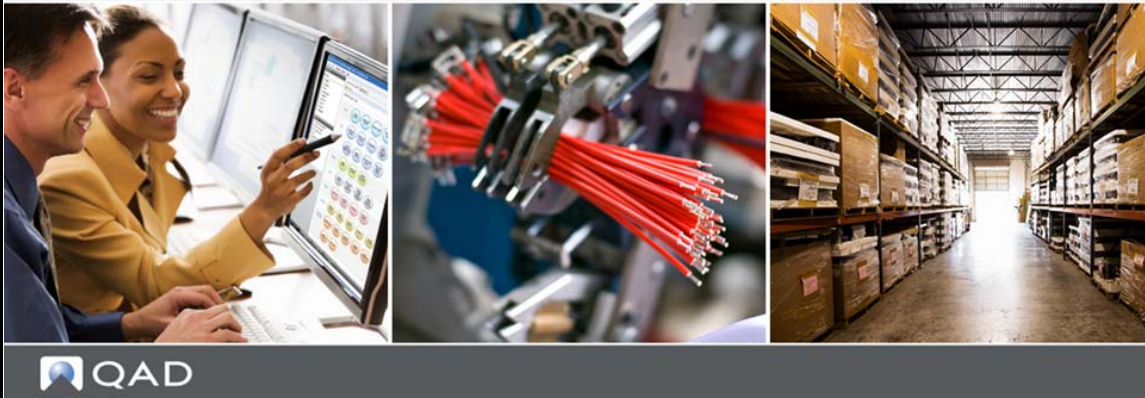
- More advanced extension of object datasets
- Only the interface is predefined, with three custom tables per business class
- Everything else must be coded



CUST\_COH\_150

## Non-intrusive Customization: Architecture

Non-intrusive Customization: Architecture



## Introduction to QAD EE Architecture

### Introduction to QAD EE Architecture

- QAD Reference Architecture: Multiple applications within one architecture
- Distinction between two major parts of the Global Application:
  - Legacy MFG/PRO: Client-server architecture, procedural code, pure Progress 4GL logic
  - Financials: Multi-tier architecture, event-driven, clear split between business and UI logic (.NET client for UI logic and 4GL business components for business logic)

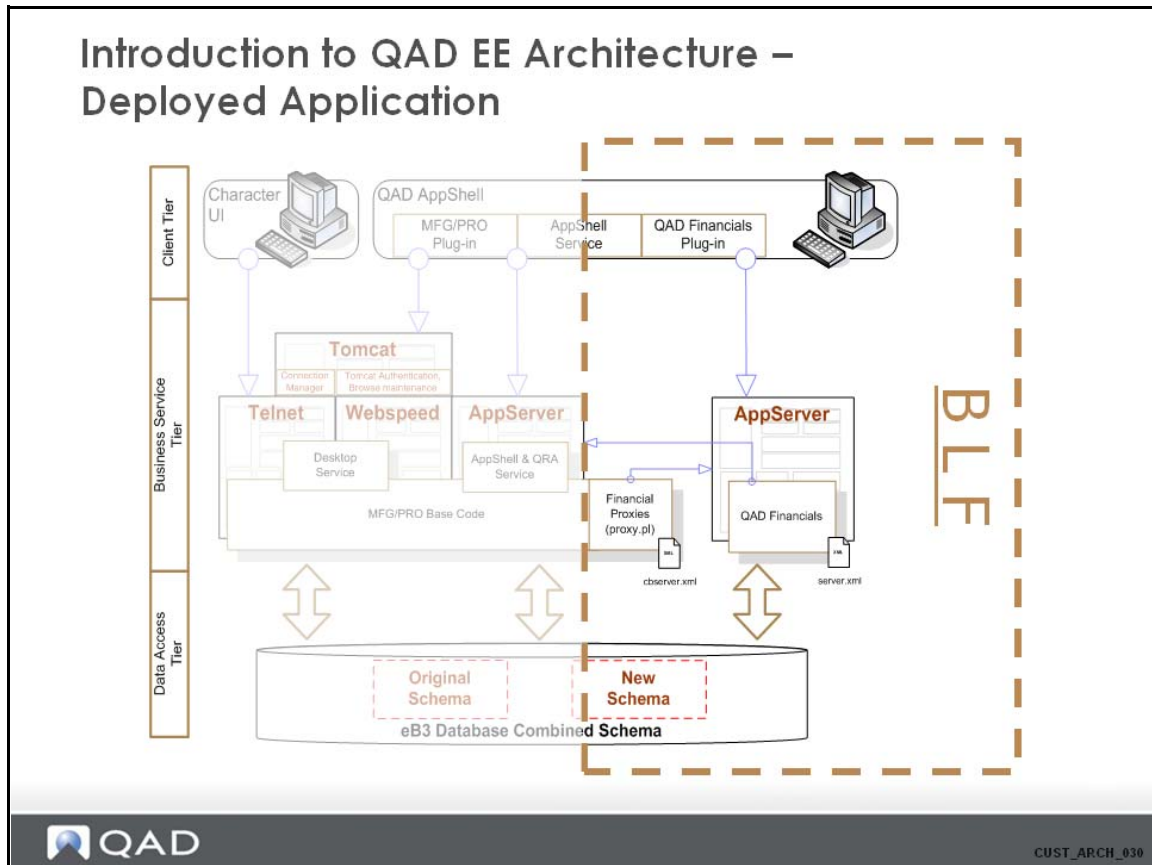


CUST\_ARCH\_020

Customization is different for standard MFG/PRO and Financials due to the different internal architecture of the two modules.

The Financials design is based on the need for a component model for the application. This model features code reuse, encapsulation, and clear interfaces.

## Introduction to QAD EE Architecture



This slide first gives a global, high-level overview of the architecture for the QAD EE application.

The general rule of the architecture is a clear separation between UI / BL / Data access for the future.

Most important in the scope of this training is the right-hand side, with a further explanation about the BLF implementation of the architecture.

Internal integration of the different pieces in the application is based on the proxy mechanisms.

Proxy mechanisms rely on the deployment of pieces of code on the consumer side. These pieces represent the business logic and are basically gateways, or wrappers, for the real business logic running on another server. Proxy code typically exposes the API interface of a component, but hides the logic for connecting to the backend AppServer.

## Introduction to QAD EE Architecture

## Introduction to QAD EE Architecture

- MFG/PRO Development:
  - Using standard Progress tooling
  - No specific client .NET development. Rendering in .NET client
  - Development according to a set of rules and guidelines
- Financials Development:
  - Using Component Builder for business logic and data access logic
  - Using Visual Studio .NET for UI logic
  - All development based on foundation infrastructure. All classes inherit from foundation classes



CUST\_ARCH\_040

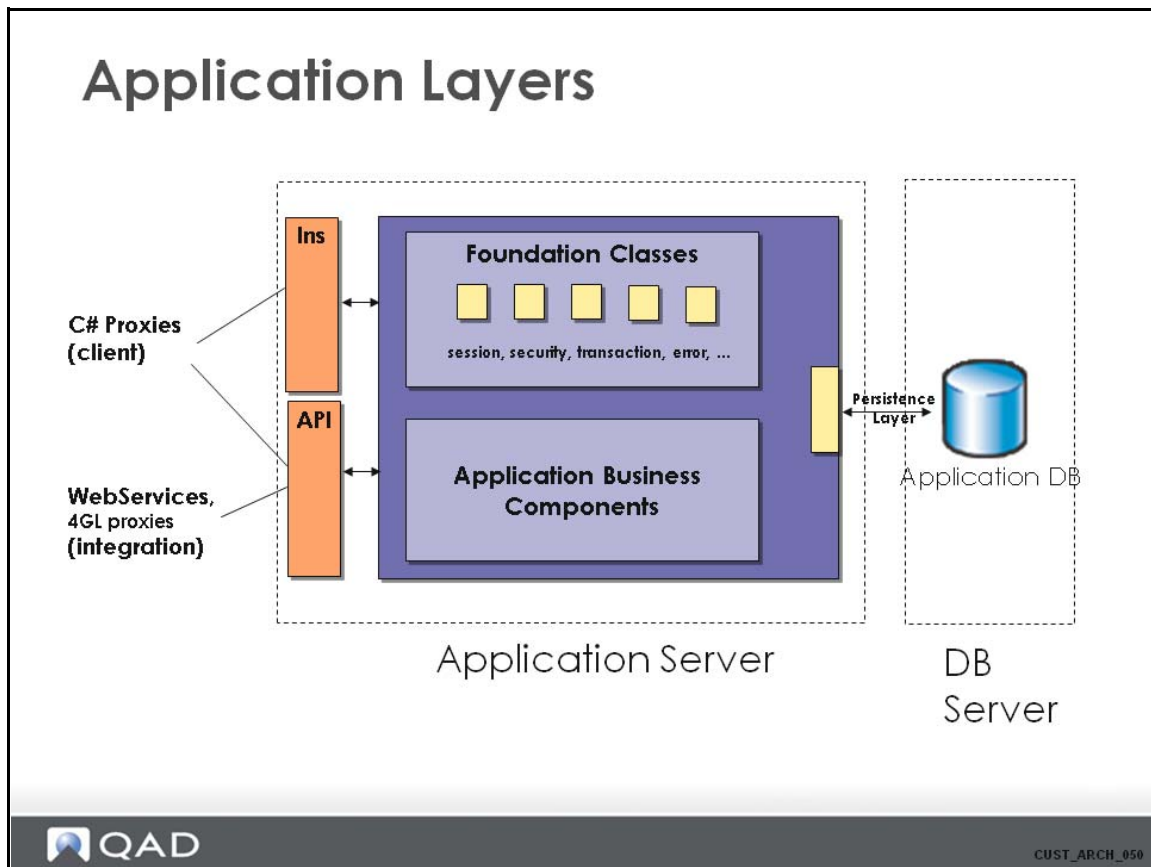
MFG/PRO code is written in the traditional style. This means that there is basically one layer containing a mixture of business, UI, and database access logic. This is the typical approach for classical, procedural-written, client-server code. The .NET UI is rendered and does not contain real application code on the .NET client side.

Due to this, MFG/PRO customization is mostly intrusive and done in the main .p program for the function (except for non-intrusive customization done via ICT).

Financials are written for a component model, which is more event-driven. Business logic is developed in Progress and UI code is developed in C#.NET. No behavioral code for the UI is written in Progress. The meta data about information for objects is retrieved from the business logic and is not duplicated on the UI side.

There is a fundamental difference in data handling between the two models. Because MFG/PRO is a client-server application using direct database access, data shown on the screen is retrieved directly from the database. Updates are immediate, with large physical transactions and locking of records within the transaction. For Financials, data manipulation is always on a dataset, which contains a copy of the data in the database. The object datasets consist of temp tables. Updating an object in Modify mode does not imply an active physical transaction. It uses an optimistic lock approach when updating data in the database.

## Application Layers



This slide shows the final result when the Enterprise Financials application is deployed.

We should explain how components/functions are delivered as part of the foundation and how functions are delivered as part of the application.

We should briefly mention the mechanism of the INS and API interfaces. It is important, however, to also note that methods called through the INS require state in the form of an existing session instance and/or an existing transaction instance. Therefore, updates can be part of a larger transaction, and methods called through the API do not require state, and are always stand-alone transactions.

This is described in more detail later in the presentation.

## Application Layers

### Application Layers

BL (Business Logic) Layer:

- Component model. Functionality implemented (encapsulated) in components. A business component typically offers functionality through methods and has responsibility over certain application data
- Business components have clear, documented interfaces
- The logic is accessible within the BL layer (by calls from other business components). Access is governed by the scope of the methods
- The logic is accessible via 4GL proxy, C# proxy or direct calls from another 4GL session
- Runs on Progress AppServer



CUST\_ARCH\_060

The real physical structure of the code for a business component is explained later in the presentation.

The term “Progress AppServer” means that the business logic is developed in Progress 4GL (ABL).

### Scope of Methods

It is important to understand the term “proxy”, which is central to the rest of this session.

The scope itself can be private/protected/public. This is the same as in any other OO language. When the method is public, it means that a method is available to other components within the business layer. A public method can also be marked as “API”, which means that the method is exposed to external consumers, and the proxy is generated and available for it. A public method can also be marked as “Remoting”, which means that C# proxy code is generated and available to call the method from within the C# client code.

A method or query can only be called after a component has been instantiated. See the Pattern section of this training.

## Application Layers

### Application Layers

#### Data Access Layer (Persistence Layer)

One foundation component making an abstraction of database access

- Database connections
- Read query
- Update
- Physical transaction control
- Optimistic lock checking
- Internal number sequences
- Audit support
- Persistent state



CUST\_ARCH\_070

Database connections are performed at runtime. Databases do not need to be connected at the startup of the AppServer sessions.

The following sections explain logical transactions and optimistic locking:

- For a typical classical transaction as used in MFG/PRO: As soon as a function is started, and data displays on the screen, a physical transaction is active. The records and data to be changed in the function are read with a find/for each with exclusive-lock (or share-lock). The updates are done directly in the database during the function. A transaction rollback is foreseen using the BI (before image) of the database. There is an implicit or explicit commit of the transaction controlled by the code blocks in the program.
- For a logical transaction as used in Financials: When a function is started, a logical transaction is started. The logical transaction is basically a component instance that controls other instances in which updates are prepared for writing to the database. The records and data to be changed during the function are read with a query no-lock from the database, and data is stored in temp tables datasets. No locking takes place.
- After the function makes the updates, the changed dataset is validated, and finally the logical transaction is asked to commit the changes to the database. It is only at this moment the real physical transaction takes place. An optimistic lock check occurs and the data is updated in the database.

Persistent state is explained later in this presentation.

## Application Layers

### Application Layers

Foundation for UI (User Interface) layer:

- .NET foundation libraries (AppServer.dll, BaseType.dll, BaseAdapters.dll, BaseForms.dll, Common.dll)
- Infrastructure for the following client-side mechanisms:
  - Integration in the AppShell
  - Client session, linked with back-end session (including security)
  - Exception handling
  - ...
- BLF-type of plug-in for the AppShell
- Prescribed way of developing C# classes for functions/activities
- The UI Layer uses C# proxy classes for the business components



CUST\_ARCH\_080

## Business Layer Component Structure

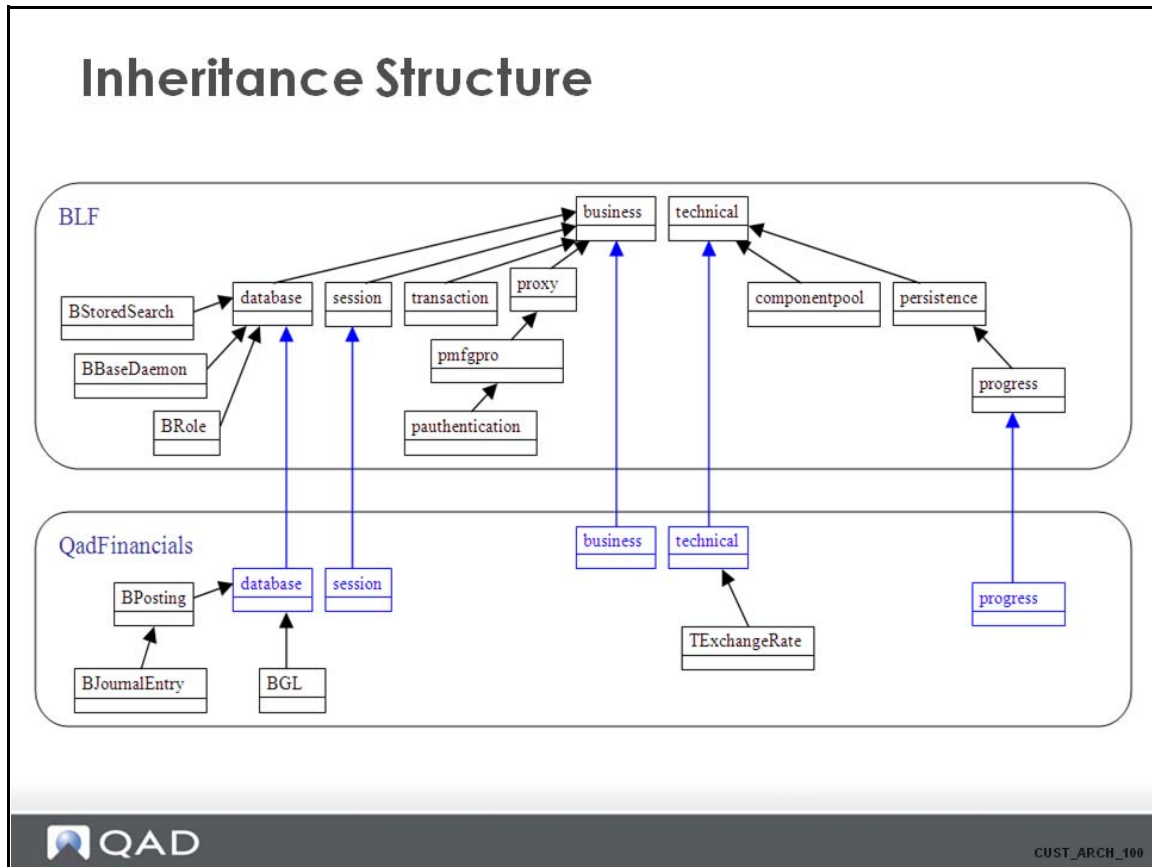
### Business Layer Component Structure

- Inheritance model (class diagram)
- Business / technical components



CUST\_ARCH\_090

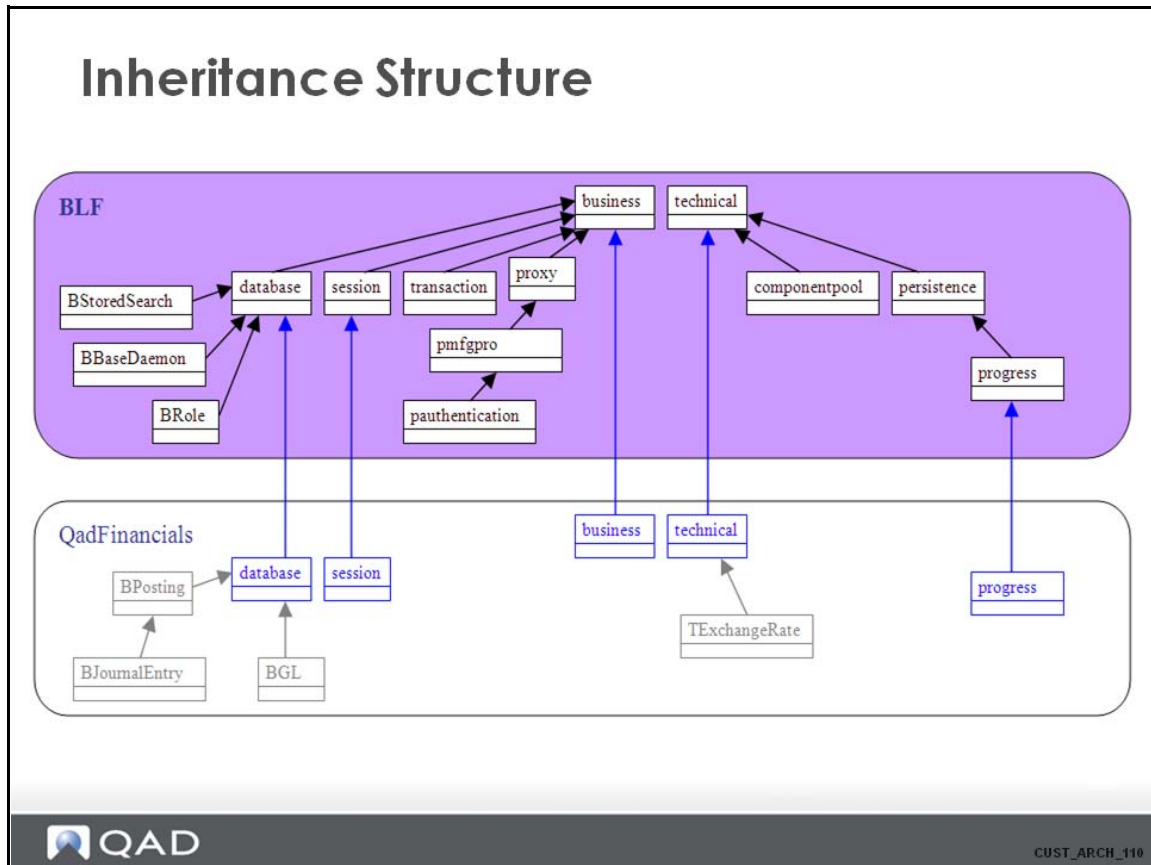
## Inheritance Structure



The application is built of components, which are maintained in different projects:

- **FoundationClasses:** This project contains the infrastructure foundation components.
- **BLF:** This project contains the application foundation components. These are components that are used to implement generic application functionalities, such as stored searches, or role-based security. This project also contains the technical infrastructure components. These are abstract components with generic logic supporting all mechanisms and patterns in the application, and non-abstract technical components with a very specific function in the infrastructure.
- **QadFinancials:** This project contains all application-level components. The business logic for all application functions is implemented in this project.

## Inheritance Structure

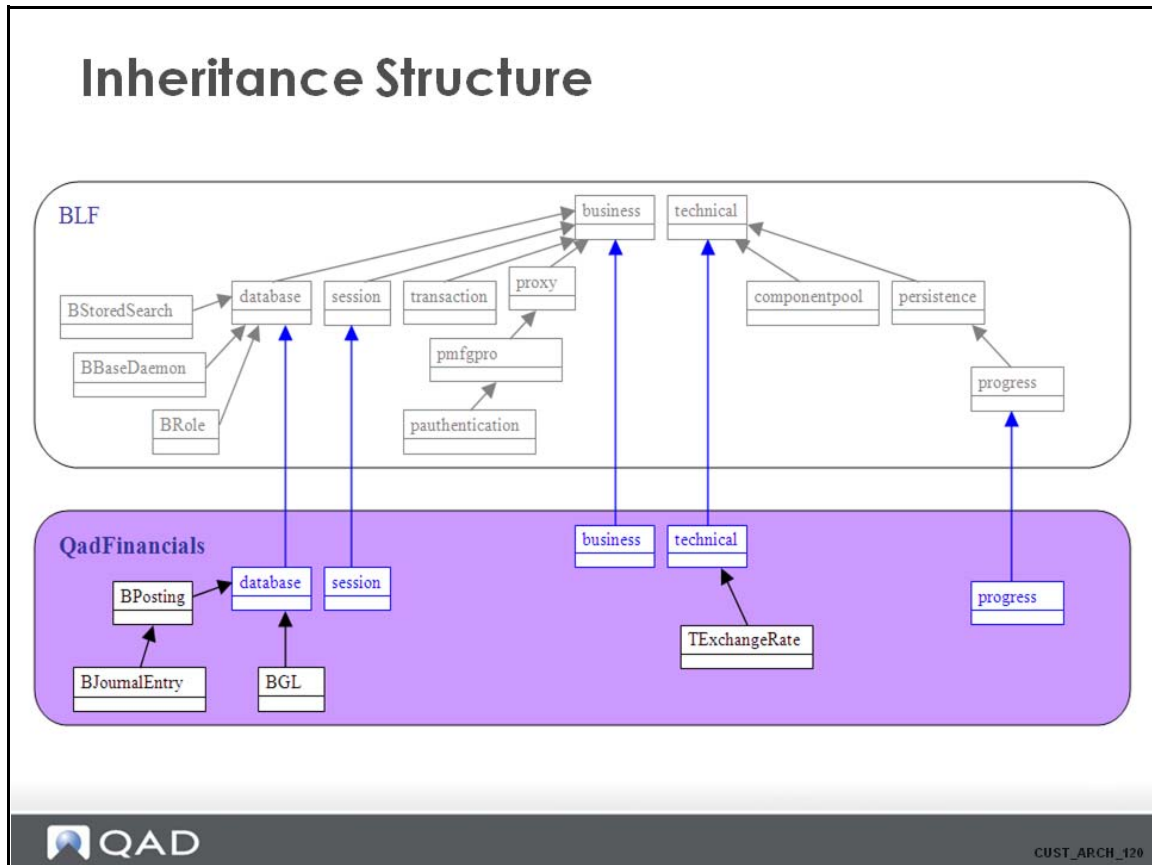


## BLF:

- “Business” and “Technical” are the ancestor/super components for most of the other components in the system. These are abstract components containing generic logic, but are not suitable for instantiation.
- “Database” is the ancestor component for all components in the application that are linked to tables in the application database, encapsulating data access to these tables.
- “ComponentPool” is a component that handles the memory management for component instantiations at runtime. It is responsible for starting the underlying .r code for components in memory, caching it, and cleaning it up. This is an important component for customization, because it starts the CustomizationController component to guarantee available customizations are detected and activated.
- “Session” is a component that contains logic to store/restore all session-dependent properties. For each client session, a session instance is started on the BL. The session component is responsible for authenticating the client and for retrieving the settings specific for the entity/domain for which the client session is logged in.
- “Transaction” is the component that contains all logic necessary for the implementation of the “logical transactions” in the system. Examples: AddInstance(), CommitTransaction(), AbortTransaction() methods.

- “Persistence layer” is the component that contains the logic required to ensure the abstraction of data access in the system. Typical methods for this component are `ConnectDb()`, `ReadData()`, `WriteData()`, `ReadQuery()`, `SaveInstance()` and `LoadInstance()`. The last two methods are used for saving and restoring the state of an existing component instance.
- Generic implementation of the daemons. For example, the XML daemon “`BXmlDaemon`” inherited from “`BBaseDaemon`”.
- Security implementation. Roles, role permissions (“`Brole`”), and role membership (“`BUserRole`”).
- Typical implementation of the PMFG/PRO proxy. This component is used to call the MFG/PRO APIs. An example is the `PAuthentication`. The session component uses this component to call the authentication service.
- The components “`BStoredSearch`” and “`BControlProperty`” contain the code used for resp. Browse stored searches and UI design mode (UI customization).

## Inheritance Structure



### QadFinancials:

- This project contains the application-specific components.
- “Session” component is what we call a “Leaf” class. This class is inherited from a component with the same name from another project (a project on which the current project is dependent). Leaf classes are used to enable generic programming, with the possibility of overriding or extending methods and functionality on a lower level. Session is a typical example of a leaf class, because on each project level, more information is important for a client session. Other examples of leaf classes are “Business”, “Database”, and “Technical”.
- The HTML documentation set is a good starting point for learning about business components in the application.

## Business Layer Component Structure - Terminology

### Business Layer Component Structure - Terminology

API vs. Instance method:

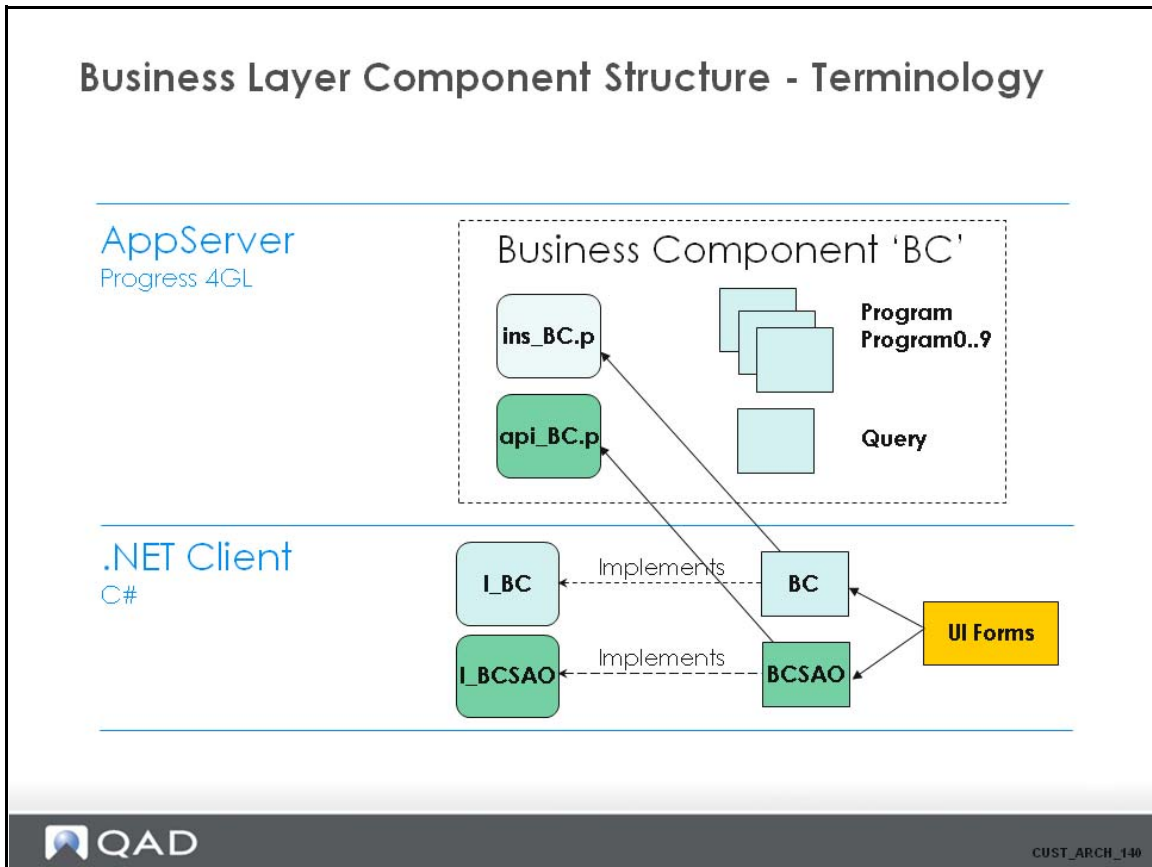
- API method: A separate stand-alone transaction with a clear function (returns True or False)

API methods can run without prior activation of a component instance

- Instance method: A method that uses (specifying or modifying) instance-dependent data (state)

The proper working of the instance-dependent methods is dependent on the state of the component instance

Business Layer Component Structure - Terminology



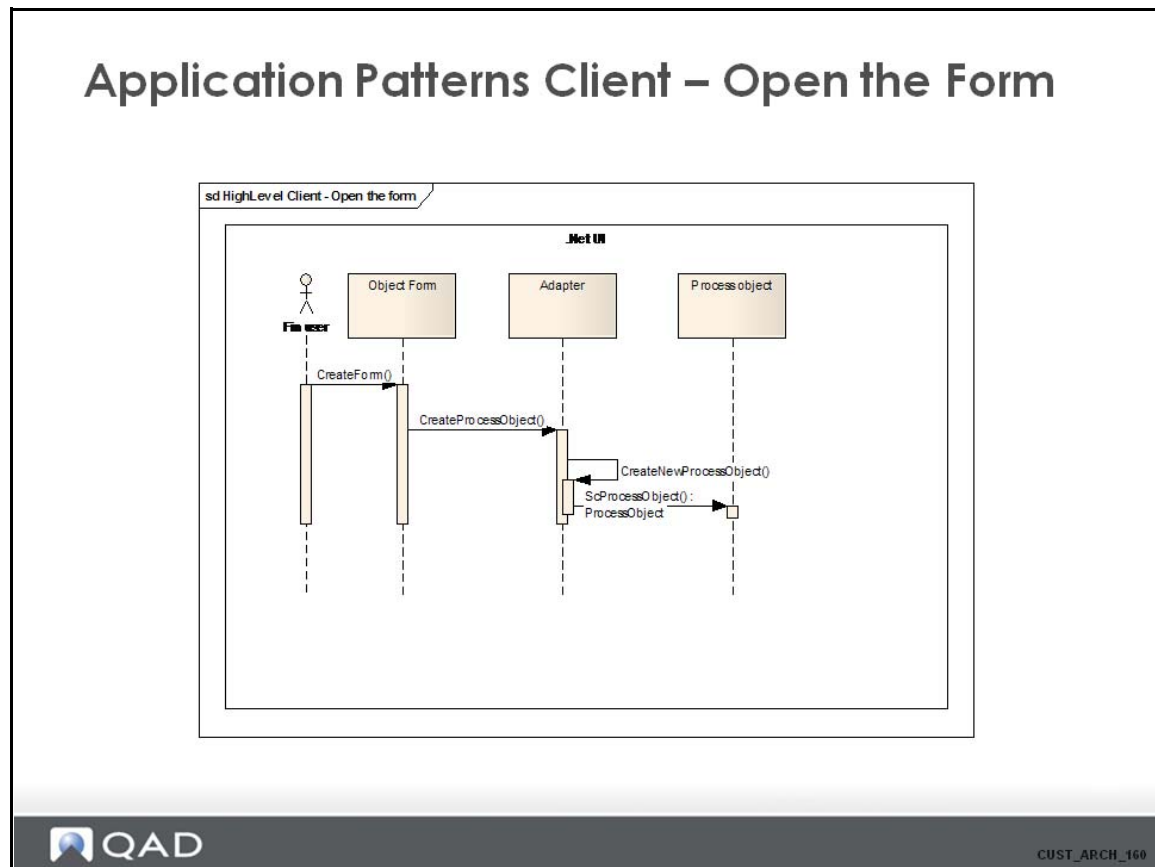
CUST\_ARCH\_140

## Application Patterns Overview

### Application Patterns

- The foundation infrastructure logic is built on predefined patterns for application functions
- For example:
  - Start an application session
  - Browse for an object
  - Start an activity
  - Create an object
  - Modify an object
  - Delete an object
  - Maintain a set of objects
- Patterns (re)use fixed flows in the code to execute code in predefined methods  
This is implemented in the abstract foundation components

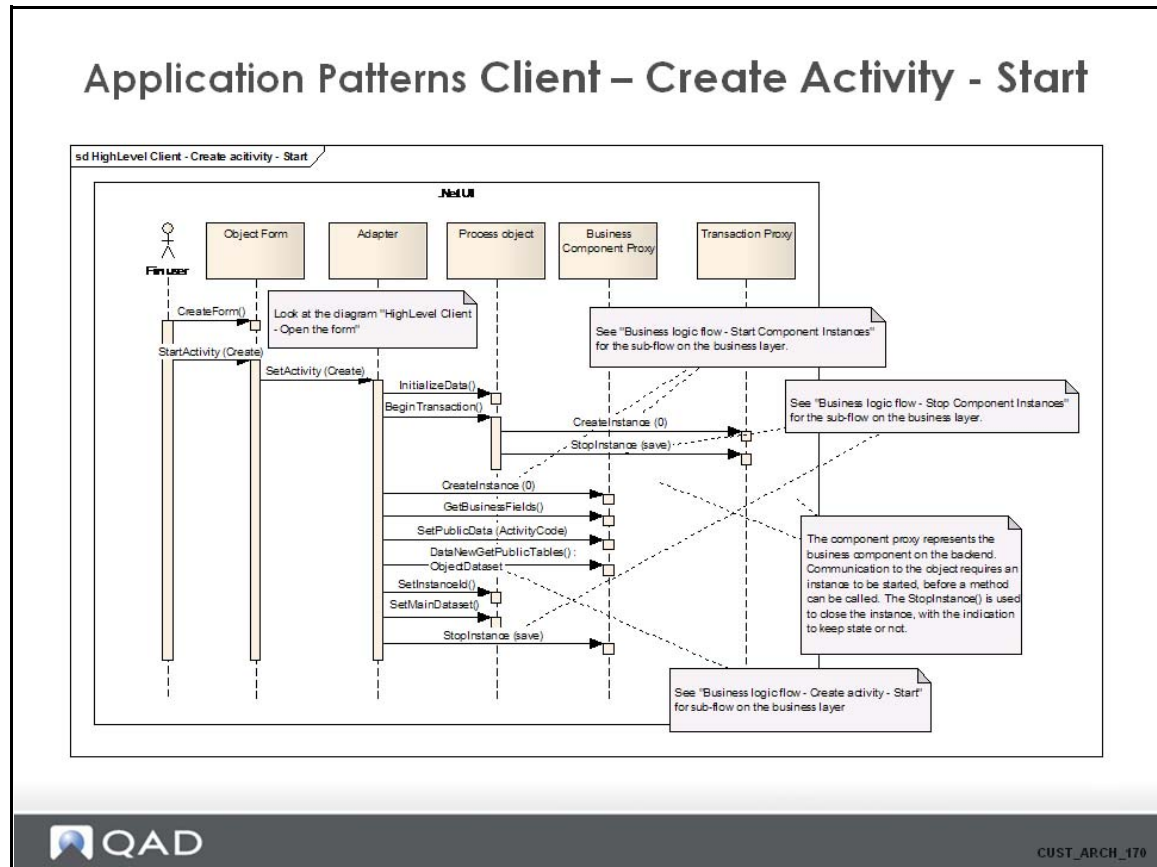
## Application Patterns Client



This diagram shows the flow that is executed when a form is opened on the UI. It does not have any direct interaction with the business layer.

Usually, the `CreateProcessObject()` method is run, and the name of a business activity (format '<BusinessComponent>.<Activity>', such as 'BGI.Modify') is supplied as a parameter. This automatically starts the necessary logic on the back-end business logic layer (as shown in the next section).

Application Patterns Client

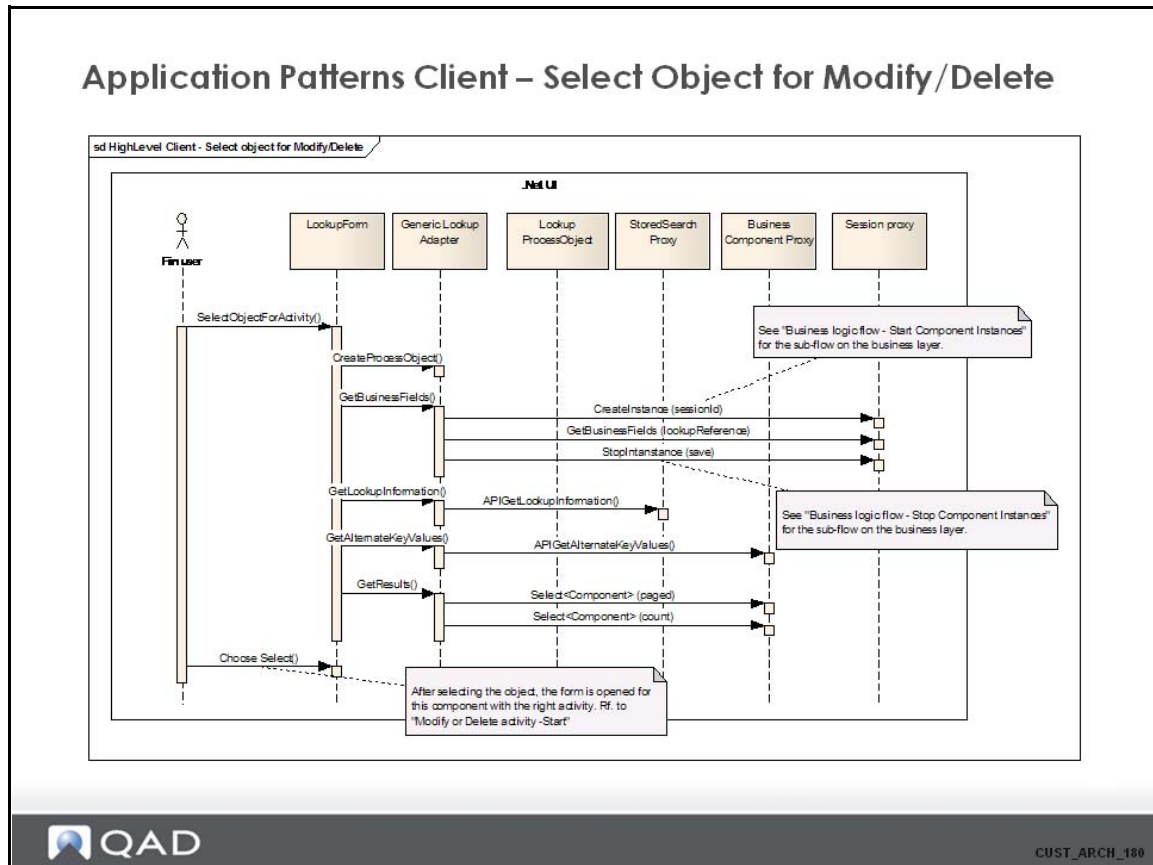


This diagram shows the flow that is executed when a Create activity is started. It assumes that the form is already instantiated. It stops when the form is displayed on the screen and is ready to receive input from the end user.

Notes:

- When a method on the business layer is called, it is called through the business component’s proxy on the client side. For a normal method, place a `createinstance()` and `stopinstance()` around the call because the normal methods need a component instance (and state) to execute.
- The `CreateInstance()` method technically creates the object instance on which the method can be called. When 0 is passed as the instance ID, the backend creates an instance, with a new unique instance ID and returns this. When an existing instance ID is passed (non-zero), the state for that instance is restored.
- The call to `GetBusinessFields()` is conditional. Since this information can be cached, the flow may skip this call if the information is still up-to-date on the client side.

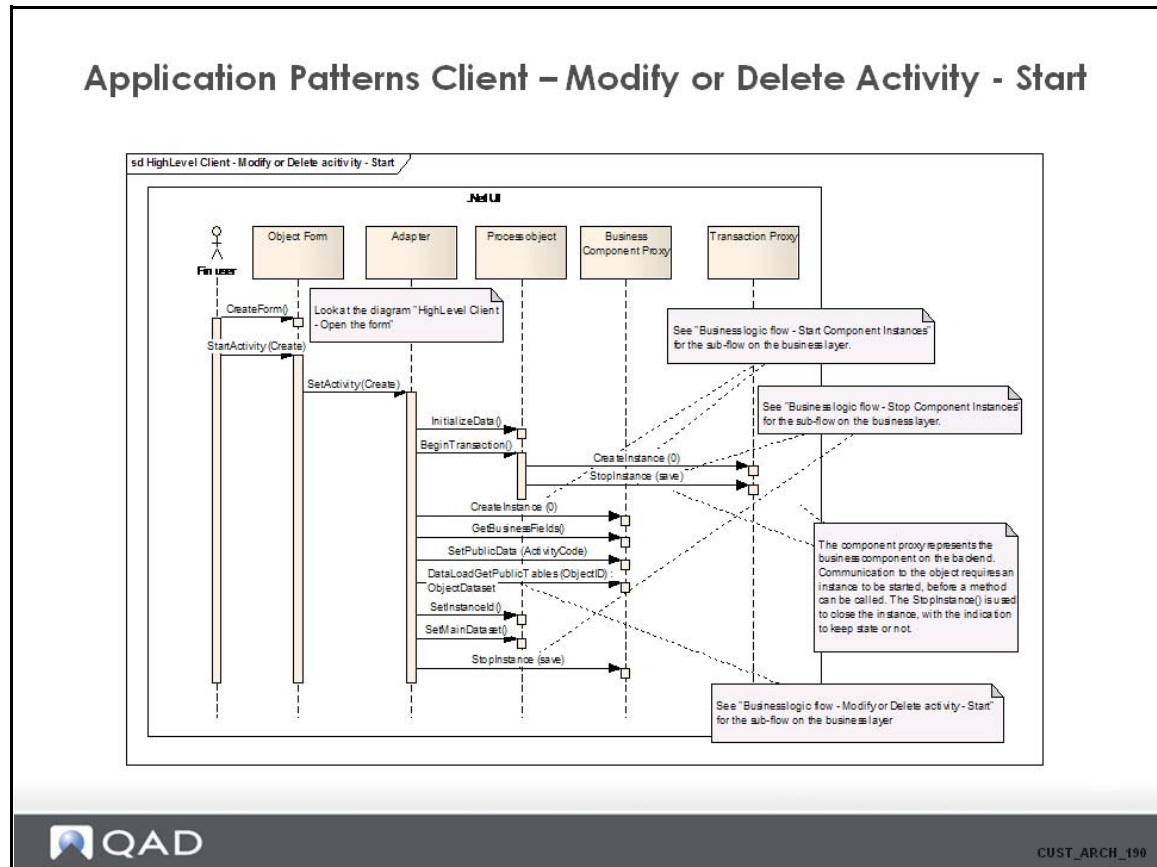
## Application Patterns Client



This diagram shows the flow that is executed when the user selects an activity for which an object must be first selected. It starts the browse form or lookup form, retrieves all meta data (such as filter fields or result fields information), and runs the query to retrieve the list of objects.

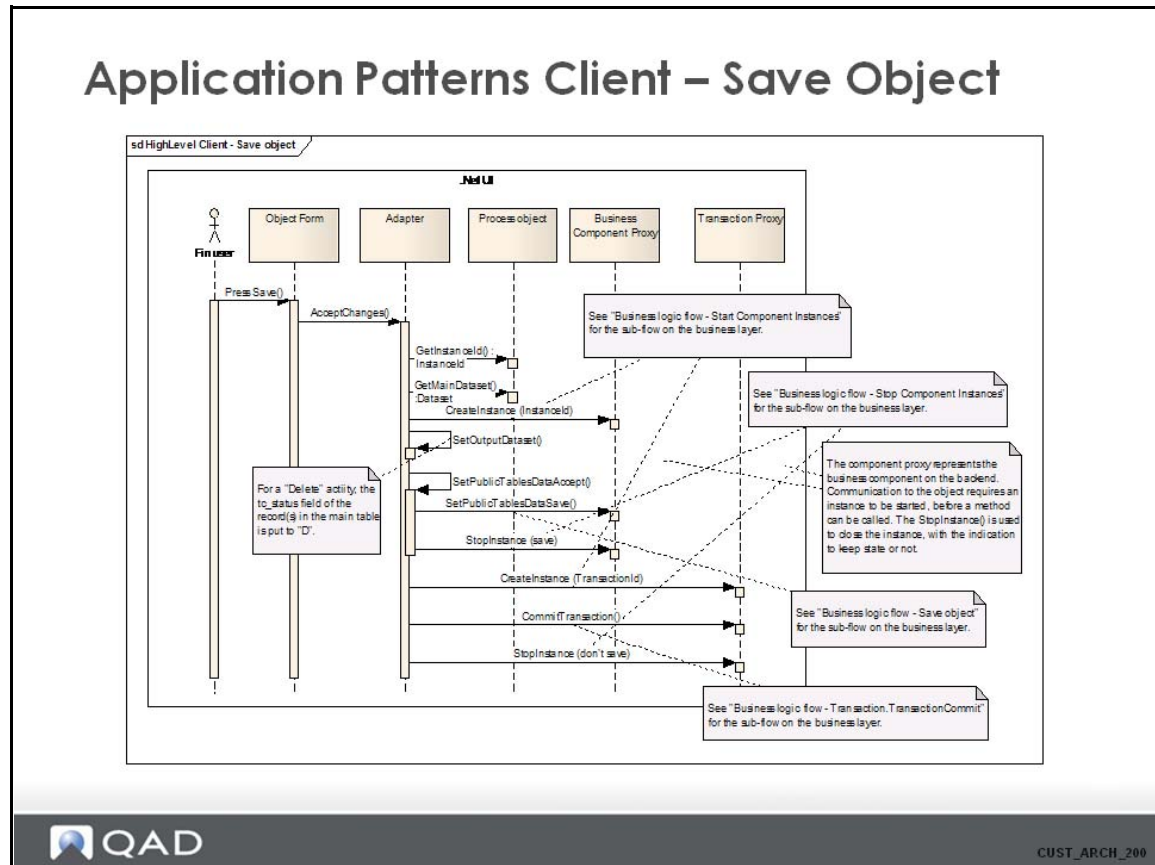
This is a common flow for all activities that require an object be selected before the object form for the activity is started (for example “Modify” or “Delete”).

Application Patterns Client



This diagram shows the flow that is executed when modifying or deleting an object. The main difference with the Create flow is the call to `DataLoadGetPublicTables` instead of `DataNewGetPublicTables`.

## Application Patterns Client

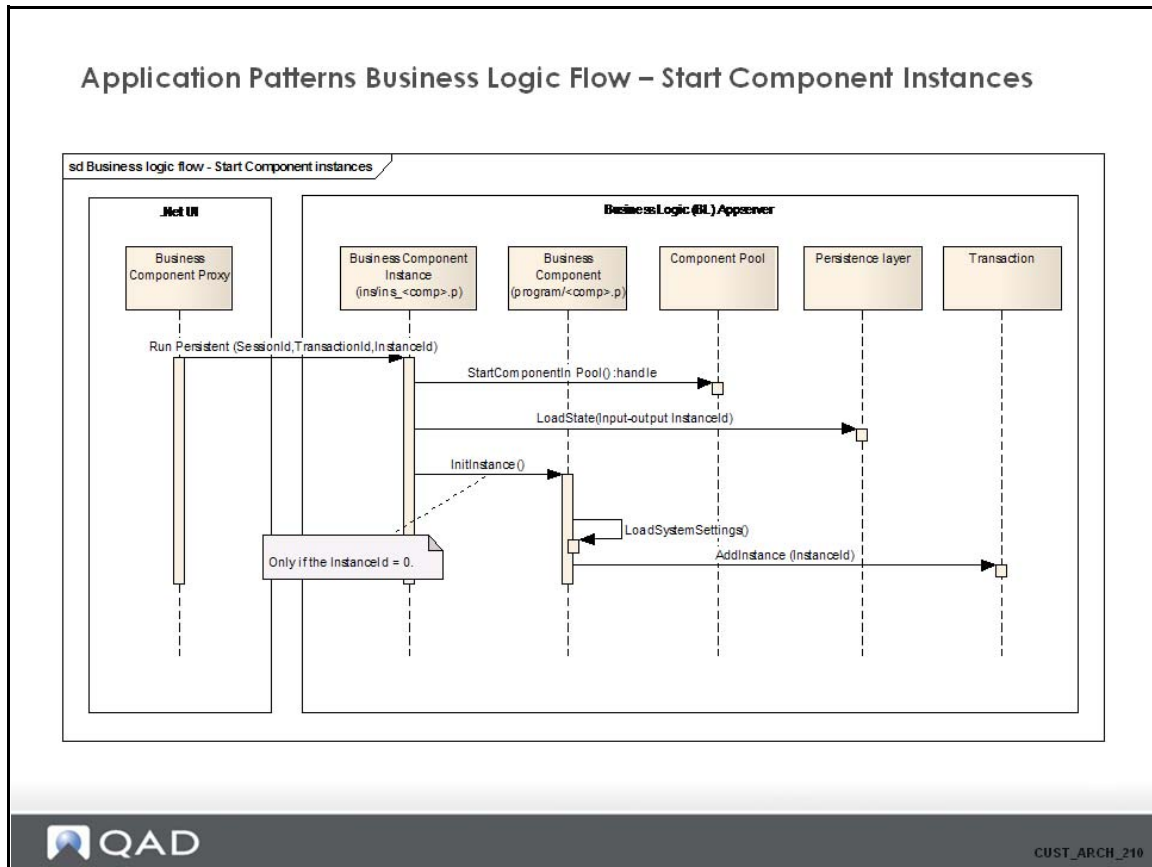


This diagram shows the flow that is executed when the user clicks Save and submits the form.

## Note:

- **SetOutputDataset:** Handles the completion of the tc\_status. It also ensures that only changed/new/deleted records are sent to the server.
- **SetOutputDataset:** When data is submitted for the “Delete” activity, the tc\_status for the main table records is set to “D”.

## Application Patterns Business Logic Flow



This diagram shows the flow that is executed when a business component instance is started. You can start a component instance in two ways:

1. With an existing instance ID, in which case the state of that instance is loaded, or
2. With a non-existing instance ID (=0), in which case the system creates an instance.

The instance is started by running the instance persistently on the AppServer. This locks the AppServer agent, and ensures that data and context remain available to the client session as long as the procedure is persistent in memory.

A component instance is started by running the ins\_\_ procedure persistently on the AppServer. SessionId, TransactionId, and InstanceId are passed as parameters.

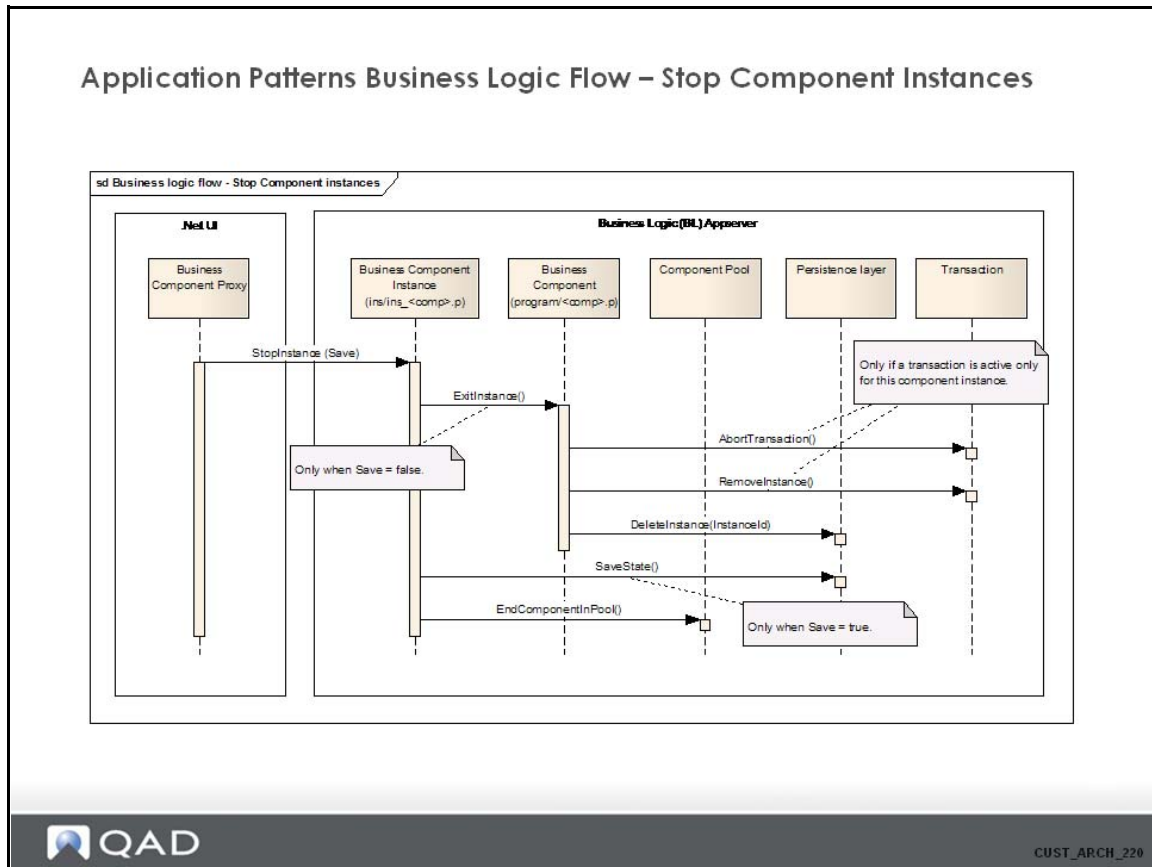
- SessionId is required. It is the component instance ID of the session started previously, which contains the information about the current session.
- InstanceId: If it is 0, it is assumed that a real new instance / state must be created. If it is <>0, the system restores the instance data associated with that instance ID.
- Complete TransactionId if a new instance must be associated with an existing logical transaction (as identified by TransactionId).

The role of the ComponentPool component should be clear in this pattern. Technically, the creation of a component instance ensures the right .r code is loaded in memory to enable execution of methods. The logic for starting a component instance is contained in ComponentPool. When the

system starts a component instance, ComponentPool first checks its internal cache of component instances (the set of related .r code persistent in memory), and returns a reference, if found, to the instance. If a reference is not found in the cache, it loads the required .r code in memory and returns a reference to it.

For customizations (see later), it is important to understand that the ComponentPool uses the CustomizationController component to establish if there is customized code for the component.

Application Patterns Business Logic Flow



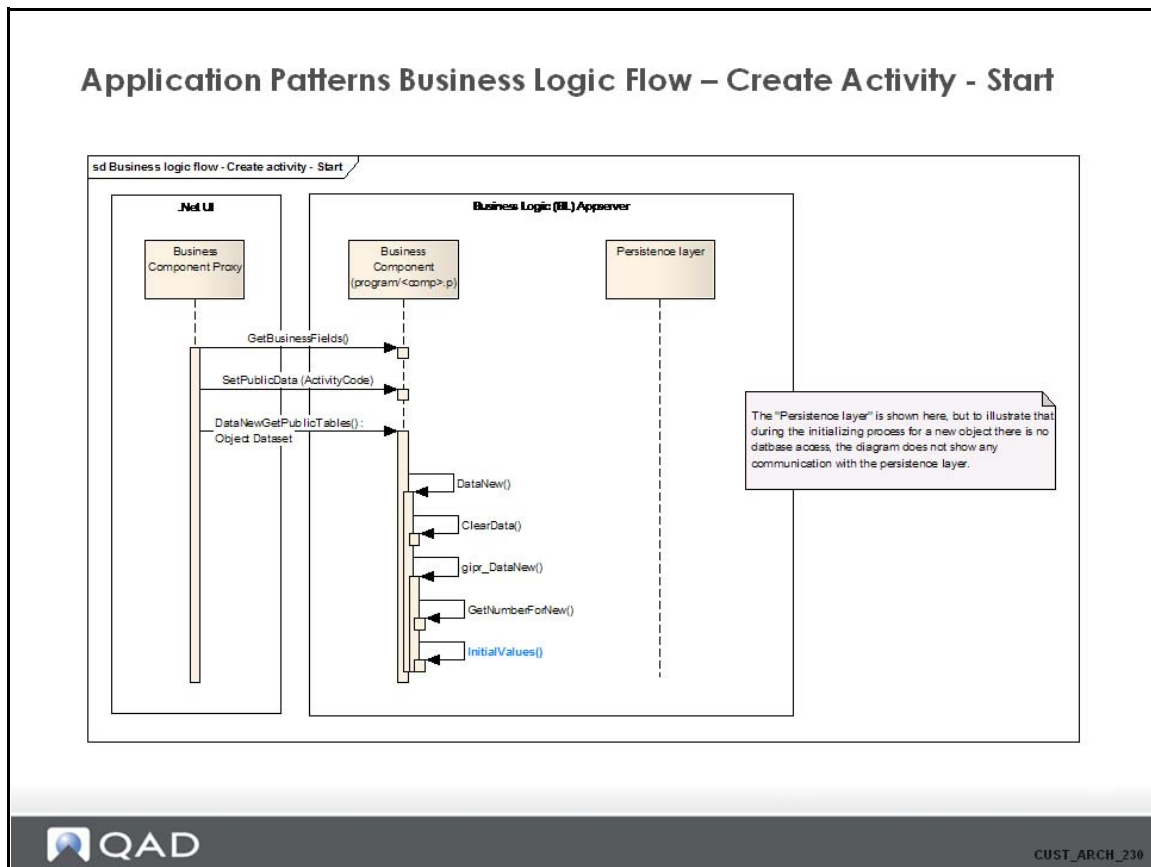
This diagram shows the flow that is executed when a business component instance is stopped. Stopping the instance from the UI means that the instance is no longer needed, and the lock on the stateless connection can be released. This is primarily done by stopping the persistent procedure.

Stopping an instance can be done in two ways (specified by the “Save” input parameter):

- Stop an instance with the intention of removing everything from the server. Not only is the program code released, but all related state information is also deleted.
- Stop the instance, but to keep the state information so that the instance with that state can be “revived” at a later stage.

The ComponentPool removes the .r code associated with the component instance from memory, but only if the number of cached components exceeds the component cache limit. The limit is defined in the server.xml configuration file for the backend Financials AppServer (<swaplimit>). Otherwise, the .r code stays persistent in memory and can be reused in a subsequent request for a component instance.

## Application Patterns Business Logic Flow

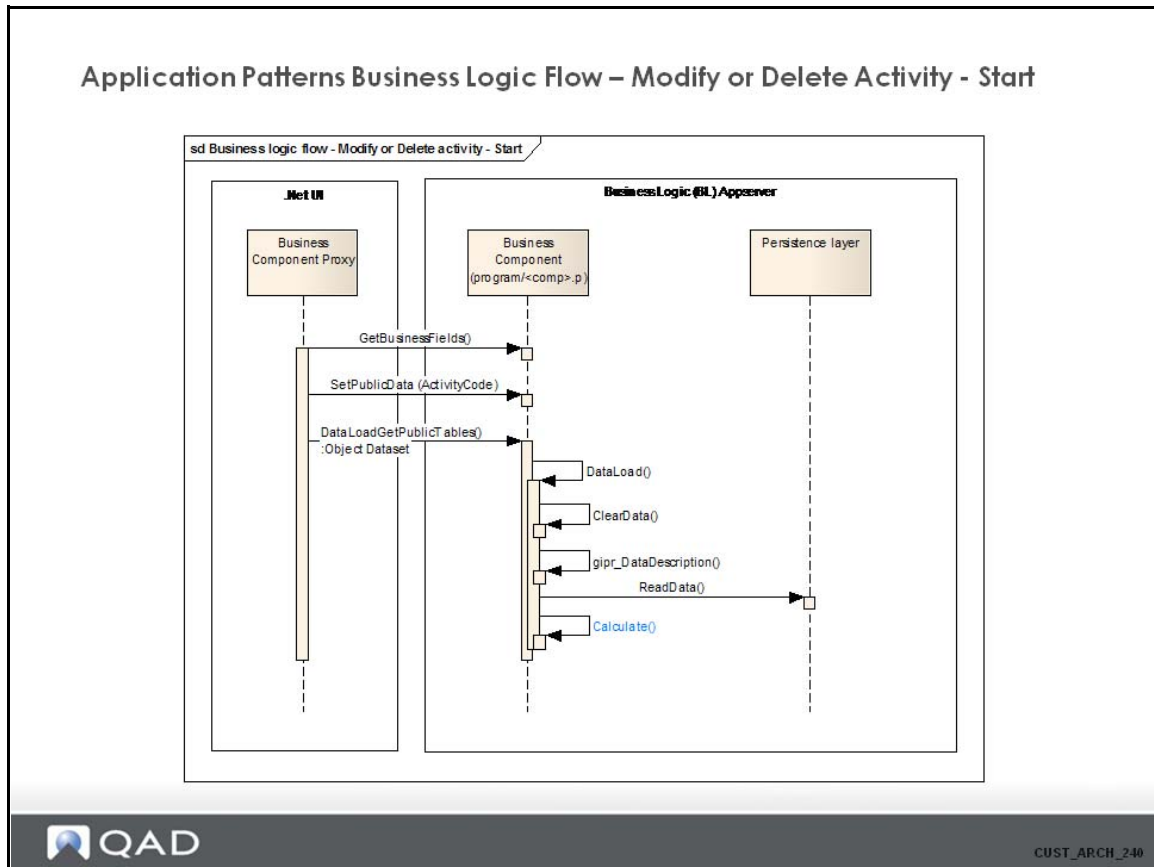


This diagram shows the program flow that is executed on the business logic layer when an object is created.

These UI calls are performed when the form is opened in Create mode.

`InitialValues` is typically a method that you can customize on the backend to specify initial values or to do initial calculations and defaulting.

Application Patterns Business Logic Flow



This diagram shows the program flow that is executed on the business logic layer when an object is loaded for modification.

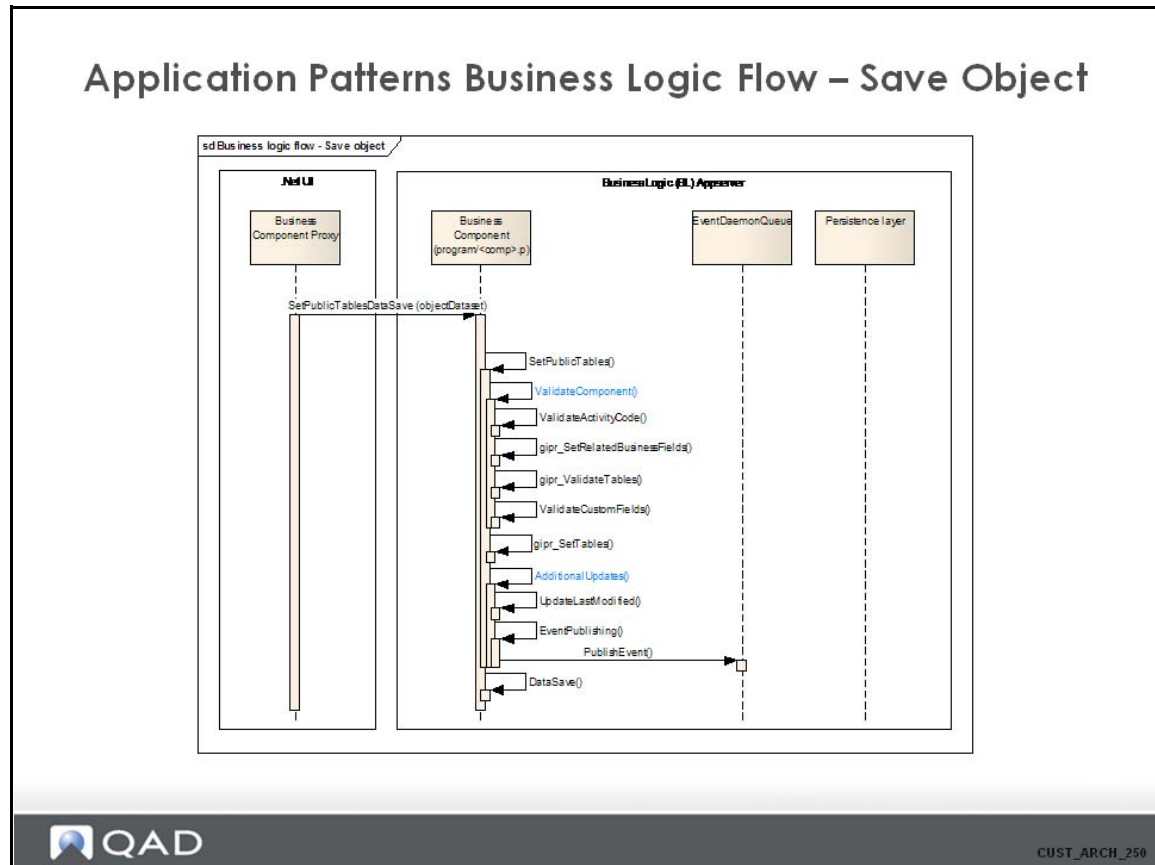
These UI calls are performed when the form is opened in Modify or Delete mode.

This flow basically handles the completion of the object dataset with data that can be modified / deleted in a later phase. The object dataset is returned to the UI, where it can be viewed.

The Calculate() method is typically used to fill in the calculated fields in the object dataset.

**Note** The GetBusinessFields() method is only executed when the local client cache for that data is not up-to-date.

## Application Patterns Business Logic Flow



This diagram shows the program flow that is executed on the business logic layer when an object is sent from the client to the backend for saving.

The objectDataset which is passed in as parameter to the SetPublicTablesDataSave(), only contains the changed rows. The internal flow merges the changes into the official object dataset on the backend during the validation process.

This diagram does not show the communication with the persistence layer in detail (this is shown in another diagram). The DataSave() method only writes data to the database when the component instance is not associated with a logical transaction. When the instance is associated with a logical transaction, the write to the database is performed in the “CommitTransaction” of the logical transaction itself, in a later step.

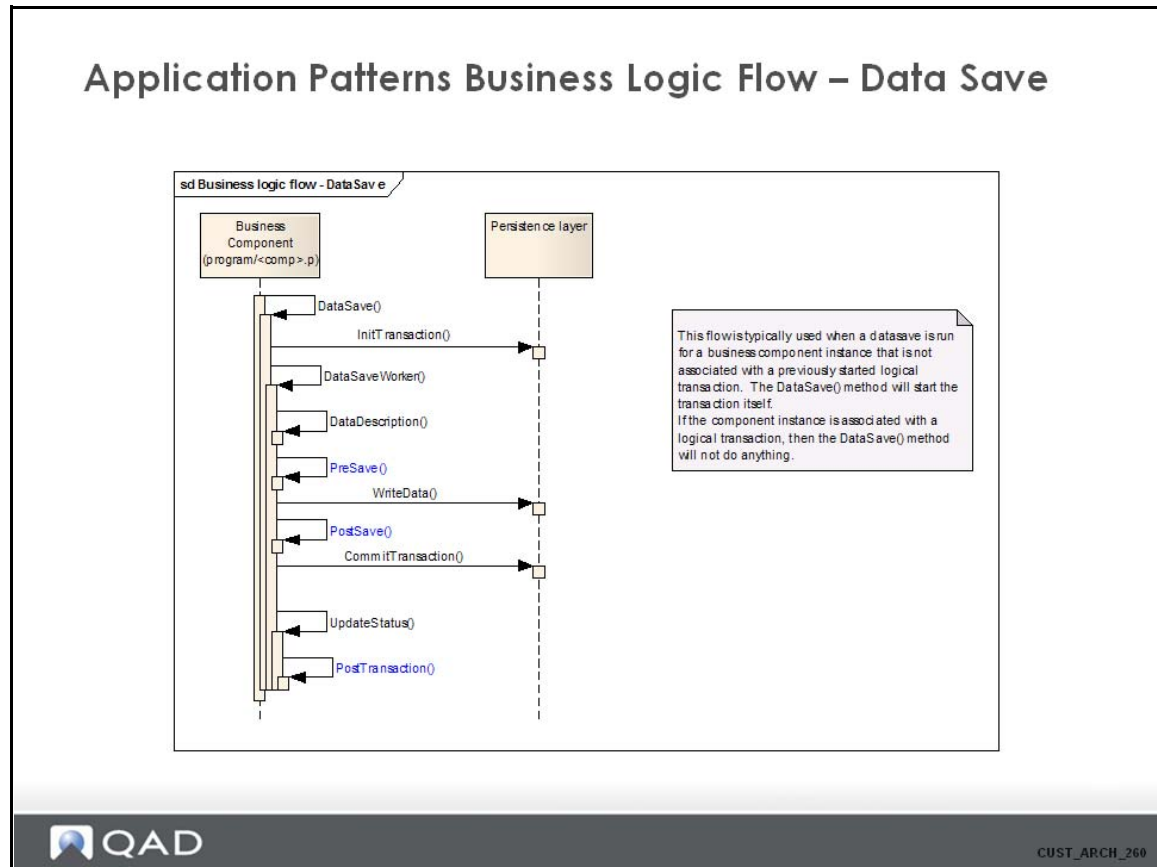
gipr\_validateTables is a generated procedure, and contains all validations based on the data model information (for example, mandatory fields and mandatory relations).

gipr\_SetTables is a generated procedure and handles the updating of the internal object dataset that contains data that is ready to be written to the database.

Typical customizations can be of:

- ValidateComponent: This method contains the logic to validate the data in the object dataset with the changed data.
- AdditionalUpdates: This method contains logic to perform updates on other components besides the current one, based on the data that is ready to be written to the database.

Application Patterns Business Logic Flow

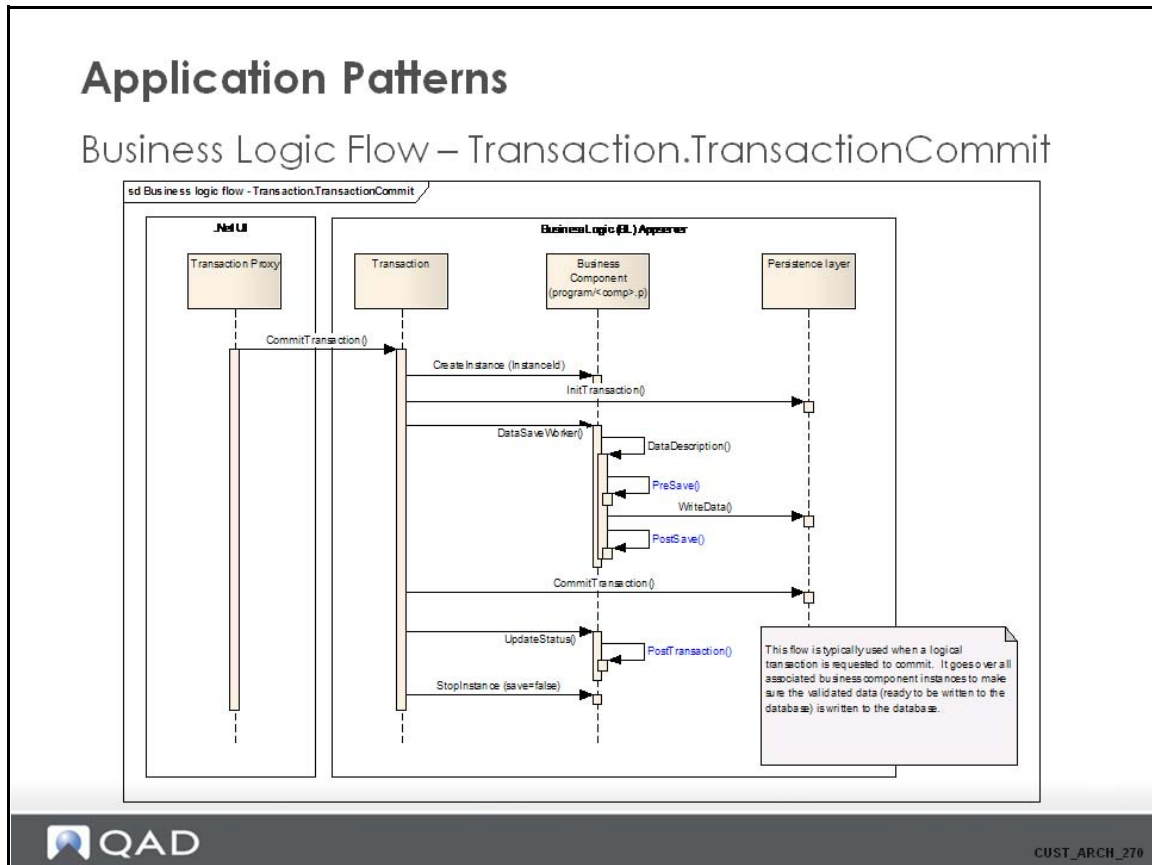


This diagram shows the detail program flow for the execution of the DataSave() method. This flow is only followed when the component instance for which the DataLoad is executed was not previously linked to a logical transaction. In the latter case, the DataSave() does nothing, and the logical transaction itself is responsible for writing changed data to the database.

It is assumed that the data in the object dataset is successfully validated.

Typical candidates for customization are the PreSave, PostSave, and PostTransaction methods.

## Application Patterns Business Logic Flow



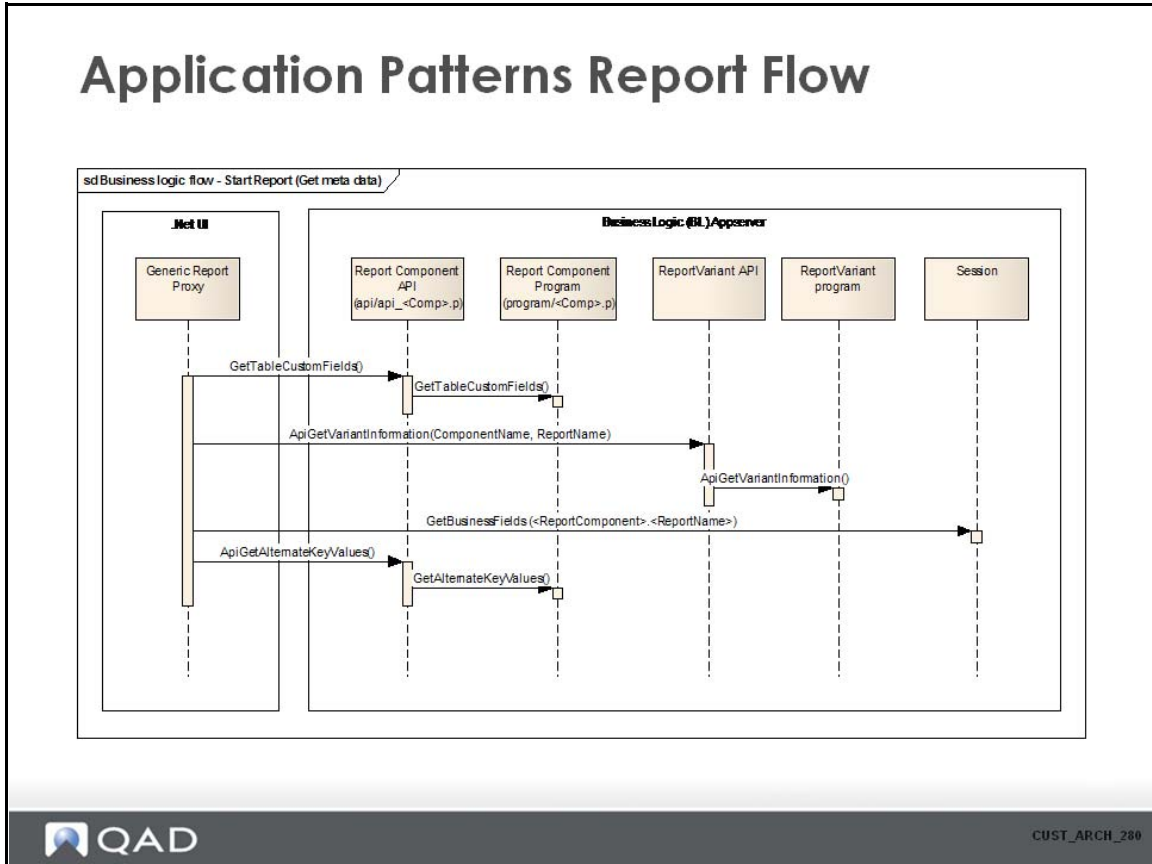
This diagram shows the detail program flow for the execution of the `Transaction.CommitTransaction()` method.

This is typically executed when a logical transaction, containing multiple associated component instances, needs to be committed.

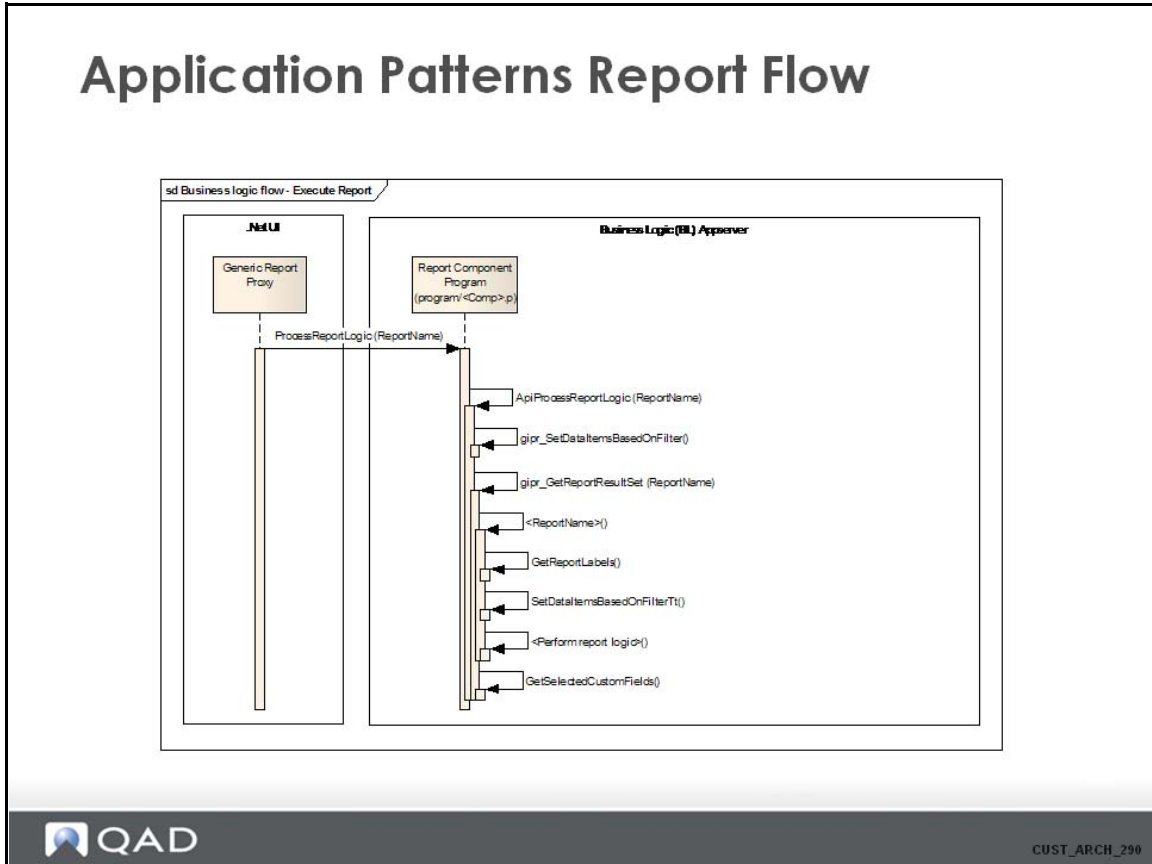
The logic loops through all associated component instances and writes the changed data to the database in sub-transactions. The Transaction component instance itself controls the life span of the physical transaction.

Typical candidates for customization are the `PreSave`, `PostSave`, and `PostTransaction` methods.

## Application Patterns Report Flow



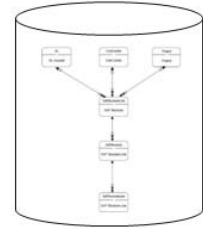
Application Patterns Report Flow



## Data Handling - Datasets

### Data Handling - Datasets

- Widely used in the Financials application layers to keep and manage data
- Imagine a small database in memory
  - A dataset contains one or more temp-tables
  - temp-tables in a dataset are related
  - temp-tables in a dataset can have indexes, and can have constraints defined
- A dataset can be passed as a parameter to a procedure
- A dataset can easily be serialized to XML format
- A Progress dataset matches a .NET dataset



## Data Handling - Datasets

## Data Handling - Datasets

```

DEFINE TEMP-TABLE tTable1 NO-UNDO
FIELD tcT1Field1 AS CHARACTER
FIELD tit1Field2 AS INTEGER
FIELD tit1ID AS INTEGER
INDEX prim IS UNIQUE PRIMARY tit1ID.


DEFINE TEMP-TABLE tTable2 NO-UNDO
FIELD tcT2Field1 AS CHARACTER
FIELD tit2Field2 AS LOGICAL
FIELD tit1ID AS INTEGER
FIELD tit2ID AS INTEGER
INDEX prim IS UNIQUE PRIMARY tit2ID.

DEFINE DATASET dDS1 FOR tTable1, tTable2
DATA-RELATION Relation1 FOR tTable1, tTable2
RELATION-FIELDS (tit1ID, tit1ID).

/* Continued on next page */

```

1  
  
2


CUST\_ARCH\_310

A dataset exists of temp-tables.

One way to define the dataset is shown in this example. It uses statically defined temp-tables, and defines a static dataset (also named a “named” or “typed” dataset). Another way could be to build the dataset dynamically using the “create dataset” statement.

1. Define the different temp-tables.
2. Define the dataset (with name dDS1) which holds the temp-tables tTable1 and tTable2. A data relation is defined between tTable1 and tTable2 within the dataset, linking the two tables using the tit1ID field.

## Data Handling - Datasets

## Data Handling - Datasets

```

/* Continued from previous page */

CREATE tTable1.
ASSIGN tTable1.tcT1Field1 = "This is content for Table1"
      tTable1.tiT1Field2 = 9873521
      tTable1.tiT1ID = 1.
CREATE tTable1.
ASSIGN tTable1.tcT1Field1 = "This is more content for Table1"
      tTable1.tiT1Field2 = 98778231
      tTable1.tiT1ID = 2.
CREATE tTable2.
ASSIGN tTable2.tcT2Field1 = "This is content for Table2 - child for Table1"
      tTable2.tiT2Field2 = YES
      tTable2.tiT1ID = 1
      tTable2.tiT2ID = 11.

DATASET dDS1:WRITE-XML("file","c:\temp\out.xml",?,?,?,?).
DATASET dDS1:WRITE-XMLSCHEMA("file","c:\temp\out.xsd").

```

3

4

3. Use the “write-xml” method on the dataset object to serialize the dataset to XML format. This can be very useful when you want to dump the data and be able to reload the same data for processing in another system.

4. Use the “write-xmlschema” method on the dataset object to generate the XML description file. This is useful for validating an XML file before trying to load it into a system.



CUST\_ARCH\_320

## Data Handling - Datasets


## Data Handling - Datasets

### Serialized Dataset in XML Format

```

<?xml version="1.0" ?>
- <dDS1 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <tTable1>
  <tcT1Field1>This is content for Table1</tcT1Field1>
  <tiT1Field2>9873521</tiT1Field2>
  <tiT1ID>1</tiT1ID>
</tTable1>
- <tTable1>
  <tcT1Field1>This is more content for Table1</tcT1Field1>
  <tiT1Field2>98778231</tiT1Field2>
  <tiT1ID>2</tiT1ID>
</tTable1>
- <tTable2>
  <tcT2Field1>This is content for Table2 - child for Table1</tcT2Field1>
  <tiT2Field2>true</tiT2Field2>
  <tiT1ID>1</tiT1ID>
  <tiT2ID>11</tiT2ID>
</tTable2>
</dDS1>

```


CUST\_ARCH\_330

The root element in the XML file is the dataset.

Every record is a sub-element of the dataset element, with the name of the table as the name of the element. An alternative hierarchical representation exists based on the data relations; in this case it would mean that the `<tTable2>` element would be a sub-element of `<tTable1>` (where `<tiID1> = 1`).

Data Handling - Datasets


## Data Handling - Datasets

### XML Schema File Presentation in Visual Studio

The screenshot displays the XML Schema Editor in Visual Studio for a file named 'out.xsd'. The schema is structured as follows:

- Root element: **dDS1** (type: dDS1)
  - Element: **tTable1** (type: tTable1)
    - Field: **tcT1Field1** (type: string)
    - Field: **tt1Field2** (type: int)
    - Field: **tt1ID** (type: int)
  - Element: **tTable2** (type: tTable2)
    - Field: **tcT2Field1** (type: string)
    - Field: **tt2Field2** (type: boolean)
    - Field: **tt1ID** (type: int)
    - Field: **tt2ID** (type: int)

A relationship line is shown between the **tt1ID** field of **tTable1** and the **tt1ID** field of **tTable2**, indicating a one-to-one relationship.


CUST\_ARCH\_340

Note the relationship between tTable1 and tTable2.

## Data Handling - Datasets

## Data Handling - Datasets

### Object Dataset

- Contains temp-tables
  - One per physical table (`t<TableName>` like `<TableName>`) (for example `tCountry` representing Country database table)
  - Three custom tables (`tCustomTable[0-2]`) with fixed set of fields (38 fields)
- Each temp-table has certain fields used by the framework
  - `Tc_Status`
  - `Tc_Rowid`
  - `Tc_ParentRowid`
- Relations based on `tc_rowid` and `tc_parentrowid`



CUST\_ARCH\_350

A business component is typically responsible for the data belonging to a certain object. For example, the business component BPosting is responsible for a Financials posting.

Typically, the object dataset contains a temp-table per physical table in the database. This temp-table contains a field for each physical field in the database, but it might also be extended with calculated fields. You can typically recognize these by their names. The calculated fields are named “`t<datatype><Name>`”.

`<datatype>` can be one of the following:

- “i” (integer)
- “d” (decimal)
- “t” (time)
- “c” (character)
- “l” (logical)
- “m” (memptr)
- “b” (blob)
- “h” (handle)
- “g” (int64)
- “p” (longchar)
- “a” (raw)

- “r” (rowid)

The temp-tables in the object dataset are called “class tables”. There is always one (and only one) main class table, which is the parent table for the object data.

For customization purposes, each object dataset also contains three custom tables that customization developers can use to extend the data in the object datasets. They contain a fixed set of fields that the framework recognizes and can be filled in by the customization developer.

Each temp-table in the object dataset has three fields that the framework uses for multiple purposes.

- tc\_rowid: A unique value in the object dataset. For new records, this is typically filled in with a negative number. For records loaded for change, this is filled in with the rowid from the database.
- tc\_status: An indication on what the system is doing with the data in this record. “N” means that the record is being created, “C” means that the record is being updated, “D” means that the record is being deleted. “” means that the record is unchanged in the transaction.
- tc\_parentrowid: This value is only filled in for records in a child table. It contains the value of the parent records tc\_rowid.

Avoid manipulating these fields, because doing so can yield unwanted/unpredictable behavior.

## Data Handling - Datasets

## Data Handling - Datasets

### Object Dataset

- Used for data transport between different layers
- Three variants used in the code:
  - The official object dataset. This represents what is in the database, or what is validated and ready for writing to the database
  - The initial object dataset. This holds the values that were initially read from the database for an object in modify mode
  - The update object dataset. This represents the modified data that still must be validated (typically data coming from the UI client)



CUST\_ARCH\_360

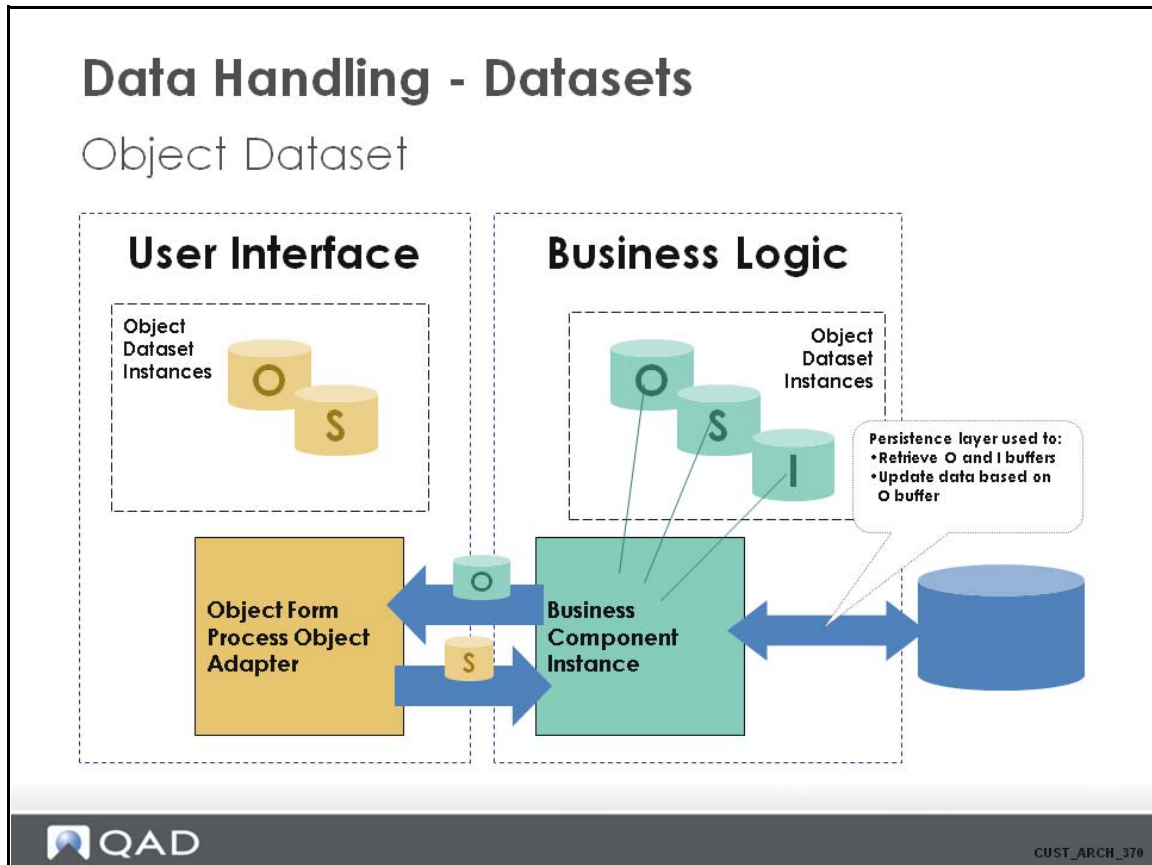
The object dataset is used to pass data between the layers. The business logic layer never directly uses database tables. The data is always retrieved via the persistence layer, and stored in the object dataset. The data the client UI layer is working with is received from the BL backend in the form of the object dataset.

The code three variants are:

- `<BusinessComponent>O`: The official object dataset. Temp-tables defined for the object dataset are `t_o<tableName>`, for which in the source code `t<TableName>` are buffers. For example, the buffer `tCountry` represents `t_oCountry`.
- `<BusinessComponent>I`: The initial object dataset. Temp-tables defined for the object dataset are `t_i<tableName>`. The data in this dataset is used for optimistic lock checking during the update.
- `<BusinessComponent>S`: The update object dataset. Temp-tables defined for the object dataset are `t_s<tableName>`. The data in this dataset is typically coming from the client and should be validated in the flow.

In the business code, the programmer mostly works directly on the official object dataset. What this means is that if he/she accesses the temp-table `t<tablename>`, the data from the official object dataset is used. During object data validation, the programmer should reference `t_s<tablename>`.

## Data Handling - Datasets



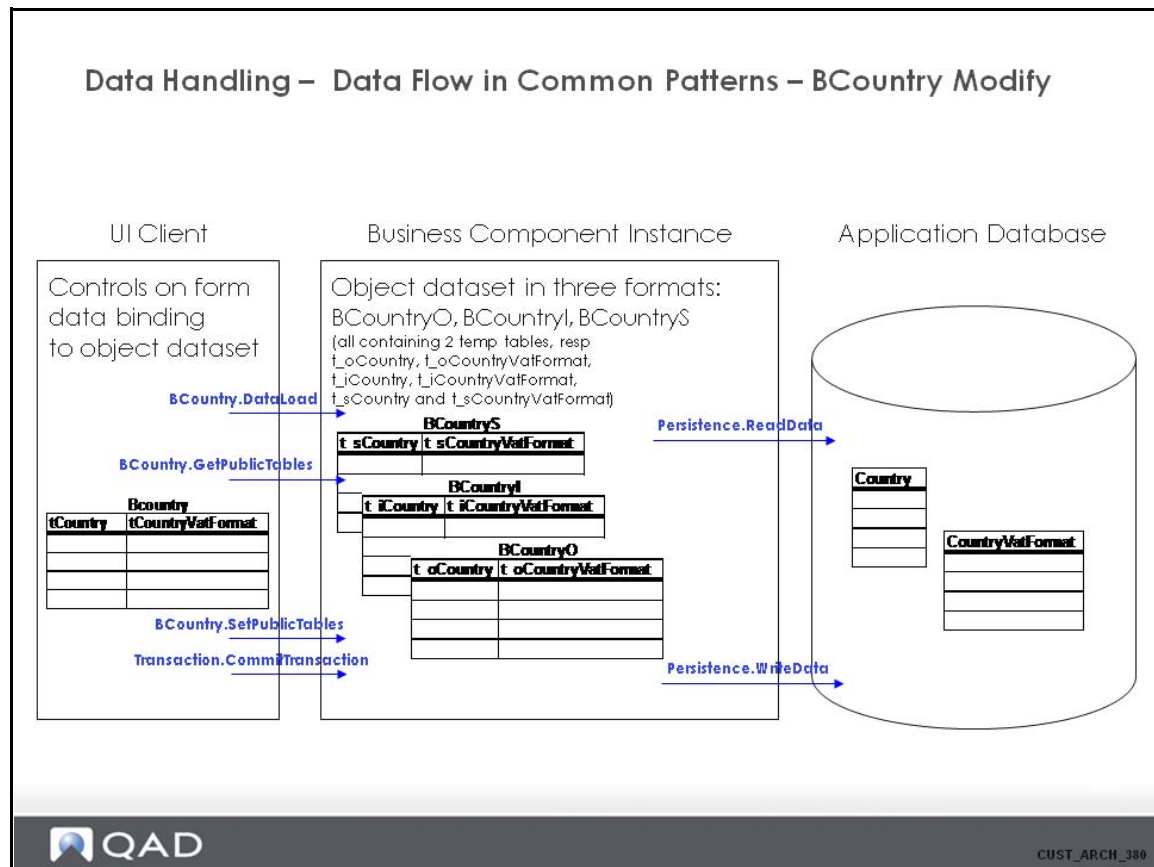
The business layer uses the object dataset instances to:

- Store data that has been read from the database (in the I and O instances)
- Prepare data to write to the database (in the O instance)
- Store data that has been received from a client, but is not yet validated. (in the S instance)

The UI layer uses the object dataset instances to:

- Receive the data from the BL (O instance)
- Send the data to the BL (S instance)

## Data Handling - Data Flow In Common Patterns



- BCountry is a business component that inherits from the “Database” component. This ensures that the component encapsulates all data access to the tables it is associated with. BCountry is responsible for tables Country and CountryVatFormat, for which a 1-N primary relation is defined in the datamodel.
- A BCountry component instance has three defined datasets (BCountryS, BCountryI, BCountryO). These datasets are used to store the records that are updated and passed around in the system.
  - BCountryI: Contains records that were originally read from the database when data is loaded for change.
  - BCountryO: Contains records to write to the database with the next DataSave / CommitTransaction.
  - BCountryS: Contains data (record(s) in tCountry and tCountryVatFormat) that is presented by a client to be updated. This is unvalidated data.
- When data is loaded in the BCountry component instance through “DataLoad”, the instance contains a “database instance”.
- A call to DataLoad triggers a call to Persistence.ReadData, which reads data from the database to the BCountryO and BCountryI datasets.
- A call to GetPublicTables transfers the object dataset to the client. The client displays the information because controls are bound to the dataset.

- A call to `SetPublicTables` transfers the changes in the object dataset from the client to the `BCountry` component instance, and data is validated.
- A call to `Transaction.CommitTransaction` (or `BCountry.DataSave`) triggers a call to `Persistence.WriteData`, which stores the changes in the application database.

## Data Handling - Data Flow in Patterns

### Data Handling – Data Flow in Patterns

Common Activity Method Flows: New

- `DataNew`: Create a new database instance
- `GetPublicTables`: Get the database instance data from the component instance to the user interface
- <<< change data on the UI representation >>>
- `SetPublicTables`: Send the changed database instance data from the UI to the component instance. Validate the changes
- `DataSave`: Write the changes to the database (only possible if the database instance was validated successfully before this)



CUST\_ARCH\_390



## Data Handling - Data Flow in Patterns

### Data Handling – Data Flow in Patterns

Common Activity Method Flows: Modify

- `DataLoad`: Load one or more existing database instances
- `GetPublicTables`: Get the database instance data from the component instance to the user interface
- <<< change data on the UI representation >>>
- `SetPublicTables`: Send the changed database instance data from the UI to the component instance. Validate the changes
- `DataSave`: Write the changes to the database (only possible if the database instance was validated successfully before this)



CUST\_ARCH\_410

Data Handling - Data Handling in Object Datasets - Modify Object

**Data Handling – Data Handling in Object Datasets – Modify Object**

Country = City  
CountryValFormat = CityVF  
ic\_state = stat

	UI layer		BL layer						Database	
	iCity	iCityVF	BCountryS		BCountryI		BCountryO		City	CityVF
			t_sCity	t_sCityVF	t_iCity	t_iCityVF	t_oCity	t_oCityVF		
Starting situation									"BE"	"Vat1"
After calling DataLoad for BE					"BE" (state="")	"Vat1" (state="")	"BE" (state="")	"Vat1" (state="")	"BE"	"Vat1"
After calling GetPublicTables	"BE" (state="")	"Vat1" (state="")			"BE" (state="")	"Vat1" (state="")	"BE" (state="")	"Vat1" (state="")	"BE"	"Vat1"
After calling CreateCountryVatFormat (grid)	"BE" (state="")	"Vat1" (state="")			"BE" (state="")	"Vat1" (state="")	"BE" (state="")	"Vat1" (state="")	"BE"	"Vat1"
After Changing data on the form	"BE" (state="C")	"Vat1" (state="C")			"BE" (state="")	"Vat1" (state="")	"BE" (state="")	"Vat1" (state="")	"BE"	"Vat1"
After calling SetPublicTables with validate OK	"BE" (state="C")	"Vat1" (state="C")	"BE" (state="C")	"Vat1" (state="C")	"BE" (state="")	"Vat1" (state="")	"BE" (state="C")	"Vat1" (state="C")	"BE"	"Vat1"
After calling SetPublicTables with validate NOT OK	"BE" (state="C")	"Vat1" (state="C")	"BE" (state="C")	"Vat1" (state="C")	"BE" (state="")	"Vat1" (state="")	"BE" (state="C")	"Vat1" (state="C")	"BE"	"Vat1"
After calling DataSave/Commit	"BE" (state="C")	"Vat1" (state="C")	"BE" (state="C")	"Vat1" (state="C")	"BE" (state="")	"Vat1" (state="")	"BE" (state="C")	"Vat1" (state="C")	"BE"	"Vat1"

**Data flow in case of changed object, with one new detail line**

CUST\_ARCH\_420

Besides the normal logical validations of data, the system also checks optimistic locking. If the DataSave() fails on optimistic locking, the system updates the records in the t\_i and t\_o temp tables with the latest (correct) values from the database, and updates the t\_s records. The user is also notified about this update.

## Data Handling - Data Flow in Patterns

### Data Handling – Data Flow in Patterns

Common Activity Method Flows: Delete

- `DataLoad`: Load one or more existing database instances
- `GetPublicTables`: Get the database instance data from the component instance to the user interface
- <<< delete confirmation >>>
- `DataDelete`: Validate whether the database instance can be deleted, and set status to 'D'
- `DataSave`: Delete the database instances marked with `tc_status = 'D'`



CUST\_ARCH\_430

## Data Handling - Data Handling in Object Datasets - Delete Objects


**Data Handling – Data Handling in Object Datasets – Delete Object**

ObjectDataset Bcountry (class tables Country and CountryVatFormat)

Country= Ctry  
CountryVatFormat= CtryVF  
tc\_status= stat

	UI layer		BL layer				Database		
			BCountryS		BCountryI		BCountryO		
	tCtry	tCtryVF	t sCtry	t sCtryVF	t iCtry	t iCtryVF	t oCtry	t oCtryVF	
Starting situation								"BE"	"Vat1"
After calling <b>DataLoader</b> BE					"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE"
After calling <b>GetPublicTables</b>	"BE" (stat="")	"Vat1" (stat="")			"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE"
After clicking "Delete" button								"IN"	"Vat11"
After calling <b>SetPublicTables</b> with validate OK	"BE" (stat="D")	"Vat1" (stat="")			"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE"
After calling <b>SetPublicTables</b> with validate NOT OK	"BE" (stat="D")	"Vat1" (stat="")	"BE" (stat="D")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="D")	"Vat1" (stat="")	"BE"
After calling <b>DataSaveCommit</b>	"BE" (stat="D")	"Vat1" (stat="")	"BE" (stat="D")	"Vat2" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="D")	"Vat1" (stat="")	"IN"
								"Vat2"	"Vat11"

Dataflow in case of a deleted object (Country "BE")


CUST\_ARCH\_440

## Error Handling

### Error Handling

- No user interaction possible
- Collect error messages in table `tFcMessages`
- `GetPublictFcMessages`  
or  
output parameter `tFcMessages`  
for the API interface



CUST\_ARCH\_450

The business logic writes messages to a temp-table, not by creating records directly in the table, but by calling method `SetMessage`.


This table is then returned to the client.

## Error Handling

### Error Handling

tFcMessages:

- tcFcBusMethod
- tcFcContext
- tcFcExplanation
- tcFcFieldLabel
- tcFcFieldName
- tcFcFieldValue
- tcFcIdentification
- tcFcMessage
- tcFcMsgNumber
- tcFcRowid
- tiFcSeverity
- tcFcType


CUST\_ARCH\_460

### tcFcMessage

The actual (error) message

### tcFcType

Type of message

E (standard correctable error; for example, an error caused by incorrect user input)

D (non-correctable error; for example a restricted delete error)

S (system error; for example a configuration file not found error)

W (warning)

I (information)

### tiFcSeverity

1 (highest severity)

to

5 (lowest severity)

tcFcMsgNumber

Unique key to identify the error message and look it up in the application source code

tcFcBusMethod

Name of the business method that raised the error

tcFcFieldLabel

tcFcFieldName

tcFcFieldValue

tcFcRowid

If the error is a validation error on data in the object dataset, these columns indicate what record and what column on that record contains an incorrect value.

tcFcContext

tcFcExplanation


tcFcIdentification

## Error Handling

### Error Handling

`oiReturnStatus`

- = 0 business code executed without issues
- < 0 business code raised an error  
(any updates were rolled back)
- > 0 business code raised a warning  
(any updates were committed)


CUST\_ARCH\_470

(Almost) every business method has an output parameter `oiReturnStatus` to indicate if the method executed correctly.

Known return statuses:

- +1 Warning
- 1 Validation error
- 2 Optimistic lock error
- 3 Standard run-time error
- 4 No results found
- 5 Fatal error
- 98 Progress runtime (error not raised by the application, but by the Progress runtime)

## Non-intrusive Customization: Development

Non-intrusive Customization: Development



## Requirements for Customization

### Requirements for Customization

General:

- Manageable
- Upgradeable
- Acceptable learning curve
- Documented
- Training available



CUST\_DEV\_020

#### General requirements

##### [G1] Manageable

The customizations should be easily manageable. This means that a developer is able to add and change customization code using his own preferred development tools. For Progress this might be the Progress Editor or OE architect.

##### [G2] Upgradeable

The effort required to move non-intrusive customizations from one version of the standard product to another should be minimized and predictable.

When customizations are in place at a customer installation, it should be relatively easy for the customer to upgrade the customizations to a newer version of the standard product.

##### [G3] Acceptable Learning Curve

It should be easy enough for an experienced developer to start writing non-intrusive customization code. The system should come with enough documentation, samples, and templates.

It may also be necessary to provide customization training.

##### [G4] Extra Localizations

It should be possible for local service organizations to add specific localization code via non-intrusive customizations and deliver this as add-ons with the standard product.

[G5] Documented

Customizations should come with sufficient documentation. People should be able to use the documentation to set up and define a customization, code it, and maintain it. The documentation should also include the object-specific documentation. For example, what are the extendable methods of the Journal Entry component?

[G6] Training Available

Customization should come with training for QAD services on how the non-intrusive customization works. The training material should cover some of the most used customization techniques. Training material should be constructed so that Services can use it to explain how to implement NI customization to partners and customers.

## Requirements for Customization

## Requirements for Customization

Application:

- Add fields on existing forms for certain functions
- Change validations for application objects
- Add custom detail tables
- Change lookups
- Change field information
- Change behavior by adding program code
- Create custom queries for lookups
- Create custom browses
- Change form layout
- Change UI logic and add sub-forms
- Add GoTos (Related Views / Drill function)



CUST\_DEV\_030

Application changes

### [A1] Add fields on existing functions

It should be possible to add new custom fields to existing functions in the application. This means that an extra field can be added to the screen for a certain object (such as a GL account). For this, a developer needs to be able to write logic to retrieve and save the value for this field.

Typically, this type of field is stored in a customer-managed shadow table for the standard table. You cannot add fields directly in the standard tables.

The developer also needs to be able to specify other meta data for the custom fields (such as format, control type, label, lookup query, possible values).

This type of customization is separate from that already provided by the user-defined fields.

### [A2] Change validations

It should be possible to change validations associated with a certain application function. This should include field-level validations (like simple checking whether the value is in a certain range) and more complex inter-field and inter-table validations.

This also covers adding validations for user-defined and new custom fields.

### [A3] Add custom detail tables

It should be possible to add a custom table to represent detailed information on a certain object, and give access to the table on the screen. With this, the customization developer needs to be able to write logic to retrieve, store, and validate this data.

The developer also needs to be able to specify other meta data for the fields in the new table (such as format, label, lookup query, possible values).

### [A4] Change lookups

It should be possible to extend the lookup queries with extra fields, so that custom fields can be shown on lookup grids.

In addition to adding custom fields in the query result grid, it should be possible to use these fields as filters.

### [A5] Change field information

A customization developer needs the ability to change certain meta data about fields in the standard application, such as format, label and lookup query. Care should be taken that all labels are translatable, using the same standard built-in mechanism.

### [A6] Change behavior by adding program code

It should be possible to change the application behavior by adding Progress 4GL code on the business layer. Typically, a customer should be able to add code when an object is initialized or validated, and also when an object is deleted. They should also be able to add code in well-defined places in the program flow typical to CB objects.

### [A7] Create own queries for lookups

A customization developer should be able to create his own queries that can be used for lookups. This is required in combination with new custom fields / tables. For these new fields, he must be able to specify the lookup query.

### [A8] Create own browses

A customization developer should be able to create new browses as replacements for the standard component browse. For example, if the browse for Supplier Invoice Approve needs modification, the customization developer should be able to create an own query, and point to this for the Supplier Invoice Approve activity.

### [A9] Form changes

It should be possible to add more controls to the forms without writing C# code. It should be possible to add a grid, a tab page (on an existing tab control), and a tab control.

### [A10] Change UI logic and add sub-forms

It should be possible to write extensions to the UI logic. The customization developer should be able to add code on certain events and add code before or after the normal code on the execution of predefined infrastructure methods.

It should also be possible to add extra forms for an existing application activity. These can be activated when certain events occur.

[A11] Add GoTos

A customization developer should be able to add related views to an existing object form. This includes creating their own queries, and adding them to the list of related views for a certain component. This should follow the same mechanism as the standard GoTo function. They should be available from the browse and the object form.

## BL Customizations

### BL Customizations

- Use of template procedures delivered with the standard application
- Each business component has its own template procedure, which must be implemented to activate the Customization for the component
- All Customization procedures must be put in a folder named `customcode` on the server running the business backend
- The `customcode` folder must be a subfolder of a folder in the `PROPATH` of the AppServer running the business backend



CUST\_DEV\_040

The standard application comes with a customization controller.

This piece in the application looks for procedures that contain customized code in a subfolder named “`customcode`”. It is started at startup of every AppServer agent for the Financials business layer.

## BL Customizations

## BL Customizations

### Customization Procedures

- Basis for Customization code
- Contain:
  - Version number
  - For each method of the business component a before and after hook  
`"<businessComponent>.<methodName>.[Before | After]"`  
 For example:  
`BCountry.AdditionalUpdates.Before,`  
`BCountry.ValidateComponent.After.`
  - Standard code can be extended using the *before* and *after* hooks. It can never be overridden or replaced



CUST\_DEV\_050

The name of the method and the business component can be used to navigate the HTML documentation to find the right documentation and source code for the standard implementation of the method.

Each method in a business component has by default at least one output parameter called `oiReturnStatus`. The value is negative if an error occurred, greater than 0 if a warning occurred, and 0 if the method executed without errors or warnings.

The following applies to the `oiReturnStatus`:

- If the standard code already set the `oiReturnStatus` value to a negative value, the “after” hook is not executed.
- If the code in the “before” hook sets the `oiReturnStatus` value to a negative value, the standard code is not executed, and the method stops execution and returns.
- The exception to the above rule is the central validation method “ValidateComponent”. The is because, for component validation, we want as much information as possible returned on invalid data.

## BL Customizations

## BL Customizations

### Customization Procedures

- Object datasets available
- Parameter passing is done via a buffer. For example, setting the output parameter `oiReturnStatus` would be done as follows:  
`assign t_Parameter.oiReturnStatus = viReturn.`
- Temp-tables from dataset parameters are automatically available  
 Data sets from the custom code are "bound" to the standard code
- Data access is fully controlled by Customization developer



CUST\_DEV\_060

Data access in the standard component logic is always implemented using the persistence layer as Data Access layer. For the customization, we do not want to apply this rule because the interface to the persistence layer is not straightforward. For the standard code development within the QAD development teams, the Component Builder tool automatically generates the necessary calls to the persistence layer, which is not possible for the customization code.

#### Attention!

Since everything stays persistent in memory, care should be taken for record scopes. For example, if you write code to update a table in the `PostSave.After` event, since the transaction is not finished, an exclusive lock on a record is downgraded to a share lock. To avoid problems, it is best to use a locally defined buffer (defined inside the internal procedure) for doing any database update. Or use the "release" statement as soon as the record is updated, and you do not need it anymore in the remaining flow.

**BL Customizations**

## BL Customizations

Example – Add initialization for a user-defined field

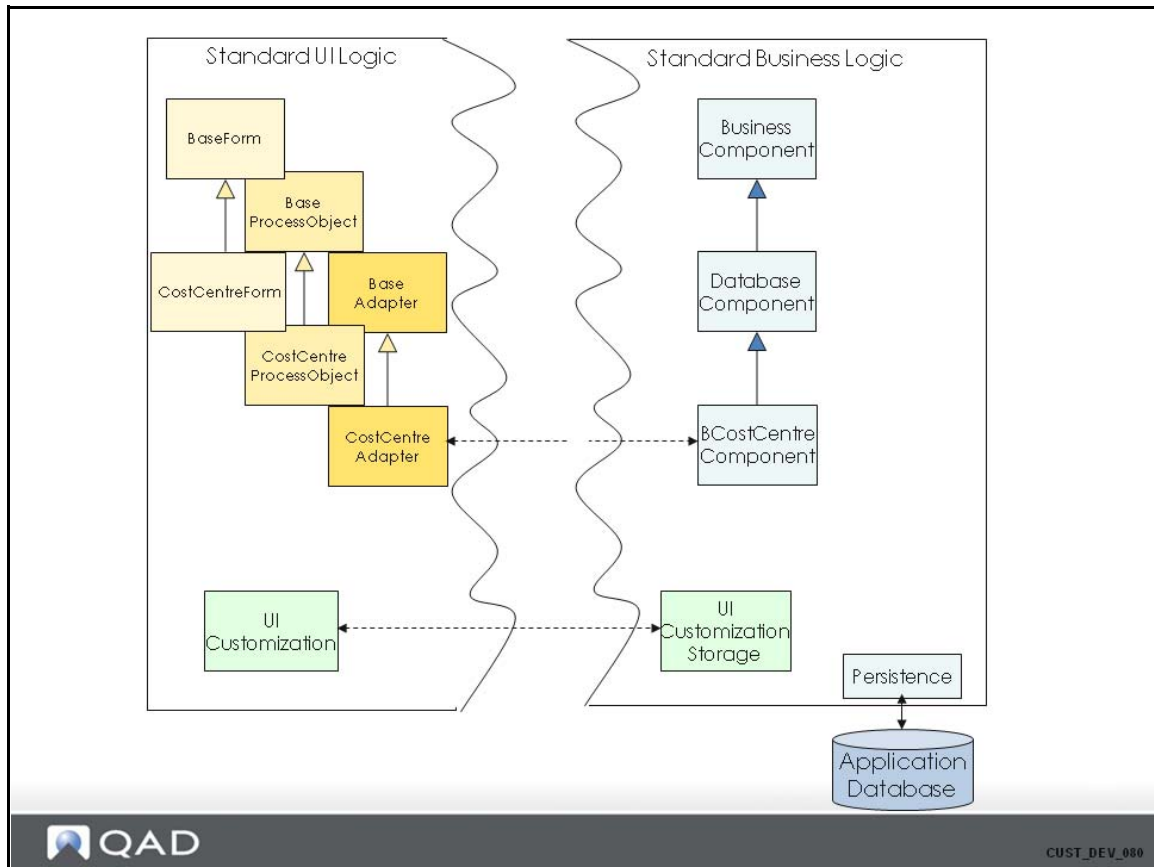
```
PROCEDURE BCountry.InitialValues.after:  
  if t_parameter.icTableName = "Country"  
  then assign tCountry.CustomInteger0 = 1000000.  
END PROCEDURE.
```



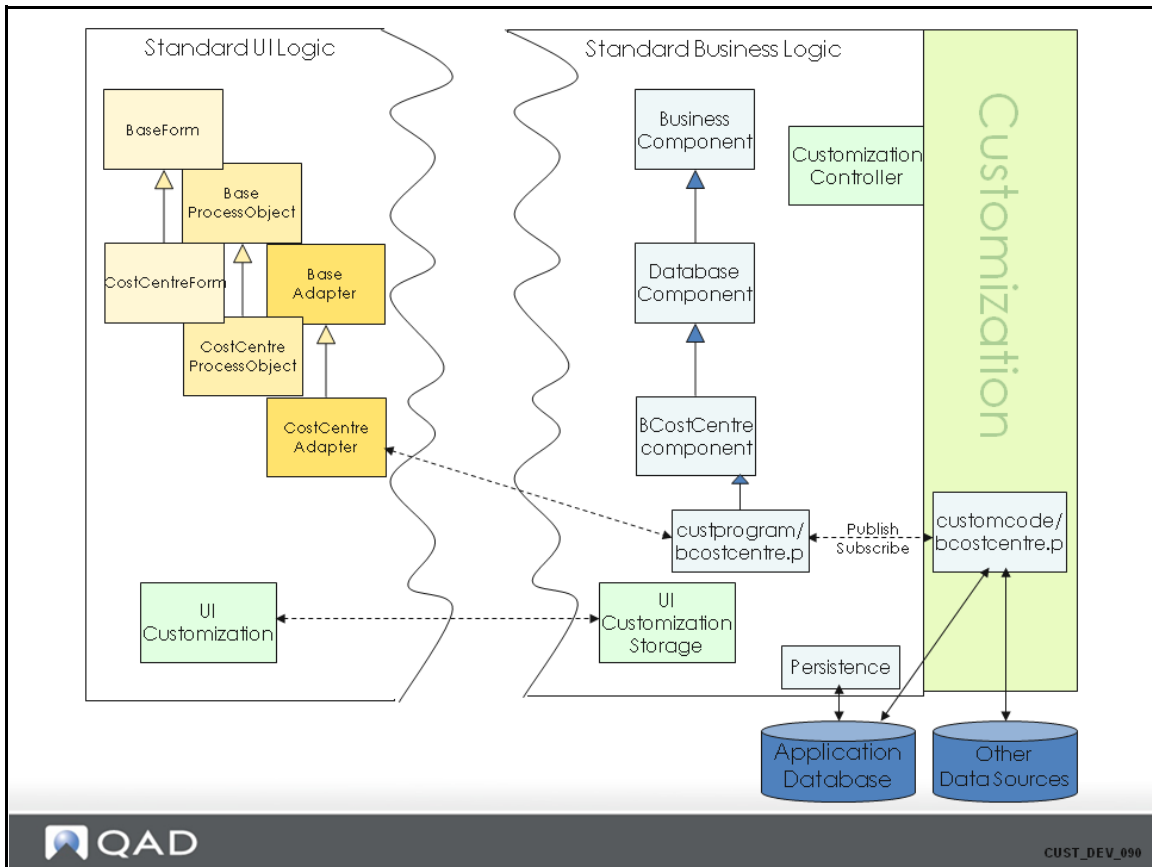
CUST\_DEV\_070

The object dataset is accessed through the temp-tables that make up the object dataset. In this example, we use the table tCountry. The newly created buffer is available, so no find/for each is required.

## Customization Diagram



Customization Diagram



The ComponentPool session super-procedure starts the customization controller component. At startup (typically startup of the AppServer agent), it identifies all available customizations by searching in the PROPATH for a folder named “customcode”. It keeps in memory which components are customized (for which components custom code is found).

ComponentPool uses this information to automatically start the customization code together with the other component code.

## Customization - Development Steps

### Customization – Development Steps

Approach is similar to standard development:

1. Analysis
2. Design
3. Implementation
4. Test
5. Deploy



CUST\_DEV\_100

## Customization - Development Steps

## Customization – Development Steps

### Step 1: Analysis

- Describe the required Customizations on a functional level
- Examples:
  - Which validations must be added?
  - Which calculations must be done differently?
  - Which new data elements should be added?
  - How should a screen display the new data?



CUST\_DEV\_110

In this step, you should look at the function in the application you want to customize, and describe exactly what you intend to change.

This can be a very high-level description and does not need to go into technical details. At this stage, the person making the analysis is purely focused on functionality and not on whether the customization is feasible with non-intrusive customization or requires intrusive customization.

## Customization - Development Steps

## Customization – Development Steps

### Step 2: Design

- Read the technical documentation on the object model of the QAD EE Financials (the documentation is shipped with the product)
- Locate the business component(s) that need to customization to achieve the goals set in the analysis
- Locate the base Customization code and the templates needed for the implementation
- Add custom tables and fields to the database
- Look for access to any external system required for the Customization



CUST\_DEV\_120

In this step, the designer uses the functional description to further detail the work required to implement the customization. In this phase, the requested function is considered in terms of the technical possibilities of non-intrusive customization. If the requested functionality can be implemented, the actions mentioned above provide the right starting point for the developer to implement the customization.

The design step is the most difficult. This requires comprehensive knowledge of the application, the data model, the application structure, and the patterns. In complex cases, QAD services might need to be involved to explain how the function works and how a certain customization can be achieved.

## Customization - Development Steps

## Customization – Development Steps

### Step 3: Implementation

- For each component requiring an extension, copy the template for the Customizations from the `customization/template` folder to the `develop` folder
- Uncomment the procedures that need to contain the Customization code and fill in the code



CUST\_DEV\_130

The customization in the training environment is located at: `/dr01/qadapps/qea/fin/Customization`.

## BL Customizations

### BL Customizations

#### Set Up

- The `<QADAPPSINSTALL>/fin/Customizations` folder contains the Customization template procedures
- To maintain customized code, use a separate folder (`<CUST>`) with the following subfolders:
  - `customcode`
  - `work`
- The `<CUST>` folder must be in the `PROPATH` of the Financials AppServer
- You must trim the Financials AppServer after each change to the custom code



CUST\_DEV\_140

In the new separate folder `<CUST>`, the `customcode` subfolder must contain only the compiled customization procedures.

The Financials AppServer needs to be restarted after the `PROPATH` is set.

The changes to customization procedures only take effect after you trim the Financials AppServer.

You can trim the Financials AppServer using the following commands from the Linux prompt. The example assumes the `DLC` variable is set properly, `PATH` points to the `$DLC/bin` folder, and it trims 10 AppServer agents:

```
asbman -name qadfinlive -trim 10
```

You can also trim the Financials AppServer using the “Status” tab on the “System Admin” page in the training environment. Simply click on the “..qadfinlive.” label. This trims 10 AppServer agents.

## BL Customizations

## BL Customizations

Set up in OpenEdge Architect:

- Start from an empty workspace
- Create an empty project
- Set project properties:
  - Startup parameters
  - Build destination (<propath>/customcode)
  - PROPATH (contains definition + template folder)
  - Database connections (from `server.xml`)



CUST\_DEV\_150

### Startup parameters

```
-s 512 -mmax 4000 -inp 32000 -tok 20000 -TB 30 -TM 30 -Bt 3000 -
errorstack -cpinternal utf-8 -cpstream utf-8 -cpcoll ICU-UCA -
cpcase basic
```

### Build destination

```
Z:\training\Customization_component_based\customcode
```

PROPATH (PROPATH for compilation does not match the PROPATH for runtime)

```
Z:\qadapps\qea\fin\Customization
```

### Database connections

physical name =	qaddb	traindb
logical name =	qaddb	traindb
host =	qaddemoqaddemo	
service =	7744	7845

## Customization Exercise - Set Up

### Customization Exercise – Set Up

Verify the training environment is set up correctly:

- Locate the  
`<QADAPPSINSTALL>/fin/Customization`  
folder
- Create subfolders in `<CUST>` folder:  
`/dr01/training/Customization_component_based` (or  
`z:\training\Customization_component_based on windows`)
  - `customcode`
  - `work`
- Check the PROPATH of the Financials AppServer in  
`/dr01/progress/dlc101c/properties/ubroker.properties`



CUST\_DEV\_160

## BL Customizations

### BL Customizations

#### Initial development

In the `work` subfolder:

- Copy the template `.p` file from `fin/Customization/template`  
     Always keep the same name (for example, `bposting.p`)

- Perform the required changes in the `.p` file
  - Keep the first three lines:

```
/* Customization code for component BCountry */
&scoped-define class-version 39.0
{ definition/bcountry.i }
```

- Remove all sections for methods that do not get customized
  - For the method to be customized, uncomment the *before* or *after* hook and add code
- Best to work with a `build.p` file that automatically compiles all Customization `.p` files into the right location (the `customcode` subfolder)



CUST\_DEV\_170

We remove all unused methods from the copied procedure to make comparing this file with the template after an upgrade easier.

The build file should contain following blocks of code:

1. Set the `PROPATH`. The `PROPATH` should point to the current folder (“work”) and the parent folder of the folder where the template was copied from (`fin/Customization`).
2. Connect the database if the custom code contains direct table access (for each `/ find / query`) and if the database is not connected yet.
3. Compile statements for all custom code for the different components.

For example:

```
compile bcountry.p save into ../customcode.
```

```
compile bposting.p save into ../customcode.
```

**Note** OpenEdge Architect compiles code automatically, so there is no longer need for a `build.p`.

## BL Customizations

# BL Customizations

## Build Procedure Example

```

/* Specify the propath first */
ASSIGN PROPATH = "Z:\qadapps\qea\fin\Customization"

/* For direct access to the database tables */
IF NOT CONNECTED("qaddb") THEN CONNECT -H qaddemo -S7744 qaddb

/* Compile all customized code */
COMPILE Z:\training\Customization_component_based\work\bcostcentre.p
  SAVE INTO Z:\training\Customization_component_based\customcode
COMPILE Z:\training\Customization_component_based\work\bcountry.p
  SAVE INTO Z:\training\Customization_component_based\customcode

```



CUST\_DEV\_180

The section with the connection to the database is optional. It is only required when direct table access is done straight from the custom code.

In the above example, it connects to the central qad applications database. If another database were used, the connection also must be made at runtime. To specify this connection, you update the central `server.xml` file.

**BL Customizations**

## BL Customizations

### Deploy Customizations

- Make sure the compiled `.x` code for the Customization is placed in the correct folder (the `customcode` folder)
- Trim the Financials AppServer to make sure the changes are picked up correctly
- Use the System Monitor function to verify the Customization is active



CUST\_DEV\_190

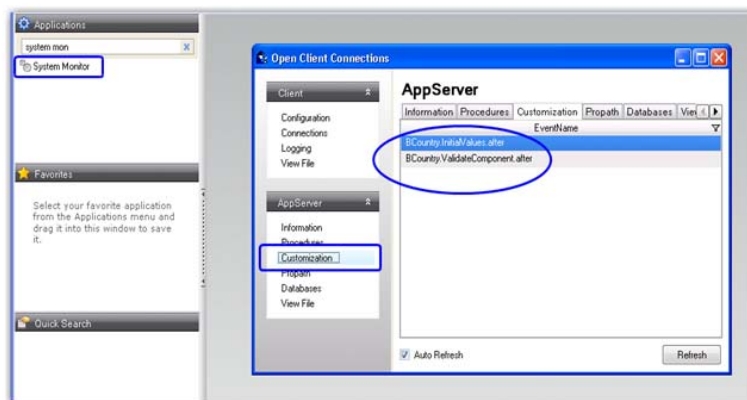
When activating a new customization or making sure changes in existing customization code are picked up, the Financials AppServer needs to be trimmed. This is necessary because the business layer caches component code in memory. It keeps procedures running persistently for best performance. By trimming the AppServer, all active AppServer agents are restarted. This triggers the customization controller to look in the `customcode` folder again, and make sure the right customization code is started the next time a business component gets instantiated.

## BL Customizations

### BL Customizations

Use System Monitor to check:

- If Customization is active
- Which hooks are implemented by the backend Customization



The system monitor reveals a lot of information about the business layer running on the back-end AppServer.

One of the interesting pieces of information is the list of customized hooks on the business layer. You can retrieve this by selecting “Customization” in the AppServer section.

**BL Customizations**

## BL Customizations

### Maintain Custom Code

- Change the custom code:
  - In existing methods
  - For new Customizations in the same business component, copy the proper section from the template code file to the custom code file
  - For new Customizations in another business component, repeat the steps for the initial custom coding (copy template, change code)
- Execute the build script to recompile the custom code
- Trim the Financials AppServer



CUST\_DEV\_210

The customization developer has the ability to add code using custom code, but it is not possible to override standard logic.

## BL Customizations

### BL Customizations

#### Maintain Custom Code

In case of a version upgrade or patch install:

- Perform a source code comparison between the most recent Customization template code and the code in the `customcode` folder
- If the code differs only on the line of the version number and for methods for which no Customization is in place, just change the version number to the correct value in the `customcode` file
- Else, repeat the initial steps to create custom code for the component, and copy the custom code from the old to the new implementation

In this case, make sure the required parameters are used properly

- Always recompile ALL customized code



CUST\_DEV\_220

## BL Customizations

## BL Customizations

### Versioning

- Every standard component has a version number consisting of a major and a minor number
- The component version is associated with the template for the Customization code from the moment the component is released
- The Customization controller checks both numbers and reports inconsistencies as warnings at runtime
- Version numbers can change with upgrades or patch installs



CUST\_DEV\_230

The mechanism built into the application ensures that every time a customization is started, the version is checked against the version of the standard component code that is currently used in memory. In a normal situation, this is not an issue. The system reports (in the form of a warning) as soon as there is a version mismatch. This is not only at runtime, but also at compile time. Compiler pre-processed code checks these numbers. So, if you try to compile a customization that was originally copied from the customization folder for an older version of the component, the compiler shows the exact version numbers and indicates that there is a conflict. However, this does not stop compilation, but we advise you to review the code before updating the version number and recompiling.

Note that after every upgrade / patch install, the customer must recompile the custom code. The `PROPATH` must point to the standard components, which may have a different version number. When everything compiles correctly, there is no error, and no further action is needed.

If the compiler displays the version conflict warnings, it is possible that there is still no issue, and that changes in the standard code do not impact the correct working of the custom code. In this case, the proposed procedure is to compare the custom code with the latest version of the template for the same component (using `file-compare`). Go through the differences and check only what has changed for the customized method(s). In effect, this is a basic check of the parameters of the method. In most cases, the changes do not impact custom code, and you should update the version number specified in the custom code (`&scoped-define class-version`) to the correct version, and recompile.

**Note** The component major version number indicates public interface changes (each time parameters in public methods change, the major version is incremented by 1). The component minor version number indicates internal changes in the component (which can be code changes, but also parameter changes to non-public methods). When one of the two numbers change, you must go through the process described above.

**BL Customizations**

## BL Customizations

Most important Customization methods:

- InitialValues
- Calculate
- ValidateComponent
- AdditionalUpdates
- PreSave/PostSave
- GetBusinessFields



CUST\_DEV\_240

## BL Customizations - InitialValues

### BL Customizations

#### InitialValues

- Always executed as part of the pattern for a new record in the object dataset. The name of the table is passed as input parameter `icTableName`.
- Method is executed right after the infrastructure code is executed to create a new record in the object dataset
- The record that is being created is in scope
- Best to implement the `InitialValues.After` method



CUST\_DEV\_250

The `InitialValues()` method is executed after the infrastructure code filled in the fixed fields `tc_rowid`, `tc_parentrowid` and `tc_status`. In the standard code, the identifier field is also filled in. For example `tCountry.Country_ID`. An internal sequence is used to determine the value.

**Note** Developers never use a direct create statement on class tables. Instead, they run method `AddDetailLine`. This method runs method `InitialValues` for initializing the newly created record. When adding a record in a class table in custom code, we also recommended that you use this method.

## BL Customizations - InitialValues

## BL Customizations

### InitialValues - Example

```
/*  
Procedure: InitialValues  
Description:  
Add code here to initialize the calculated fields of a 'new' record (= a record that must be created in the  
application database) in a class temp-table.  
Parameters:  
input icTableName  
    (Name of the database table of which a record is created in the class temp-table.)  
output oiReturnStatus  
    ()  
*/  
  
PROCEDURE BCountry.InitialValues.after:  
    if t_parameter.icTableName = "Country"  
    then assign tCountry.CustomInteger0 = 1000000.  
END PROCEDURE.
```

## Hands-on Exercise (3)

### Hands-on Exercise (3)



#### Specify Default Value for Customer Category (1)

## BL Customizations - Calculate

### BL Customizations

#### Calculate

- Executed as part of the DataLoad of an object. This is typically done when the application user selects an object from the browse and chooses Modify, Delete, or View
- Records are not automatically in scope. The programmer needs to use a for each statement
- Best is to implement `calculate.After()`, because then the standard calculated fields are already known



CUST\_DEV\_280

Each time a Financials object is loaded for modification, the application goes through the DataLoad pattern. The first thing that happens is the data is read from the database. Right after that, the Calculate method is executed. The intent of this method is to “calculate” the data for the calculated fields in the object dataset.

Typically, a customization developer could use this method to get data from external systems and force it in the official object dataset of the object that is being loaded.

Multiple objects, main table records, and child table records can be loaded during the data load. Therefore, the programmer needs to use for-each statements to loop over the records in the object dataset.

## BL Customizations - Calculate

## BL Customizations


### Calculate - Example

```

/*
Procedure: Calculate
Description:
Add code here to initialize the calculated fields of the class temp-tables after loading existing records
from the application database.
Parameters:
output oiReturnStatus
  ()
*/

PROCEDURE BCountry.Calculate.after:
  FOR EACH tCountry WHERE tCountry.CountryIsEUCountry :
    ASSIGN tCountry.CustomShort9 =
      ENTRY(LOOKUP(tCountry.CountryDescription,
        "Netherlands,Greece,Germany,Switzerland,Belgium,United
        Kingdom,Spain,France,Italy"),
        "Amsterdam,Athens,Berlin,Bern,Brussels,London,Madrid,Paris,Rome").
    END.
END PROCEDURE.

```


CUST\_DEV\_290

Save the capital of some European countries into CustomShort9.

## Hands-on Exercise (4)

### Hands-on Exercise (4)



### Specify Default Value for Customer Category (2)



CUST\_DEV\_300

## BL Customizations - ValidateComponent

### BL Customizations

#### ValidateComponent

- Executed every time object data needs to be validated before it can be written to the database
- Code in the method should specifically use the `t_s` variant of the class tables
- In case of a validation error, the `oiReturnStatus` needs to be given the value -1
  - Use a call to `SetMessage` to pass the correct error message
- The code should take into account that there might be more than one record in the main class table
- Standard validation cannot be overridden
- In case the `ValidateComponent.Before()` assigns `oiReturnStatus` to -1, the standard execution flow continues



CUST\_DEV\_310

ValidateComponent is the central validation method for each component. It is typically intended to contain all object-wide validation. Besides the ValidateComponent, very specific field-level validations might also be in place.

This method is typically executed when data is sent from the UI client to the server. The first step the BL takes is to validate the data. The data that is not yet validated is available in the “S” object dataset (this means records in the `t_s` temp-tables). For example, a developer needs to use the data in `t_sCountry` if he/she wants to validate incoming changed data for a country object.

After the data has been validated, the system updates the data in the official object dataset (O buffers) with the validated changed data, executes `AdditionalUpdates`, and performs a `DataSave`.

ValidateComponent needs to validate everything. So, if data for more than one object is loaded in the object dataset, the code should validate everything. This means that the code should have a for-each statement to go over all records in the class tables.

## BL Customizations - ValidateComponent

# BL Customizations

## ValidateComponent - Example

```
/*  
Procedure: ValidateComponent  
Description: ...  
*/  
PROCEDURE BCountry.ValidateComponent.After:  
  DEFINE VARIABLE viReturn AS INTEGER NO-UNDO.  
  FOR EACH t_sCountry:  
    IF t_sCountry.CustomInteger0 = 0 THEN DO:  
      ASSIGN t_parameter.oiReturnStatus = -102.  
      run SetMessage ('The number of habitants ($1) must be greater than 0.',  
        STRING(t_sCountry.CustomInteger0), 't_sCountry.CustomInteger0',  
        string(t_sCountry.CustomInteger0),  
        'E', 3, t_sCountry.tc_Rowid, 'CUSTOM-001', '', '', '', output viReturn).  
      RETURN.  
    END.  
  END.  
END PROCEDURE.
```

## Hands-on Exercise (5)

### Hands-on Exercise (5)



#### Validation on State Tax Field

## BL Customizations - Additional Updates

### BL Customizations

#### `AdditionalUpdates`

- Executed during the object update pattern, after the `ValidateComponent` and before the `DataSave`
- This method typically contains code to apply changes to other objects as a consequence of a change to this object
- At the moment of the execution of this method, no physical transaction is active

## BL Customizations - Additional Updates

## BL Customizations

### AdditionalUpdates - Example

```
/*  
Procedure: AdditionalUpdates  
Description:  
This method is part of the SetPublicTables flow.  
When executed, data in the input class tables (prefix t_s) is validated and found correct, and copied into  
the class tables (prefix t_o).  
This method can be extended to do updates that do not require a validation or that involve running business  
methods of other business classes. These classes should be started with ADD-TO-TRANSACTION = 'true'. Also  
make sure to add these classes in method StopExternalInstances.  
Parameters:  
output oiReturnStatus  
  ()  
*/  
  
PROCEDURE BCountry.AdditionalUpdates.after:  
  FOR EACH tCountry where tCountry.tc_status <> "":  
    output to /dr01/logging/country_log.log append.  
    put unformatted string(now) tCountry.CountryCode skip.  
    output close.  
  END.  
END PROCEDURE.
```



CUST\_DEV\_350

## BL Customizations - PreSave and PostSave

### BL Customizations

#### PreSave and PostSave

- These methods are executed immediately before and after the object data is written to the database
- Executed during the physical transaction for updating the object data in the database

If the custom code is assigning `oiReturnStatus` with a negative value, the entire transaction is rolled back



CUST\_DEV\_360

These methods are typically used to update records in the database and the update needs to be in the same physical transaction. An example is the update of custom tables in the database when the `tCustomTable*` tables are used in the object dataset.

## BL Customizations - PostSave

# BL Customizations

## PostSave - Example

```
/*  
Procedure: PostSave  
Description:  
Actions to take after writing current instance to the database, and before final commit of the transaction.  
Use the field tc_status to test the status of the updated records:  
' ' = unchanged  
'N' = new  
'C' = changed  
'D' = deleted  
Parameters:  
output oiReturnStatus  
  ()  
*/  
  
PROCEDURE BCountry.PostSave.after:  
  for each tCountry where tCountry.tc_status = "D":  
    create customlog.  
    assign customlog.tablename = "country"  
           customlog.tablekey = tcountry.countrycode  
           customlog.tableaction = "delete".  
  end.  
END PROCEDURE.
```

## Hands-on Exercise (6)

# Hands-on Exercise (6)



**Replicate Customer Category**

## BL Customizations - GetBusinessFields

### BL Customizations

`GetBusinessFields`

- A client executes this method to retrieve meta data about information that can be used on the object form
- The information for the business fields is returned in temp-table `tBusinessFields`
- Not only used to retrieve information about business fields for the object form, but also for queries and reports



CUST\_DEV\_390

## BL Customizations - Business Fields Mechanism

### BL Customizations

#### Business Fields Mechanism

The dataset (with one temp table) `tBusinessFields`, which is returned by the `GetBusinessFields()` method, contains the following fields:

<code>icColumnLabel</code>	character	Column label
<code>icControlType</code>	character	Control type
<code>icDataType</code>	character	Can be any of the following values: c,d,i,l (resp. character, decimal, date, integer or logical)
<code>icDisabledForActivities</code>	character	business field is disabled for any of these activity codes (comma separated list of activity codes)
<code>icDisplayFormat</code>	character	Display format (progress display format like "%40")
<code>icFdDescription</code>	character	Description of the field (currently not used by the UI)
<code>icFdFieldName</code>	character	The name of the business field.
<code>icFdFieldType</code>	character	can be "B", "F", "C" (resp. business, filter or custom)
<code>icFilterOperators</code>	character	This is the list of possible filter operators. This is used in case of "F" typed business fields.
<code>icHiddenForActivities</code>	character	business field is hidden for any of these activity codes (comma separated list of activity codes)
<code>icLookupFilterField</code>	character	This is the filter field in the query that is referenced by <code>icLookupQuery</code> . It receives the value of the current business field upon calling the lookup with the <code>icLookupQuery</code> reference.
<code>icLookupQuery</code>	character	The query that should be used when the lookup is chosen for the business field.
<code>icLookupReturnField</code>	character	This is the business field from the result set from the <code>icLookupQuery</code> for which the value is returned and filled in into the current business field.
<code>icRelatedObject</code>	character	Business component to which the business field refers. This activates the "goto" link.
<code>icSideLabel</code>	character	side label
<code>icStoredSearch_ID</code>	integer	Linked stored search (this is only available for user-defined business fields)
<code>icValidationMask</code>	character	The name of the validation mask for the field (this is not used by the UI at the moment)
<code>icValueList</code>	character	The list of discrete values the business field can have. This is used to populate the combo-box.
<code>icInitialValue</code>	character	The initial value of the business field. Only used in the "create" activity.
<code>icSequence</code>	integer	A sequence of the business field. Only used as unique key in the <code>tBusinessFields</code> temp table.



CUST\_DEV\_400

All of the information represented on UI forms is managed using the concept of business fields. The business logic provides all of the information represented by UI fields. The UI only uses this information to represent and reformat the fields. Typically, the information provided is the format, the label, the lookup query, the control type, and the list of possible values.

This information is provided by the backend through the method `GetBusinessFields`. Typically, the UI calls `session.GetBusinessFields()`, which gets a "reference" as input.

The reference is typically a business object name like for example, "BJournalEntry", "BCreditor". This reference can also point to a query.

`GetBusinessFields` returns the "meta data" for the query. This meta data consists of a description of the filter fields of the query and the columns in the query resultset.

`session.GetBusinessFields()` is a wrapper that calls the component-specific `GetBusinessFields()` method. It is really this method that contains the code that is executed to retrieve the business field information.

For business components, the standard component code uses generated code from the datamodel (for the normal fields). This is combined with code generated based on the extended information of the object dataset (for example, calculated fields). For queries, the code is generated using information about the query parameters, the joined tables in the query, and the output dataset.

For a business component, all business fields are of type "B", meaning a real business field.

For a query, `GetBusinessFields()` is called from within the browse or lookup. There are three clear types of business field returned. Type "F" fields are the filter business fields. These fields are used in the browse / lookup in the filter area. Type "B" fields are the fields that are available in the result set of the query, so these fields can be used on the result grid of the browse / lookup. Type "C" fields are the custom fields (or user-defined fields) available in the result set of the query.

It should be clear that the customization developer can change the UIs, or at least how fields are shown on the UI, by changing the behavior of the `GetBusinessFields()` method.

`tcControlType` can be any of these values: `NumericDecimal` / `NumericInt` / `Bool` / `DateTime` / `TextBox` / `ComboBox`

`tcValueList` is a `chr(2)` separated list of a display value and an internal value so, `<label1> chr(2) <value1> chr(2) <label2> chr(2) <value2> chr(2) ...`

## BL Customizations - GetBusinessFields

# BL Customizations

## GetBusinessFields - Example

```

/* Procedure: GetBusinessFields ... */
PROCEDURE BCountry.GetBusinessFields.after:
  DEFINE VARIABLE vcList AS CHARACTER NO-UNDO.
  DEFINE VARIABLE vcItemLabel AS CHARACTER NO-UNDO.
  DEFINE VARIABLE vcItemCode AS CHARACTER NO-UNDO.
  FIND FIRST tBusinessFields WHERE
    tBusinessFields.tcFcFieldName = "Country.tcCustomCombo2" NO-ERROR.
  IF AVAILABLE tBusinessFields THEN DO:
    INPUT FROM /dr01/lists/countrytypes.txt.
    REPEAT:
      IMPORT vcItemCode vcItemLabel.
      ASSIGN vcList = (IF vcList = ""
        THEN vcItemLabel + CHR(2) + vcItemCode
        ELSE vcList + CHR(2) + vcItemLabel + CHR(2) + vcItemCode).
    END.
    ASSIGN tBusinessFields.tcValueList = vcList.
  INPUT CLOSE.
  END.
END PROCEDURE.

```



CUST\_DEV\_410

## Hands-on Exercise (7)

# Hands-on Exercise (7)



## Change the Field Format of a Business Field



CUST\_DEV\_420

## BL Customizations - GetTranslation

### BL Customizations

#### GetTranslation

- The business code executes this method to retrieve the correct translation of a string
- You usually do not customize this method, but use it from your Customization
- Note: This method is not an internal procedure, but a function. Therefore, do not use run, use dynamic-function



CUST\_DEV\_430

## BL Customizations - GetTranslation

## BL Customizations

### GetTranslation - Example

```
PROCEDURE BCountry.ValidateComponent.before:
  DEFINE VARIABLE viReturn AS INTEGER NO-UNDO.
  DEFINE VARIABLE vcMessage AS CHARACTER NO-UNDO.
  FOR EACH t_sCountry:
    IF t_sCountry.CustomInteger0 = 0 THEN DO:
      ASSIGN t_parameter.oiReturnStatus = -102.
      ASSIGN vcMessage = DYNAMIC-FUNCTION
        ( 'GetTranslation' in {&TARGETPROCEDURE}, 12345, PCode',
          'The number of habitants ($1) must be greater than 0.').
      run SetMessage
        (vcMessage,
         STRING(t_sCountry.CustomInteger0),
         't_sCountry.CustomInteger0',
         string(t_sCountry.CustomInteger0),
         'E', 3, t_sCountry.tc_Rowid,
         'CUSTOM-001', '', '', '', output viReturn).
    RETURN.
  END.
END.
END PROCEDURE.
```



CUST\_DEV\_440

## Translations

### Translations

blf.resx

 Strings ▾ Add Resource ▾ Remove Resource ▾

Name	Value	Comment
4924	Login ID for this Daemon	Login ID for this Daemon 30 side label for database table field blfdb.fcDaemon.DaemonLogin BE
4925	Daemon Login	Daemon Login 30 column label for database table field blfdb.fcDaemon.DaemonLogin BBaseDae
4926	Daemon Log Level	Daemon Log Level 30 column label for database table field blfdb.fcDaemon.DaemonLogLevel BE
4928	Daemon Name	Daemon Name 20 label for parameter icDaemonName of query BBaseDaemon.DaemonInfo BBase
493	You must enter the year.	You must enter the year. 200 message from business method BNumber.ApiSetFreeNumbersVali
4930	Password for this Daemon	Password for this Daemon 30 side label for database table field blfdb.fcDaemon.DaemonPassw
4931	Password	Password 30 column label for database table field blfdb.fcDaemon.DaemonPassword BBaseDae
4932	Keep Processed Items	Keep Processed Items 30 side label for database table field blfdb.fcDaemon.DaemonKeepProce
4933	Keep OK	Keep OK 8 column label for database table field blfdb.fcDaemon.DaemonKeepProcessedOKItem
4934	Number Treated in One Run	Number Treated in One Run 30 side label for database table field blfdb.fcDaemon.DaemonNrOf
4935	No in One Run	No in One Run 20 column label for database table field blfdb.fcDaemon.DaemonNrOfRequestsIn
4938	OS Command String	OS Command String 30 column label for database table field blfdb.fcDaemon.DaemonOsComma
494	You must enter the type.	You must enter the type. 200 message from business method BNumber.ApiSetFreeNumbersVali
4940	Last Start Time	Last Start Time 20 label for query table field DaemonLastStartTimeQInfoCalc of query BBaseDa
4942	Last End Time	Last End Time 20 label for query table field DaemonLastEndTimeQInfoCalc of query BBaseDaem
4944	Activity	Activity 20 Preprocessor RESOURCE-TYPE-ACTIVITY-TR BBusinessComponent
494752128	Daemon status different from	Daemon status different from 28 label for parameter icDaemonStatusDiffersFrom of query BBase
4948	Creation Date	Creation Date 20 label for query table field LastModifiedDate of query B500ViolationRule1.Sele
495	The numbering record with year &1 and type &2 already exists.	The numbering record with year &1 and type &2 already exists. 200 message from business me

CUST\_DEV\_450

Translatable strings are maintained in a resource file. A resource file is an XML-formatted text file. One resource file is created per project.

Inside a resource file, a unique number identifies each string. Each resource file is then translated into each required language and renamed as `<project>.<language>.resx` like `f.e.qadfin.fr.resx`.

## Translations

The screenshot displays two windows from the QAD system interface. The top window, titled 'System Synchronize', shows a table with the following data:

Topic Name	Selected	Update Information
Superuser Role	<input type="checkbox"/>	
System	<input type="checkbox"/>	
Translations	<input checked="" type="checkbox"/>	
Users	<input type="checkbox"/>	

The bottom window, titled 'Language Browse for View', shows a search interface with the following details:

- Search criteria: Installed Language equals yes
- Viewing 1 - 2 of 2 records
- Records per page: 100
- Table columns: Language, Description, Active, Installed Lang

Language	Description	Active	Installed Lang
fr	french	yes	yes
us	english (U.S.)	yes	yes

The QAD logo is visible in the bottom left corner, and the text 'CUST\_DEV\_460' is in the bottom right corner of the screenshot area.

Translation resource files are loaded in the system synchronize function. Translations are loaded for all installed language codes.

## Translations

## Translations

How to define a translatable string

1. Reuse an existing string
2. Create a string in an existing resource file
3. Create a resource file

```
...  
<resheader name="dump time">  
  <value>07:48:36</value>  
</resheader>  
  <resheader name="project">  
    <value>QADFIN</value>  
  </resheader>  
  <resheader name="ancestor projects">  
    <value>BLF</value>  
  </resheader>  
<data name="1">  
...  
...
```



CUST\_DEV\_470

When modifying an existing resource file, you must redo your changes whenever a new version of the application is installed. Creating a resource file is upgrade-safe. You can use any editor you like to maintain resource files, like Visual Studio, or even Notepad. When creating a resource file, you can make a copy of an existing resource file and start from there. Make sure that the project codes are set correctly. The project must match the resource file name. Ancestor project must be "QADFIN" (as your resource file replaces the QADFIN resource file).

## Translations

# Translations

When creating a resource file:

```
<?xml version="1.0" encoding="utf-8" ?>  
<ProjectCode>qadfin</ProjectCode>
```



CUST\_DEV\_480

The application must know where to find your resource file. This is indicated in the file `xml/projectcode.xml`.

## Hands-on Exercise (8)

### Hands-on Exercise (8)



**Change the Label of a Business Field**



Chapter 3

# **Non-intrusive Customization: Supporting Tools**

## BL Customization - Tools

### BL Customizations - Tools

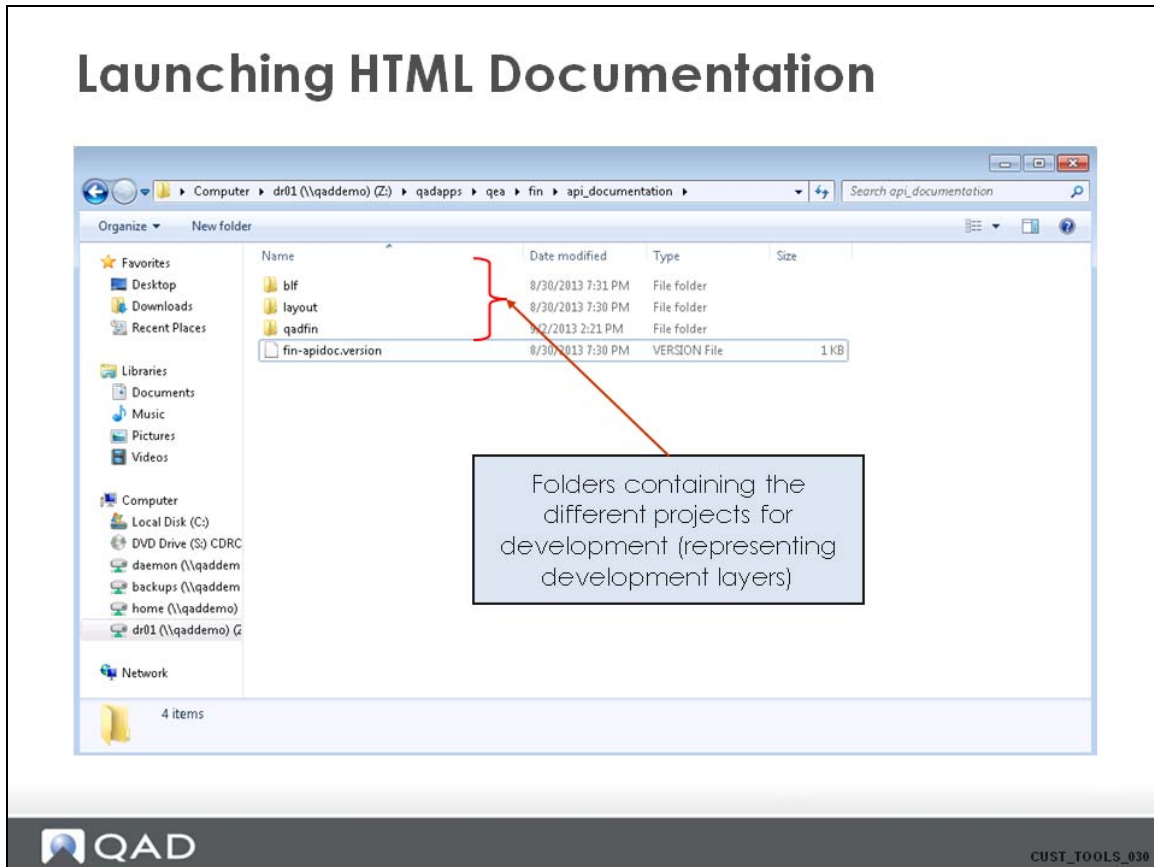
- Documentation in HTML format
- HTML pages containing information about projects and components in projects
- Structure:
  - Separate folder for each project. Accessed through `index.htm`
  - Separate sub-folder for each business component. Accessed through `index.htm`.
  - Each method has a separate HTML file (`m-<name>.htm`)
  - Each query has a separate HTML file (`q-<name>.htm`)
  - Each structured data item (temp table or dataset) has a separate HTML file (`d-<name>.htm`)



CUST\_TOOLS\_020

The HTML documentation represents the documented class model for the application business layer. The documentation is written by developers and generated in HTML by the development tool that they use (Component Builder tool).

## Launching HTML Documentation

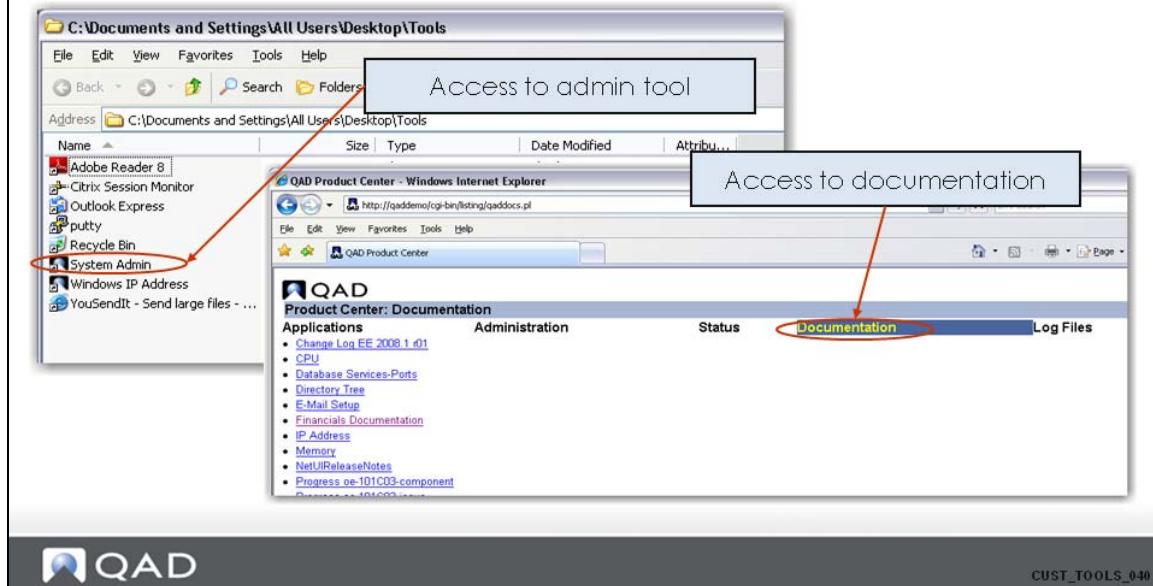


Launching the documentation from Windows.

## Launching HTML Documentation

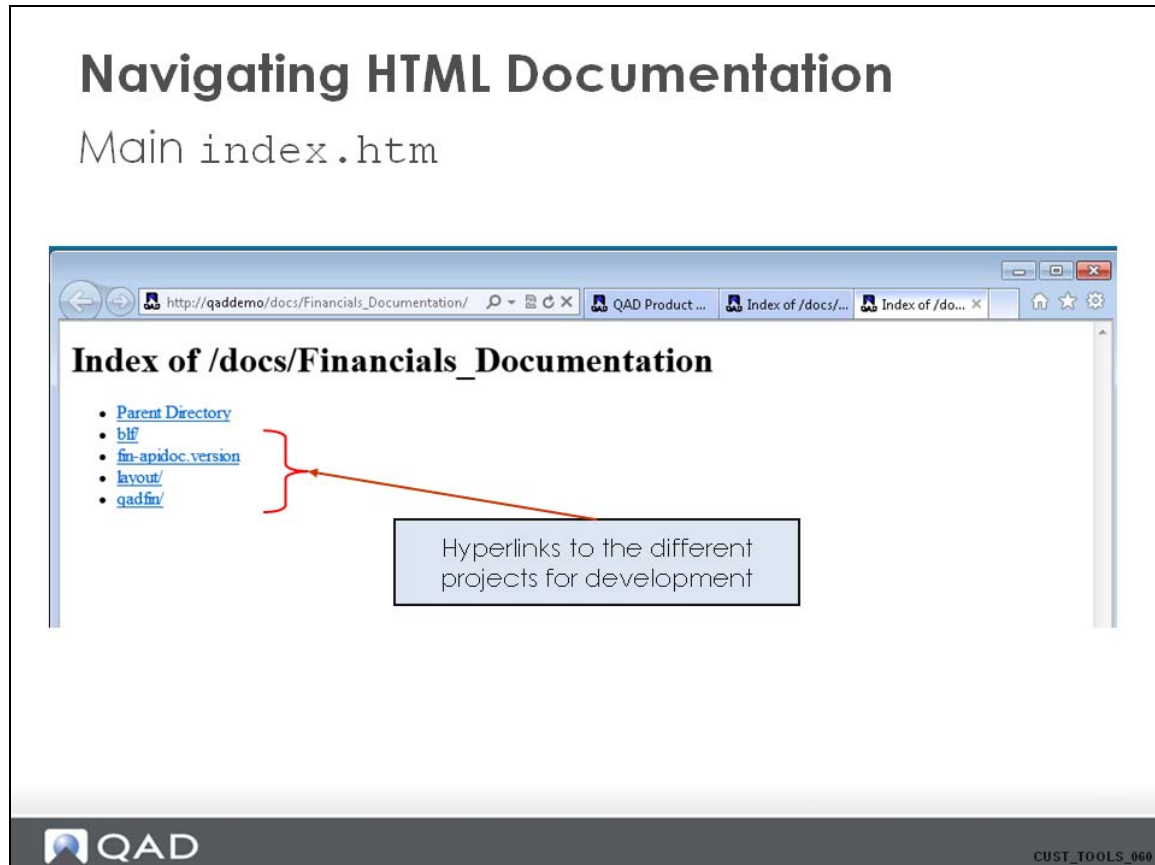
# Launching HTML Documentation

- Access via System Admin under Tools folder on desktop
- Choose the Documentation tab



Launching the documentation from the admin tools.

## Navigating HTML Documentation



The main page for HTML documentation contains a hyperlink to each of the development projects that are used to create the final application code.

FC and QADFC are foundation-level projects. FC (Foundation Classes) contains a mixture of abstract classes and technical classes that are required to support all application patterns. This way, the FC project contains the pure technical implementation of the application infrastructure. QADFC (Qad Foundation Classes) is dependent on FC and contains functional implementation classes for mechanisms like role-based security, domains, entities, and so on. This way, the QADFC project contains the more functional part of the application infrastructure.

QADFIN is the application project. This is the biggest project and contains all of the business components that are required for the Financials functionality in the QAD application.

## Navigating HTML Documentation

# Navigating HTML Documentation

## Project QadFinancials index.htm

The screenshot shows the QAD Enterprise Financials Documentation page for project QadFinancials. The page includes the following sections:

- project QadFinancials**: inherited from BLF, version BLF 2010.1, namespaces QAD.QADFIN\_IF + QAD.QADFIN\_BO.
- Business components**:
  - [BAccountingInterfaceFilters](#)
  - [BAdmMemInformation](#)
  - [BDInvoiceMultCy](#)
  - [BJournalEntryMultCy](#)
  - [BMultCyProcessor](#)
  - [BOpenBalance](#)
  - [Session](#)
- Budgeting**:
  - [BBudget](#)
  - [BBudgetGroup](#)
  - [BBudget\\_nik](#)
  - [BBudget\\_nikAcc](#)
  - [BBudget\\_nikAccDet](#)
  - [BBudget\\_nikActual](#)

Annotations on the right side of the screenshot explain the structure:

- Dependent = Inherits from**: Points to the 'inherited from BLF' text.
- List of business components...**: Points to the 'Business components' section.
- ... categorized in (Business components, Report components and Technical components) ...**: Points to the list of business components.
- ... organized per business area**: Points to the 'Budgeting' section.

The main page for project documentation contains the following:

- Dependency information (projects on which this project is dependent).
- Namespaces (the namespaces used for the components in this project in .NET C#).
- A list of business components, categorized using the business areas (such as Budgeting, Business Relation, and General Ledger).
- A list of report components, categorized using the business areas.
- A list of technical components, categorized using the business areas.
- ...

The components are all hyperlinked and you can use them to drill down for detailed component information.

## Navigating HTML Documentation

# Navigating HTML Documentation

## Project QadFinancials index.htm

The screenshot shows a web browser window with the URL [http://qaddemo/docs/Financials\\_Documentation/qi](http://qaddemo/docs/Financials_Documentation/qi). The page content is as follows:

```

Find: index
  • LDebug
  • TDocumentLink
  • TExchangeRate
  • TFRWTreeView
  • TIntegrationUtilities
  • TJournalRole
  • TMapEntitySecurity
  • TReportStrings
  • TReportSwitchMenu
  • TSetGLSequenceNumber
  • TSetStatutoryCurrency
  • TTSM

Includes
  • BGL_ApGL_TransExtCalculation.i
  • BudgetCalculateMeasureByPeriod.i
  • ProcessFilter.i

Preprocessors
  {&ACTIVITYCODETYPE-EMPLOYEE} EMPLOYEE.U
  {&ACTIVITYCODETYPE-EMPLOYEE-TR} trim(#T-'Employee'-40(3963))-7#)
  {&ACTIVITYCODETYPE-ITEM} ITEM.U
  {&ACTIVITYCODETYPE-ITEM-TR} trim(#T-'Item'-40(4275))-7#)
  {&ACTIVITYCODETYPES} {&ACTIVITYCODETYPE-EMPLOYEE-TR} + chr(2) + {&ACTIVITYCODETYPE-EMPLOYEE-TR}
  
```

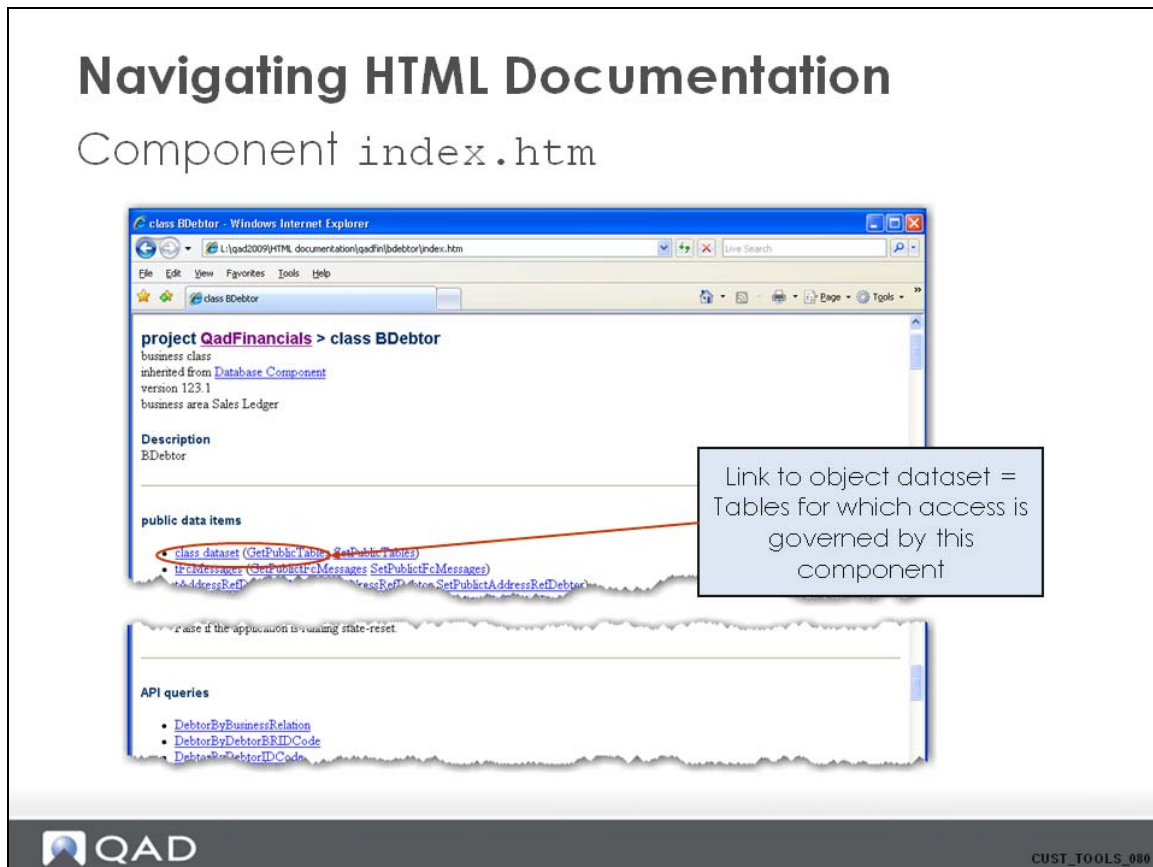
The main page for project documentation contains the following:

- ...
- A list of includes defined on project level.
- A list of preprocessors available for the project.

The components are all hyperlinked and you can use them to drill down for detailed component information. This also applies to includes.

The preprocessors are also available for the customization code. It is important that customization developers use these to be less dependent on changes made in the standard application.

## Navigating HTML Documentation



The main page for the class (component) documentation contains the following:

- A link to the project to which this component belongs.
- A link to the ancestor component (that this component inherits from).
- The internal version of the component (at the moment of generation of the HTML documentation).
- The name of the business area to which this component belongs.
- Under “public data items”:
- The object dataset (class dataset) for which data can be set and retrieved with resp. SetPublicTables and GetPublicTables.
- The list of public data item temp tables and datasets for which data can be set and retrieved with resp. SetPublic\*() methods and GetPublic\*().
- The list of public data items of simple data type (character, integer, decimal, logical, and date) for which the value can be set and retrieved with the resp. SetPublicData() and GetPublicData() methods.
- List of API queries: The queries that are available for calling by external parties.

All listed methods and queries are hyperlinked to the specific HTML files for each separate method or query.

## Navigating HTML Documentation

# Navigating HTML Documentation

## Component index.htm

**API methods**

- [ApiActivitiesAreAllowed](#)
- [apiActivityIsAllowed](#)
- [ApiCheckCreditLimit](#)
- [ApiDebtorCreditLocked](#)
- [ApiDefaultCostCentreBaseOnProfileCode](#)

**public methods**

- [ApiCheckCreditLimit](#)
- [ApiDebtorCreditLocked](#)
- [ApiDefaultCostCentreBaseOnProfileCode](#)
- [ApiDefaultDivCodeBasedOnProfileCode](#)

**activities**

- BrowseDrafts
- Create
- CreditLimit
- DebtorCreditInfo
- DebtorGOTODebtorBalanceSelect
- DebtorGOTODInvoiceMovementSelect
- DebtorGOTODInvoiceSelect
- Delete
- ExcelIntegration
- EXCELINTEGRATION\_ACTIVITY
- Modify

List of methods  
Normal style: Inherited methods, not overridden  
**Bold style:** Methods defined on this level, or overridden inherited methods

CUST\_TOOLS\_090

The main page for the class (component) documentation contains the following:

- List of API methods: The methods that are available for calling by external parties. Typically, calls to these methods are made via the 4GL proxies. There is more detail on this later in this class.
- List of public methods: The methods of the component that can be called from other components.
- List of “other” methods: The remaining methods, which do not belong to one of the above categories.
- List of activities: Activities are the secured business functions that can be executed on the business component. Most of the activities correspond with entries on the application’s menu.

All listed methods and queries are hyperlinked to the specific HTML files for each separate method or query.

If a method is inherited, but not overridden, the hyperlink brings you to the place in the code for the ancestor code.

## Navigating HTML Documentation

# Navigating HTML Documentation

M-<methodname>.htm

method GetDebtorInvoiceData - Windows Internet Explorer

project QadFinancials > class BDebtor > method GetDebtorInvoiceData

**Description**  
call query tqDInvoiceByDebtor and put all data to dDInvoiceSales.

**Parameters**

icCompanyListID	input	character	Comma-seperated list of company-IDs
idDebtorID	input	integer	
idInvoiceStartDate	input	date	
idInvoiceEndDate	input	date	
idInvoicesOpen	input	logical	
dInvoiceSales	output	temp-table	
ioReturnStatus	output	integer	Return status of the method

**internal usage**

References to this method from other application components

List of parameters. Naming convention: First letter of parameter name indicates Input or Output (i/o) Second letter of parameter name indicates Data Type (c/v/d/i/t) Example: icCompanyListID is an input character parameter

QAD CUST\_TOOLS\_100

The method page contains the following information:

- The name of the method and the project and class (component) to which the method belongs.
- The return data type. Usually void. For methods of type “function”, it is a real data type.
- The description of the method.
- The list of parameters and descriptions (if available).
- The places in the code that reference this method is referenced (where it is called).

## Navigating HTML Documentation

# Navigating HTML Documentation

M-<methodname>.htm

The screenshot displays a web browser window showing program code for a method named `BDebtor`. The code includes a query call and a method call, both highlighted with red circles and labeled with callouts.

```

program code (program9/bdebtor.p)

empty temp-table tDInvoiceSales.

if iDebtorID <> ? and iDebtorID <> 0 and
icCompanyListID <> ? and icCompanyListID <> '' :U
then do:
  <Q-4 run BInvoiceByDebtor
  [Start] in BInvoice
  /* For each on company list to improve performance */
  COMPANYBLOCK:
  do viCompanyCnt = num-entries(icCompanyListID, ',':U) to 1 by -1:
    assign viTempCompanyID = integer(entry(viCompanyCnt, icCompanyListID)) no-error.

    if error-status:error
    then do:
      assign vcMessage = trim(#T-7'Could not retrieve company ID from Company ID List.':255(65177)T-7#)
      oiReturnStatus = -3.
      <M-6 run SetMessage
      (input vcMessage (icMessage),
       input ':U (icArguments),
       input ':U (icFieldName),
       input ':U (icFieldValue),
       input 'E':U (icType),
       input 3 (iSeverity),
       input ':U (icRowid),
       input 'QadFin-6950':U (icFcMsgNumber),
       input ':U (icFcExplanation),
       input ':U (icFcIdentification),
       input ':U (icFcContext),
       output viFcReturnStatus (oiReturnStatus)) in BDebtor>
  
```

The callouts in the image are:

- Query call:** Points to the `<Q-4 run BInvoiceByDebtor` tag.
- Method call:** Points to the `<M-6 run SetMessage` tag.

The browser's address bar shows "Local intranet" and the page title is "CUST\_TOOLS\_110".

The method page contains the program code. The developer inserts this code as part of the implementation of the method in the Component Builder.

The code can contain specific CB tags or calls. These calls can be the following:

- Query calls. (`<Q-...>`) These represent execution of queries to retrieve data in records of a temp table. Values for the input parameters are also visible in the query call tag. The name of the query within the tag is hyperlinked and can be used to drill down to the specific documentation for the called query.
- Method calls. (`<M-...>`) These represent execution of methods. Values for the parameters are also visible in the method call tag. The name of the method within the tag is hyperlinked and can be used to drill down to the specific documentation for the called method.
- Include calls. (`<I-...>`). These indicate the place where an include file is included in the code.

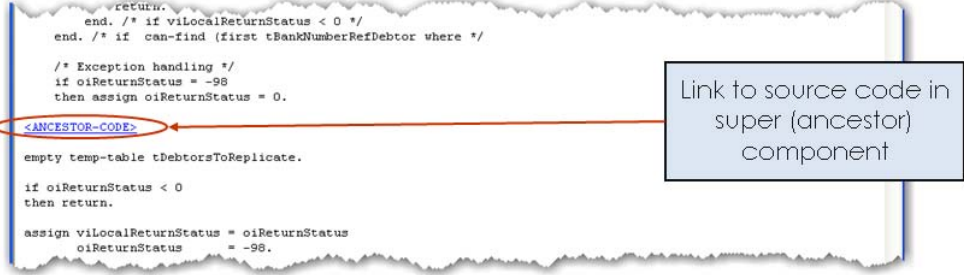
## Navigating HTML Documentation

## Navigating HTML Documentation

M-<methodname>.htm

```
return.  
end. /* if vLocalReturnStatus < 0 */  
end. /* if can-find (first tBankNumberRefDebtor where */  
  
/* Exception handling */  
if oiReturnStatus = -98  
then assign oiReturnStatus = 0.  
  
<ANCESTOR-CODE>  
  
empty temp-table tDebtorsToReplicate.  
if oiReturnStatus < 0  
then return.  
  
assign vLocalReturnStatus = oiReturnStatus  
oiReturnStatus = -98.
```

Link to source code in super (ancestor) component



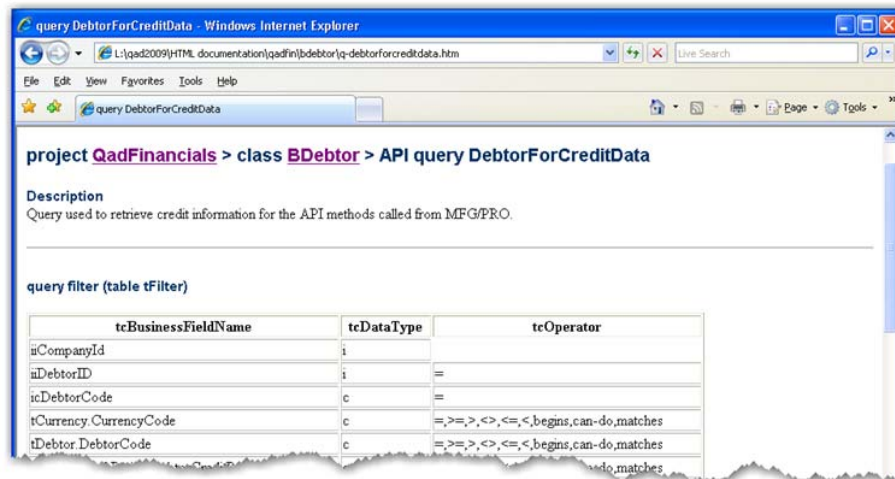
The screenshot shows a code block with a callout box pointing to the <ANCESTOR-CODE> tag. The callout box contains the text 'Link to source code in super (ancestor) component'. The code block is surrounded by a blue border and has a torn paper effect at the top and bottom edges.

For inherited methods, the code normally includes an <ANCESTOR-CODE> tag. This is the place where the code in the method of the ancestor component is executed. The tag is hyperlinked for easy navigation to the ancestor code.

## Navigating HTML Documentation

# Navigating HTML Documentation

Q-<queryname>.htm



CUST\_TOOLS\_130

The query page contains the following information:

- The name of the method and the project and class (component) to which the query belongs.
- The description of the query.
- The list of parameters and descriptions (if available).
- The query filter. This displays all of the possible records that can be passed via the tFilter dataset or temp table.

## Navigating HTML Documentation

# Navigating HTML Documentation

Q-<queryname>.htm

The screenshot shows a query page with the following content:

```

each Debtor where
Debtor.SharedSet_Id = vi_DEBTOR_sharedset(@CompanyId) AND
Debtor.Debtor_ID = nDebtorID AND
Debtor.DebtorCode = icDebtorCode

first Currency (inner-join) where
Currency.Currency_ID = Debtor.Currency_ID AND

first DebtorCreditRating (outer-join) where
DebtorCreditRating.DebtorCreditRating_ID = Debtor.DebtorCreditRating_ID AND
  
```

query resultset tqDebtorForCreditData

field name	data type	db field	description
tcCurrencyCode	character	Currency.CurrencyCode	Currency Code
nDebtor_ID	integer	Debtor.Debtor_ID	Record ID
tcDebtorCode	character	Debtor.DebtorCode	Customer Code
tcDebtorCreditRatingCode	character	DebtorCreditRating.DebtorCreditRatingCode	Code
tdDebtorFixedCredLimTC	decimal	Debtor.DebtorFixedCredLimTC	Fixed Credit Limit
tdDebtorHighCredit	decimal	Debtor.DebtorHighCredit	High Credit



CUST\_TOOLS\_140

The query page contains the following information:

- The query (condition).
- The description of the query result set.

## Navigating HTML Documentation

# Navigating HTML Documentation

Q-<queryname>.htm



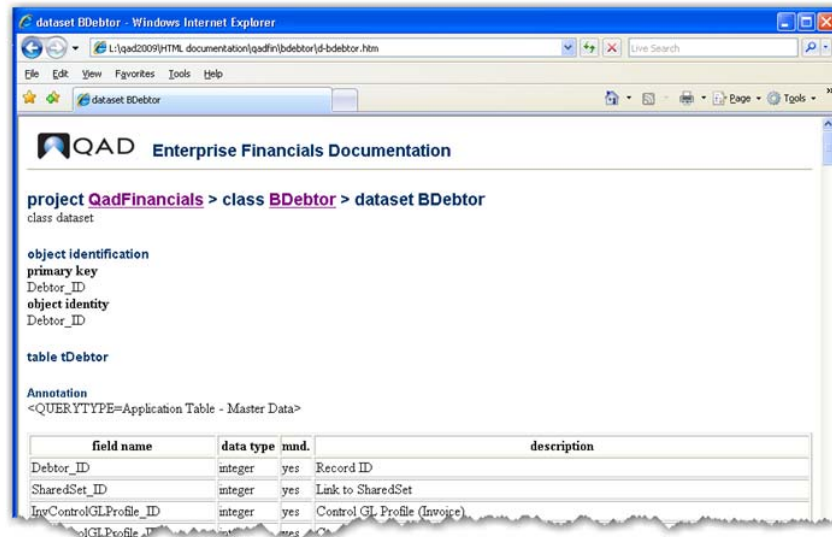
CUST\_TOOLS\_150

The query page contains the places in the code that reference (call) the query.

## Navigating HTML Documentation

# Navigating HTML Documentation

D-<datasetName>.htm



The dataset page contains the list of fields for each table in the dataset.

## Navigating HTML Documentation

# Navigating HTML Documentation

D-<datasetName>.htm

table tDebtorSafDefault

field name	data type	mnd.	description
DebtorSafDefault_ID	integer	yes	Record ID
Debtor_ID	integer	yes	Link to Debtor
SafConcept_ID	integer	yes	SAF Concept Code
Saf_ID	integer	yes	SAF Code
tcSafConceptCode	character	no	SAF Concept Code
tcSafCode	character	no	SAF Code
LastModifiedDate	date	no	Last Modified Date
LastModifiedTime	integer	no	Last Modified Time
LastModifiedUser	character	no	Last Modified User
tc_Rowid	character	yes	primary index
tc_ParentRowid	character	yes	= tDebtor.tc_Rowid
tc_Status	character	no	update status



CUST\_TOOLS\_170

For an object dataset (for example, BDebtor), the tc\_parentrowid shows the relation to the parent table in the object dataset.

## Hands-on Exercise (9)

### Hands-on Exercise (9)

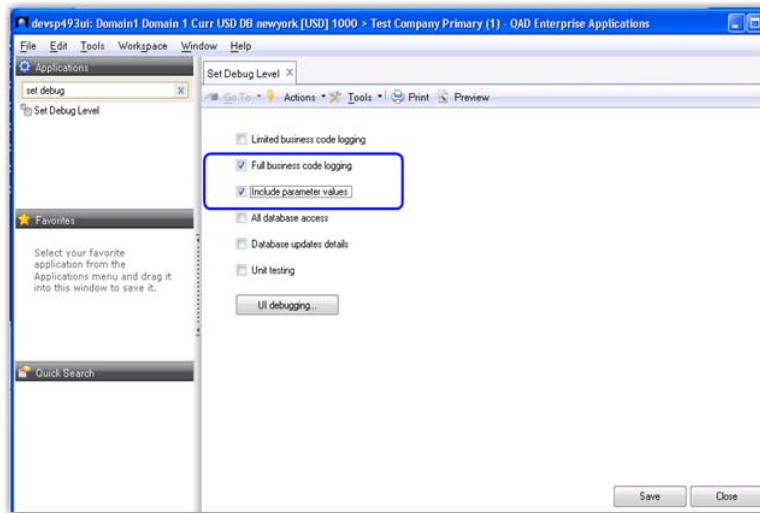


Using the HTML Documentation Set

## BL Customizations - Tools

### BL Customizations – Tools

- BL Code Tracing (CT Log file)
- Use Set Debug Level



The ability to have a full code execution trace on the backend BL is indispensable for more complex customization work. All components follow the same standard patterns for the normal maintenance functionality, and for most other functions too. But some more complex components have many sub-flows or deviations from the normal flow. With the BL Code tracing capability, one can find the exact flow.

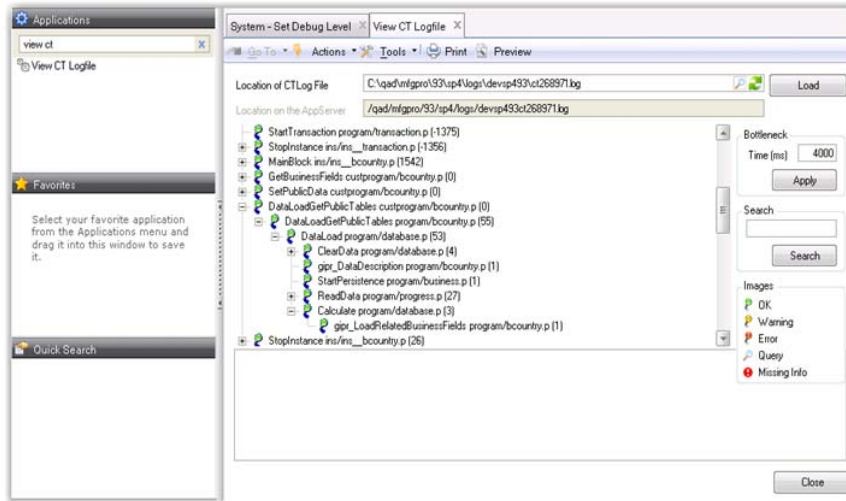
Use the “Set Debug Level” function in the menu to activate the backend logging. Make sure that all of the forms are closed before activating this feature. Typically, you select the “Full business code logging” and the “Include parameter values”. You can use the same function to deactivate the logging.

When the logging is on, a file named `ct<sessionID>.log` is created on the server. All methods and queries executed on the backend are logged.

## BL Customizations - BL Code Tracing

# BL Customizations – BL Code Tracing

Use View CT Logfile



CUST\_TOOLS\_200

To view the contents of the current CT log file (containing the business code trace), use “View CTLog” from the menu.

## Hands-on Exercise (10)

# Hands-on Exercise (10)



**Use Business Logic Logging**

## BL Customizations

### BL Customizations

Methods from same component can be called directly from Customization code. For example:

- `SetMessage` to set a certain message accompanying an error status
- `GetPublicData` to retrieve the value of a public data item:

```
RUN GetPublicData  
("vcActivityCode", OUTPUT  
vcActivityCode, OUTPUT viReturn)
```

- `SetPublicData` to assign a value to a public data item



CUST\_TOOLS\_220

## BL Customizations

## BL Customizations

Methods from other components need to be called through their proper instance

Define variable viSessionId as integer no-undo.  
 Define variable vc as character no-undo.  
 Define variable viReturn as integer no-undo.  
 Define variable vh as handle no-undo.  
 Define variable viSafInstance as integer no-undo.

- 1 run GetPublicData(input " viSessionId" , output vc, output viReturn).  
Assign viSessionId = integer(vc).
- 2 run ins/ins\_\_bsaf.p persistent set vh.
- 3 run MainBlock in vh  
(input viSessionId, input 0, input 0, input no, input "",  
input-output viSafInstance , output viReturn).
- 4 run <method> in vh (<parameters>).
- 5 run StopInstance in vh  
(input "", input "", input "", input no, output viReturn).  
delete procedure vh.



CUST\_TOOLS\_230

1. Retrieve the current session ID. This is required to start an instance of the business component.
2. Run the instance procedure of the business component persistently to make sure that the methods become available.
3. Run the MainBlock in the instance procedure. This ensures that the state of the business component instance is OK. Remark the viSafInstance. If this value is 0, it means that the instance is initialized. If it has a certain value, the mechanism tries to restore the instance based on previously saved state data.
4. Run the method.
5. Stop the instance. Watch the fourth parameter, which indicates whether to keep the instance for later reuse. Right after the StopInstance call, the handle to the persistent procedure must be deleted (to prevent memory leaks)!

## BL Customizations

### BL Customizations

- Using another business component:  
Example – Create Projects
- Project code must be 'P<99999>' with a sequence number that has no gaps
- We will use methods `GetNumber` + `CommitNumber` + `ReleaseNumber` of class `BNumber` for this



CUST\_TOOLS\_240

For this case, we assume that projects cannot be deleted and that projects are not created through QXtend or other integration. Read the HTML documentation on the business methods used.

## BL Customizations

## BL Customizations

### Using another Business Component

```
variables.i
```

```
Define variable viSessionId as integer no-undo.  
Define variable vc as character no-undo.  
Define variable viMethodReturn as integer no-undo.  
Define variable viReturn as integer no-undo.  
Define variable viCompany as integer no-undo.  
Define variable viNum as integer no-undo.  
Define variable vh as handle no-undo.  
Define variable viBNumberInstance as integer no-undo.
```

**BL Customizations**

## BL Customizations

### Using another Business Component

```
startnumber.i
```

```
run GetPublicData (input "viSessionId", output vc, output
viReturn).
Assign viSessionId = integer(vc).
run ins/ins__bnumber.p persistent set vh.
run MainBlock in vh
  (input viSessionId, input 0, input 0, input no, input "",
  input-output viBNumberInstance , output viReturn).
```



CUST\_TOOLS\_260

**BL Customizations**

## BL Customizations

Using another Business Component

```
stopbnumber.i
```

```
run StopInstance in vh  
  (input "", input "", input "", input no, output viReturn).  
delete procedure vh.
```

## BL Customizations

## BL Customizations

### Using another Business Component

```
procedure bproject.initialvalues.before:
{ variables.i }

if t_parameter.icTableName = "project" then do:
  run GetPublicData
  (input "viCompanyId", output vc, output viReturn).
  Assign viCompany = integer(vc).
  { startbnumber.i }
  run GetNumber in vh
  (input viCompany, input 9999, input "PROJECTCODE",
  output viNum, input 0, input "", output viMethodReturn).
  { stopbnumber.i }
  if viMethodReturn < 0
  then Assign t_parameter.oiReturnStatus = viMethodReturn.
  else Assign tproject.projectcode = "P" + string(viNum,"99999").
end.
end procedure.
```



CUST\_TOOLS\_280

GetNumber reserves the project code number; if another user starts project create before we save our project, the user does not get the same number.

## BL Customizations

## BL Customizations

### Using another Business Component

```

procedure bproject.postsave.before:
{ variables.i }

for each tproject where tproject.tc_Status = "N":
  viNum = integer(substring(tproject.ProjectCode,2)).
  run GetPublicData
    (input "viCompanyId", output vc, output viReturn).
  Assign viCompany = integer(vc).
  { startbnumber.i }
  run CommitNumber in vh
    (input viCompany, input 9999, input "PROJECTCODE",
    input viNum, input 0, input "", output viMethodReturn).
  { stopbnumber.i }
  if viMethodReturn < 0
  then Assign t_parameter.oiReturnStatus = viMethodReturn.
end.
end procedure.

```



CUST\_TOOLS\_290

CommitNumber confirms that the number was used and is never used again.

## BL Customizations

## BL Customizations

### Using another Business Component

```

procedure bproject.exitinstance.before:
{ variables.i }

for each tproject where tproject.tc_Status = "N":
  viNum = integer(substring(tproject.ProjectCode,2)).
  run GetPublicData
    (input "viCompanyId", output vc, output viReturn).
  Assign viCompany = integer(vc).
  { startbnumber.i }
  run ReleaseNumber in vh
    (input viCompany, input 9999, input "PROJECTCODE",
      input viNum, input 0, input "", output viMethodReturn).
  { stopbnumber.i }
  if viMethodReturn < 0
  then Assign t_parameter.oiReturnStatus = viMethodReturn.
end.
end procedure.

```



CUST\_TOOLS\_300

ReleaseNumber cancels the reservation on the number used in the project being created. If you create another project, it reuses the number.

## BL Customizations

### BL Customizations

Methods from technical components need to be called through the component program

```
Define variable vh as handle no-undo.
```

- 1 `run program/<classname>.p persistent set vh (<main method parameters>).`
- 2 `run <method> in vh (<parameters>).`
- 3 `run gipr_DeleteProcedure in vh.  
delete procedure vh.`



CUST\_TOOLS\_320

Technical components provide technical functions. Unlike business components, technical components do not have any business responsibility and therefore do not have business functions that external consumers can call. External consumers are external programs that connect a certain way to the application backend and call functions.

In contrast to business components, technical components do not have instances that must be started before functions can be called on them. The methods in technical components can only be called from within the business layer running on the AppServer backend. In that sense, technical components are private to the business layer.

Some exceptions could be exposed to run from an external consumer, but this must be specifically written and documented.

1. Run the technical component. Unlike an instance program, a technical component does not have a fixed parameter set. Look in the HTML documentation to see which parameters to pass.
2. Run the method.
3. Stop the technical component, but first run a (generated) internal procedure to clean up any resources used by the component.

## BL Customizations

### BL Customizations

- Using a technical component:  
Example – reading an XML document
- In the BLF project, you will find an XML technical component
- We will use method `ReadXmlNodeValue` to read a single node from an XML file

## BL Customizations

## BL Customizations

### Using a technical component

```

procedure bproject.initialvalues.before:
{ variables.i }

if t_parameter.icTableName = "project" then do:
  run program/xml.p persistent set vh. /* no parameters */
  run ReadXmlNodeValue in vh
    (input search("server.xml"),
     input "logging",
     input "LoggingDirectory",
     output tproject.ProjectDescription,
     output viMethodReturn).
  run gipr_DeleteProcedure in vh.
  delete procedure vh.
  if viMethodReturn < 0
  then Assign t_parameter.oiReturnStatus = viMethodReturn.
end.
end procedure.

```



CUST\_TOOLS\_330

server.xml is a random XML file that is available. LoggingDirectory is a random node in this file. What would happen if you forget “gipr\_DeleteProcedure”? Find out using unit test logging.

## Typical Customization: Case 1

### Typical Customization: Case 1

**On an existing object form, add a field that holds extra non-standard information of the object. Provide validation for this field.**

Steps for implementation of this Customization:

- [UI] Define a user-defined field for the business component
- [UI] Use the UI design mode option on the form to add the user-defined field
- [BL] Implement the hook `<BusinessComponent>.ValidateComponent.After` to include validation on the user-defined field.
- [BL] Compile, test, and deploy the new code

DEMO



CUST\_TOOLS\_340

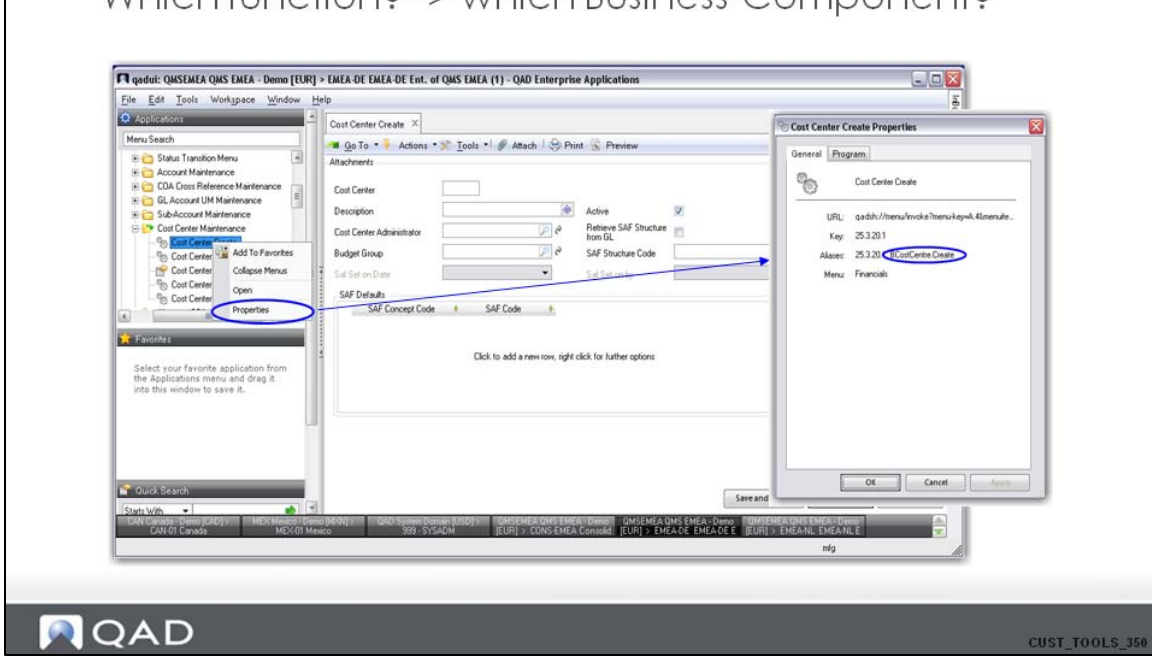
- Most of the forms in the client UI represent application objects. These forms are called “object forms”.
- Object forms are typically built up of business fields representing data for the object.
- Every object in the Financials has a set of predefined fields (called “Custom\*”) that can be used as user-defined fields, for which the user can determine the meaning.
- In the slide, [UI] means that this step is done by someone straight from the client user interface in the application. No code changes are involved in these steps.
- In the slide, [BL] means that this step is done by a 4GL developer adding code in the hooks for non-intrusive customization of the business logic running on the backend.

Demo Customization: Case 1

# Demo Customization: Case 1

## Preparation

Which function? -> which Business Component?



CUST\_TOOLS\_350

## Demo Customization: Case 1

## Demo Customization: Case 1

[UI] Define a user-defined field for the business component

Business Component: Cost Center  
 Field Name: tCostCentre.CustomCombo0  
 Description: Special Type

General | Value List

Side Label: Special Type  
 Column Label: SType  
 Display Format: x(20)  
 Display Length: 20  
 Decimal Precision: 0  
 Mandatory:   
 Lookup: None  
 Lookup Reference:   
 Stored Search:   
 Return Field:

Custom Value	Seq
CC001	1
CC002	2
CC003	3
CC004	4
CA001	5
CA002	6
BI001	7
BI002	8
BI003	9

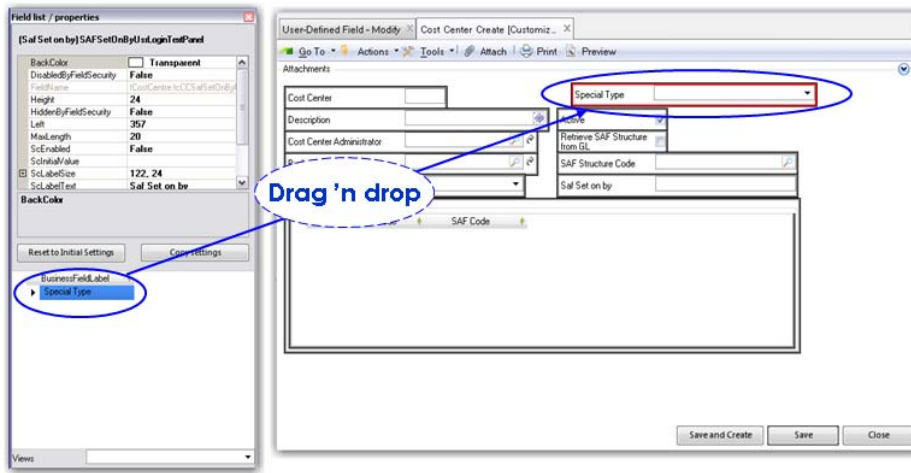
QAD CUST\_TOOLS\_370

We use a “CustomCombo” type field. For this type of field, you can specify a list of possible values.

Demo Customization: Case 1

# Demo Customization: Case 1

[UI] Use the UI design mode option on the form to add the user-defined field

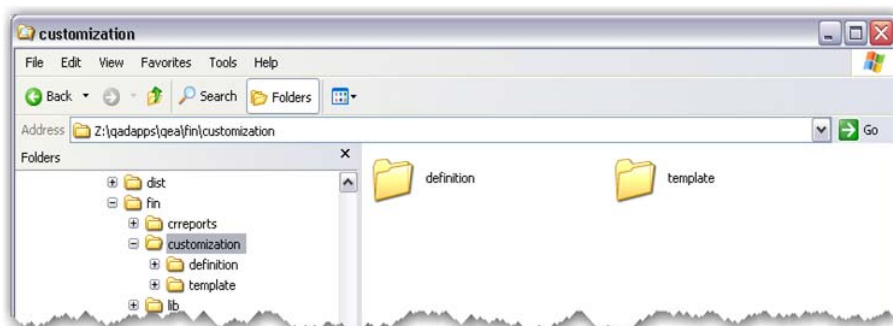


## Demo Customization: Case 1

## Demo Customization: Case 1

### Preparation

- Fin AppServer PROPATH should include a folder with `customcode` as sub-folder
- Find the `Customization` folder in the installation folder of the Financials business layer

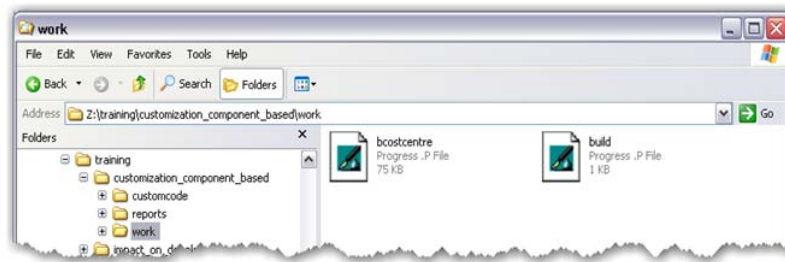


## Demo Customization: Case 1

## Demo Customization: Case 1

[BL] Implement the hook  
`<BusinessComponent>.ValidateComponent.`  
After to include validation on the user-defined  
field

Copy the file `bcostcentre.p` from the  
`Customization/template` folder to the folder that  
will contain the Customization source code



## Demo Customization: Case 1

## Demo Customization: Case 1


Edit `bcostcentre.p` and add code in `ValidateComponent`

```

**/
**/
PROCEDURE BCostCentre.ValidateComponent.After:
  DEFINE VARIABLE viReturn AS INTEGER NO-UNDO.
  FOR EACH t_sCostCentre WHERE can-do("N,C",t_sCostCentre.tc_status):
    IF t_sCostCentre.CustomCombo0 = "CC004" AND
       t_sCostCentre.tcUsrLogin <> "mfg"
    THEN DO :
      ASSIGN t_parameter.oiReturnStatus = -1.
      run SetMessage
        (input "The specific type (%1) can only be set for cost centers with admin
          input t_sCostCentre.CustomCombo0 + CHR(2) + t_sCostCentre.tcUsrLogin,
          input "Special Type",
          input t_sCostCentre.CustomCombo0,
          input "E",
          input 3,
          input t_sCostCentre.tc_Rovid,
          input "CUST-001",
          input "",
          input "",
          input "",
          output viReturn).
    END.
  END.
END PROCEDURE.

```

1  
2  
3

 CUST\_TOOLS\_400

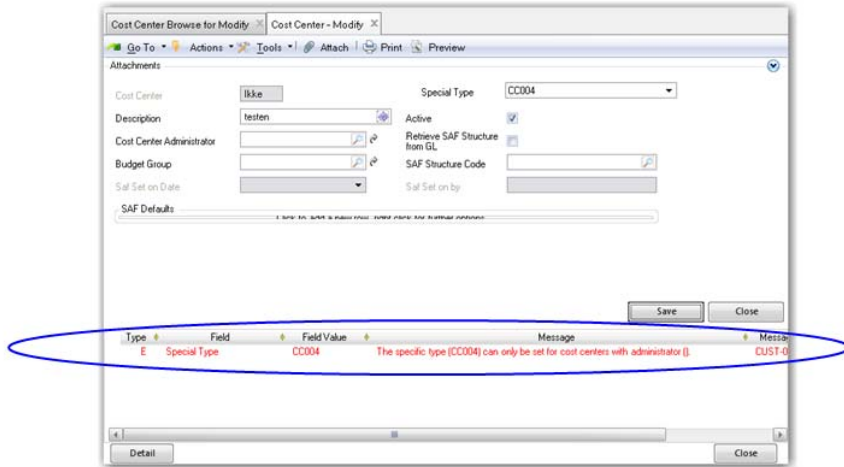
Uncomment the code for `BCostCentre.ValidateComponent.After`.

1. Loop through the data in the object dataset; only take the new and changed records to validate. Use the `t_s` buffer.
2. Assign the `oiReturnStatus` parameter to -1 (general validation error).
3. Use a call to “`SetMessage`” to pass the appropriate validation message.

Demo Customization: Case 1

# Demo Customization: Case 1

Compile, deploy customization, trim the AppServer, and run the Cost Center Create or Modify again



## Demo Customization: Case 1A

### Demo Customization: Case 1A

Provide valid values for a user-defined field

- A fixed-value list in user-defined field maintenance
- An existing browse in user-defined field maintenance (by selecting a stored search of the browse)
- Assigning a browse created in Browse Maintenance in `GetBusinessFields`:  

```
tBusinessFields.tcLookupQuery =  
"QAD.Browse.MfgProLookupProvider:<browse>:  
<return>".
```
- Using Generalized Codes Maintenance:  
36.2.13 Generalized Codes Maintenance



CUST\_TOOLS\_420

## Hands-on Exercise (11)

# Hands-on Exercise (11)



## Generalized Codes for Supplier Category

## Typical Customization: Case 2

### Typical Customization: Case 2

**On an existing object form, add an extra grid that holds records with extra non-standard information of the object.  
Provide validation for the records.**

Steps for implementing this Customization:

- [BL] Define custom tables for the business component and relationships
- [UI] Define a user-defined fields for custom tables
- [UI] Use the UI Design Mode option on the form to add the grid with the custom table
- [BL] Implement the hook  
`<BusinessComponent>.ValidateComponent.After` to include validation on the user-defined field
- [BL] Compile, test, and deploy the new code  
DEMO



CUST\_TOOLS\_440

## Demo Customization: Case 2

## Demo Customization: Case 2

[BL] Define custom tables for the business component, and relationships

Copy the right template (`bcountry.p`) and update code

```
PROCEDURE BCountry.DefineCustomRelations.after:
  create tCustomRelation.
  assign tCustomRelation.tcParentTable = "tCountry"
         tCustomRelation.tcChildTable = "tCustomTable0"
         tCustomRelation.tcChildTableDescription = "Province"
         tCustomRelation.tiIsOneToOne = false.
END PROCEDURE.
```

1

```
PROCEDURE BCountry.DataLoad.after:
  FOR EACH tCountry:
    IF NOT CAN-FIND (tCustomTable0
      WHERE tCustomTable0.tc_parentrowid = tCountry.tc_rowid )
    THEN DO:
      FOR EACH CODE_mstr WHERE CODE_filename = "CNTRY" + tcountry.CountryCode NO-LOCK:
        CREATE tCustomTable0.
        ASSIGN tCustomTable0.tcCustomShort0 = CODE_mstr.CODE_value
              tCustomTable0.tc_parentRowid = tCountry.tc_rowid
              tCustomTable0.tc_rowid = STRING(ROWID(CODE_mstr)).
      END.
    END.
  END.
END PROCEDURE.
```

2



CUST\_TOOLS\_450

The customization needs to contain code to complete the object dataset with the custom data, which in this example is coming from the generalized codes in `code_mstr` (but this can be whatever table, whatever data source). Both data retrieval and saving of the data must be covered.

1. Implement the method "DefineCustomRelations.After" to give a meaning to one of the custom tables in the object dataset (`tCustomTable0`, `tCustomTable1` or `tCustomTable2`).
2. Implement the method "DataLoad.After" to retrieve the right data from `code_mstr` to load an existing country object.

## Demo Customization: Case 2


## Demo Customization: Case 2


### Update Code

```

DEFINE TEMP-TABLE tcode_mstr LIKE CODE_mstr.
PROCEDURE BCountry.PostSave.After:
  FOR EACH tCountry:
    EMPTY TEMP-TABLE tCode_mstr.
    FOR EACH CODE_mstr WHERE CODE_mstr.CODE_fldname = "CNTRY" + tcountry.CountryCode NO-LOCK:
      CREATE tcode_mstr.
      BUFFER-COPY CODE_mstr TO tCode_mstr.
    END.
    FOR EACH tCustomTable0 WHERE tCustomTable0.tc_parentrowid = tCountry.tc_rouid AND
      tCustomTable0.tc_status <> "D":
      FIND FIRST tCode_mstr WHERE tcode_mstr.CODE_value = tCustomTable0.tcCustomShort0 NO-ERROR.
      IF AVAILABLE tCode_mstr THEN DELETE tCode_mstr.
      ELSE DO:
        CREATE CODE_mstr.
        ASSIGN CODE_mstr.CODE_fldname = "CNTRY" + tcountry.CountryCode
          CODE_mstr.CODE_value = tCustomTable0.tcCustomShort0.
      END.
    END.
    FOR EACH tcode_mstr:
      FIND CODE_mstr WHERE CODE_mstr.CODE_fldname = tCODE_mstr.CODE_fldname AND
        CODE_mstr.CODE_value = tCODE_mstr.CODE_value NO-ERROR.
      IF AVAILABLE code_mstr THEN DELETE CODE_mstr.
    END.
  END.
END PROCEDURE.

```




CUST\_TOOLS\_460

3. Implement the method “PostSave.After” to update the code\_mstr table in the database with the changes.

## Demo Customization: Case 2

## Demo Customization: Case 2

- Compile, deploy Customization, and trim the AppServer
- [UI] Define a user-defined fields for custom tables

The screenshot shows a 'User-Defined Field - Modify' dialog box. The 'Attachments' section is expanded to show the following fields:

- Business Component: Country
- Field Name: CustomTable0 to CustomChart0
- Description: Province

The 'General' tab is selected, showing the following configuration:

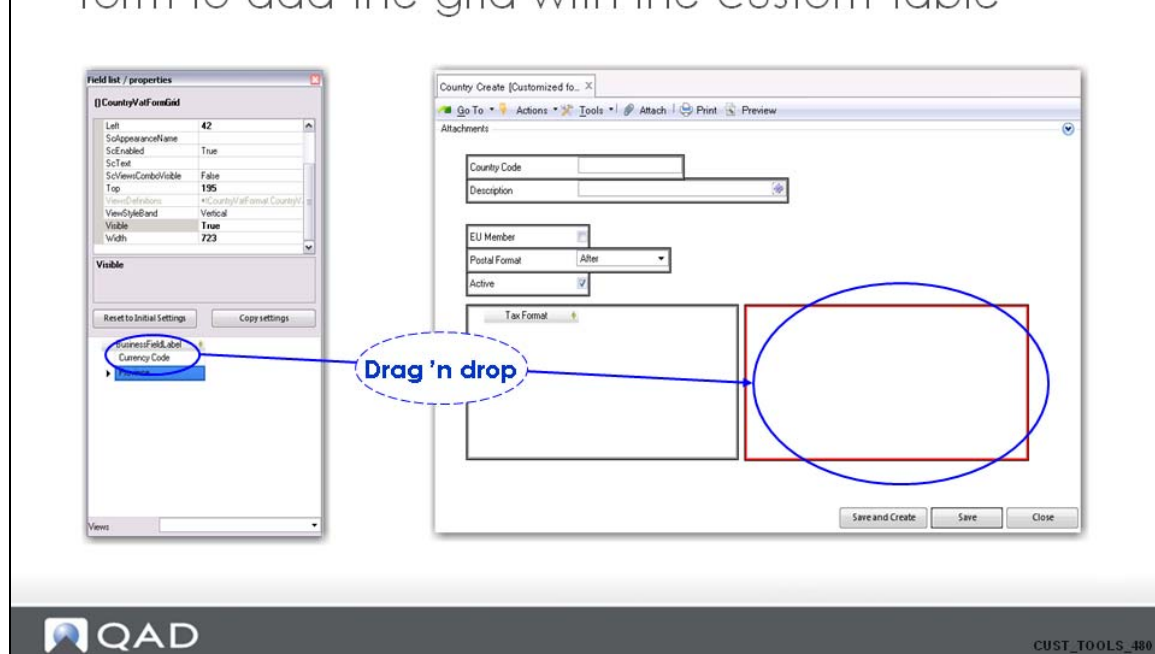
- Side Label: Province
- Column Label: Prov
- Display Format: x(20)
- Display Length: 20
- Decimal Precision: 0
- Mandatory:
- Lookup Reference:
- Stored Search:
- Return Field From Stored Search:

Buttons for 'Save' and 'Close' are located at the bottom right of the dialog.

## Demo Customization: Case 2

## Demo Customization: Case 2

[UI] Use the UI Design Mode option on the form to add the grid with the custom table



Set the property “AllowAddNew” to true in the properties panel.

Also, use the “columns” from the context menu of the new grid to make the columns visible.

Demo Customization: Case 2

# Demo Customization: Case 2

Run Country Create or Modify again

The screenshot shows a web browser window with two tabs: 'Country Browse for Modify' and 'Country Create'. The 'Country Create' tab is active. The browser's address bar shows 'Go To', 'Actions', 'Tools', 'Attach', 'Print', and 'Preview'. Below the address bar is an 'Attachments' section. The main form contains the following fields and controls:

- Country Code: Text input field containing 'BEL'
- Description: Text input field with a search icon on the right
- EU Member: Checkmark icon
- Postal Format: Dropdown menu with 'Alter' selected
- Active: Checkmark icon
- Tax Format: Section with a plus icon and the text 'Click to add a new row, right click for further options'
- Prov: Dropdown menu with 'Antwerp' and 'Limburg' visible

At the bottom of the form are three buttons: 'Save and Create', 'Save', and 'Close'.

## Custom Tables

### Custom Tables

- Three extra temp-tables in the object dataset for each business component in the application
- `tCustomTable0`, `tCustomTable1` and `tCustomTable2` with fixed number of fields (38)
- The standard application code does not use custom tables
- Customization developer is responsible for providing code to:
  - Load the data (typically in the `DataLoad.After` hook)
  - Initialize the data
  - Save the data (typically in the `PostSave.After` hook)



CUST\_TOOLS\_500

Every business component that inherits from “Database component” has these three extra temp-tables in the object dataset.

This is how you can add extra data to any object in the application in a non-intrusive way. The custom tables are part of the named (or typed) dataset that represents the object dataset. Therefore, the interface for the object does not change from the moment a customization developer starts using these temp-tables to transfer more than the standard object data.

## Custom Tables 2

### Custom Tables

- Three extra temp-tables:
- Fixed list of fields:
  - tcCustomShort [0-9]
  - tcCustomCombo [0-9]
  - tcCustomLong [0-1]
  - ttCustomDate [0-4]
  - tiCustomInteger [0-4]
  - tdCustomDecimal [0-4]
  - tcCustomNote



## Custom Tables 3

### Custom Tables

- Relations between custom tables and standard tables must be defined in the method hook `DefineCustomRelations.Before` or `DefineCustomRelations.After`
- You can only use fields in the custom tables on the UI if they are given a meaning  
This is done by the User-defined field component (Create or Modify functions)
- You can put fields from the custom tables on the UI by using the UI Design Mode
- You can put tables on the UI (in a grid) using the UI Design Mode



CUST\_TOOLS\_520

In the method `DefinCustomRelations.After`, the customization developer can specify the relations between the custom tables and the other tables. This is done by adding records in the table `tCustomRelation`. The relation is always based on the `tc_rowid` and `tc_parentrowid`.

## Demo Customization: Case 2 - DefineCustomRelations

# Demo Customization: Case 2

## DefineCustomRelations

```

PROCEDURE BCountry.DefineCustomRelations.after :
  create tCustomRelation.
  assign tCustomRelation.tcParentTable = "tCountry"
         tCustomRelation.tcChildTable = "tCustomTable0"
         tCustomRelation.tcChildTableDescription = "ProvinceTable"
         tCustomRelation.tlIsOneToOne = false.

END PROCEDURE .

```



CUST\_TOOLS\_530

The only way to make sure that the system recognizes the custom table is to add at least one record in tCustomRelation temp-table.

The record fields are:

- tcParentTable: Name of the temp-table from the object dataset. Typically “t<table>”.
- tcChildTable: Name of the custom table from the object dataset that you want to give a meaning. This can be “tCustomTable0”, “tCustomTable1” or “tCustomTable2”.
- tcChildTableDescription: The logical name to use in the system when referring to the table. For example, the UI Design mode uses this name to select the table.
- tlIsOneToOne: The cardinality of the relation. If true, it means that the table is linked 1-1 with the parent table. In this case, the fields from the table can be used to put on a form representing the parent table. If it is false, it automatically means 1-N. In this case, the user can select individual fields as well as the whole table to drag in the UI design mode.

## Demo Customization: Case 2 - DataLoad

## Demo Customization: Case 2

### DataLoad

```

PROCEDURE BCountry.DataLoad.after:
    DEFINE VARIABLE vcProvince AS CHARACTER NO-UNDO.
    FOR EACH tCountry:
        FILE-INFO:FILE-NAME = "c:\temp\Country" + tCountry.CountryCode +
        ".txt"
        NO-ERROR.
        IF FILE-INFO:FILE-CREATE-DATE <> ?
        THEN DO:
            INPUT FROM VALUE ("c:\temp\Country" + tCountry.CountryCode +
            ".txt").
                REPEAT:
                    IMPORT vcProvince.
                    CREATE tCustomTable0.
                    ASSIGN tCustomTable0.tcCustomShort0 = vcProvince
                        tCustomTable0.tc_parentRowid = tCountry.tc_rowid
                        tCustomTable0.tc_rowid = "rowid" + string(TIME).
                END.
            INPUT CLOSE.
        END.
    END.
END PROCEDURE.

```



CUST\_TOOLS\_540

This example uses a plain .txt file to store data. The data is normally stored in a database table.

## Demo Customization: Case 2 - PostSave

# Demo Customization: Case 2

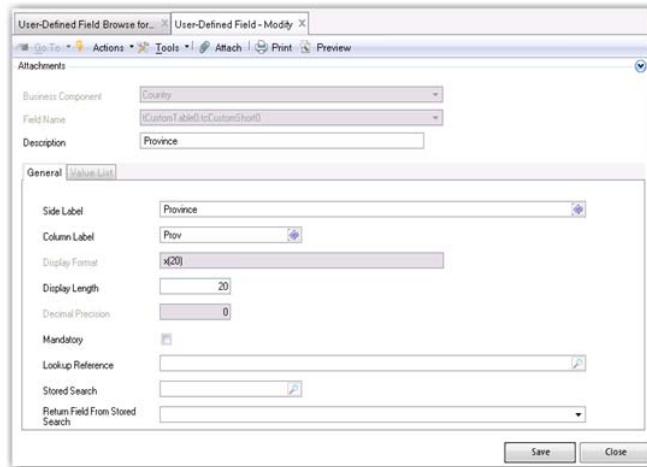
## PostSave

```
PROCEDURE BCountry.PostSave.after:  
  FOR EACH tCustomTable0 WHERE can-do("C,N",tCustomTable0.tc_status),  
    tCountry WHERE tCountry.tc_rowid = tCustomTable0.tc_parentrowid  
    BREAK BY tCountry.CountryCode:  
    IF FIRST-OF (tCountry.CountryCode)  
    THEN OUTPUT TO VALUE ("c:\temp\Country" + tCountry.CountryCode +  
".txt").  
    EXPORT tCustomTable0.tcCustomShort0.  
    IF LAST-OF (tCountry.CountryCode)  
    THEN OUTPUT CLOSE.  
  END.  
END PROCEDURE.
```

## Demo Customization: Case 2 - Compile and Deploy

### Demo Customization: Case 2

- Compile, deploy Customization, and trim the AppServer
- [UI] Define a user-defined field for custom tables



## Demo Customization: Case 2 - Add Grid

## Demo Customization: Case 2

[UI] Use the UI Design Mode option on the form to add the grid with the custom table

The screenshot illustrates the process of adding a grid to a form in QAD. On the left, the 'Field list / properties' panel shows a tree view of fields. A red oval highlights the 'Province' field, with a callout stating: "For custom tables defined as 1-1, fields are listed in sublevel". A blue dashed oval labeled "Drag & drop" indicates the action of moving the 'Province' field to a red-bordered grid area on the 'Country Create' form. A smaller inset shows the 'Province' table structure, which includes columns for 'Province' and 'Province Table'.

Set the property "AllowAddNew" to true in the properties panel.

Also, use "columns" from the context menu of the new grid to make the columns visible.

## Hands-on Exercise (12)

# Hands-on Exercise (12)



## Custom Tables in a SAF Object

## Typical Customization: Case 3

### Typical Customization: Case 3

**Create an object form, add all fields that hold the information of the object.  
Provide validation for these fields.**

Steps for implementation of this Customization:

- [BL] Design database tables and object model
- [BL] Compile and deploy the custom code
- [UI] Define a user-defined component
- [UI] Define user-defined fields
- [UI] Define menu items for the activities
- [UI] Define role permissions for the activities
- [UI] Use the UI design mode to build the object form

DEMO



CUST\_TOOLS\_590

**Demo Customization: Case 3 - Database Model**

## Demo Customization: Case 3


### Database Model

Table: LeaseCar

<u>Field Name</u>	<u>Field Label</u>	<u>Data Type</u>	<u>Flg</u>	<u>Format</u>
LeaseCar_ID	identifier	integer	im	->, >>>, >>9
LeaseCarLicensePlate	License plate	character	im	x(20)
LeaseCarType	Type of Car	character		x(40)
LeaseCarStartDate	Lease Start Date	date		99/99/99
LeaseCarEndDate	Lease End Date	date		99/99/99
LeaseCarCompany	Lease Company Code	character		x(20)

<u>Flags</u>	<u>Index Name</u>	<u>Field Name</u>
pu	ID	+ LeaseCar_ID
u	Plate	+ LeaseCarLicensePlate


CUST\_TOOLS\_600

This customization case demonstrates the maintenance of a fleet of leased cars and vans and to which employee a vehicle is assigned.

Demo Customization: Case 3 - Database Model

# Demo Customization: Case 3

## Database Model

Table: LeaseCarUsage

<u>Field Name</u>	<u>Field Label</u>	<u>Data Type</u>	<u>Flg</u>	<u>Format</u>
LeaseCarUsage_ID	identifier	integer	im	->, >>>, >>9
LeaseCar_ID	parent	integer	im	->, >>>, >>9
LeaseCarUsageEmployeeCode	Employee Code	character		x(20)
LeaseCarUsageStartDate	Usage Start Date	date		99/99/99
LeaseCarUsageEndDate	Usage End Date	date		99/99/99

<u>Flags</u>	<u>Index Name</u>	<u>Field Name</u>
pu	ID	+ LeaseCarUsage_ID
	Parent	+ LeaseCar_ID



## Demo Customization: Case 3 - Field Mapping

# Demo Customization: Case 3

## Field Mapping

<u>Table</u>	<u>Field Name</u>	<u>Table</u>	<u>Field Name</u>
LeaseCar		tCustomTable0	
	LeaseCar_ID		CustomInteger0
	LeaseCarLicensePlate		CustomShort0
	LeaseCarType		CustomLong0
	LeaseCarStartDate		CustomDate0
	LeaseCarEndDate		CustomDate1
	LeaseCarCompany		CustomShort1
LeaseCarUsage		tCustomTable1	
	LeaseCar_ID		CustomInteger0
	LeaseCarUsage_ID		CustomInteger1
	LeaseCarUsageEmployeeCode		CustomShort0
	LeaseCarUsageStartDate		CustomDate0
	LeaseCarUsageEndDate		CustomDate1

## Custom Components

### Custom Components

- One ancestor class `bCustom`
- `tCustomTable0`, `tCustomTable1` and `tCustomTable2` with fixed number of fields (38)
- Customization developer is responsible for providing code to:
  - Load the data (typically in the `DataLoad.After` hook)
  - Initialize the data (typically the `InitialValues.After` hook)
  - Validate the data (typically the `ValidateComponent.After` hook)
  - Save the data (typically in the `DataSave.After` hook)



CUST\_TOOLS\_630

Every custom business component is inherited from component “bcustom”. Every custom component has access to three custom tables only.

## Custom Components

### Custom Components

Create a custom component:

- Always start from the template `template/bcustom.p`
- File name for the custom component is `bcustom[xyz].p`

All in lower case. UNIX file names are case-sensitive



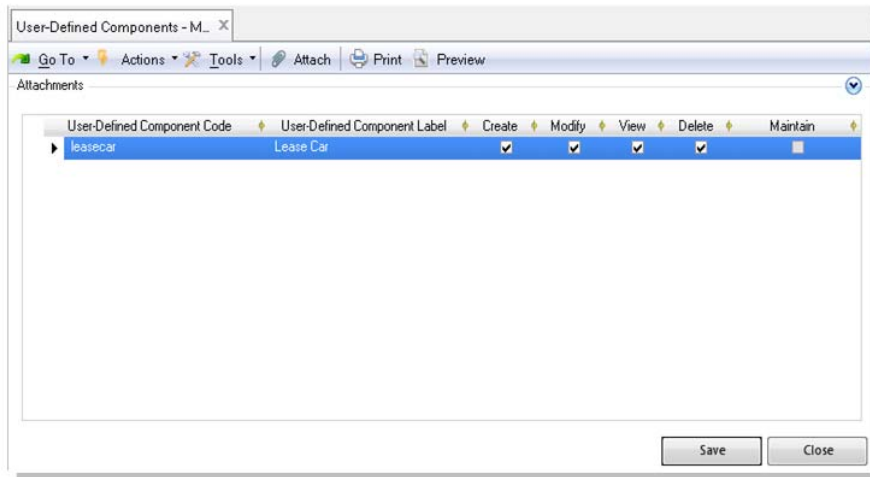
CUST\_TOOLS\_6-40

Copy the `bcustom.p` template into your source code folder and rename it as required.

## Demo Customization: Case 3 - Define User-Defined Component

### Demo Customization: Case 3

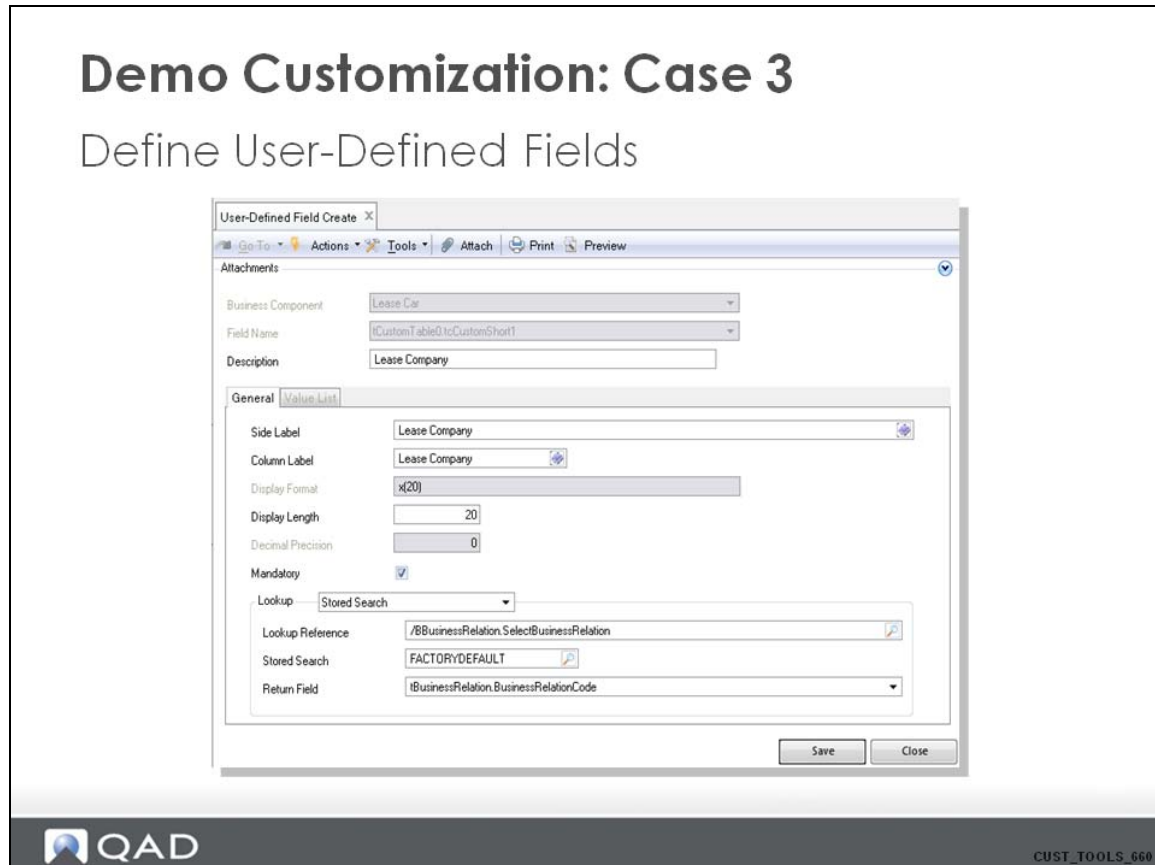
#### Define a User-Defined Component



CUST\_TOOLS\_650

Fill in code + label and select the activities Create / Modify / Delete / View.

## Demo Customization: Case 3 - Define User-Defined Fields



Define a user-defined field definition for these fields:

tCustomTable0.CustomShort0 (LeaseCarLicensePlate)

tCustomTable0.CustomLong0 (LeaseCarType)

tCustomTable0.CustomDate0 (LeaseCarStartDate)

tCustomTable0.CustomDate1 (LeaseCarEndDate)

tCustomTable0.CustomShort1 (LeaseCarCompany)

tCustomTable1.CustomShort0 (LeaseCarUsageEmployeeCode)

tCustomTable1.CustomDate0 (LeaseCarUsageStartDate)

tCustomTable1.CustomDate1 (LeaseCarUsageEndDate)

- All fields are mandatory.
- Select lookup BBusinessRelation.SelectBusinessRelation for field “Lease Company”.
- Select lookup BEmployee.SelectEmployee for field “Employee Code”.

## Demo Customization: Case 3 - Define Menu Items

# Demo Customization: Case 3

## Define menu items

Menu System Maintenance X

Go To Actions Copy Print Preview

Language ID: US

Language ID: us                      english (U.S.)

Menu: 30.4                              Lease Cars


Selection: 1

---

Label:

Name:

Exec Procedure:


CUST\_TOOLS\_670

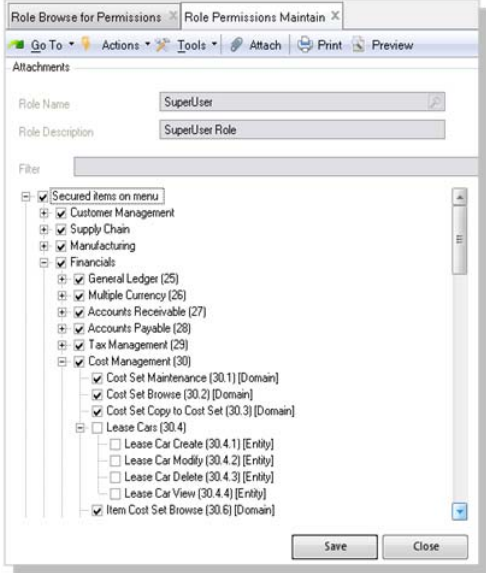
Define these menu items:

<u>Nr</u>	<u>Label</u>	<u>Procedure</u>
30.4	Lease Cars	30.4
30.4.1	Lease Car Create	urn:cbf:bcustom[leasecar].Create
30.4.2	Lease Car Modify	urn:cbf:bcustom[leasecar].Modify
30.4.3	Lease Car Delete	urn:cbf:bcustom[leasecar].Delete
30.4.4	Lease Car View	urn:cbf:bcustom[leasecar].View

## Demo Customization: Case 3 - Define Role Permissions

# Demo Customization: Case 3


## Define role permissions



The screenshot shows a software window titled "Role Browse for Permissions" with a sub-tab "Role Permissions Maintain". The window has a menu bar with "Go To", "Actions", "Tools", "Attach", "Print", and "Preview". Below the menu bar, there are fields for "Role Name" (SuperUser) and "Role Description" (SuperUser Role). A "Filter" field is empty. A tree view of permissions is shown, with the following items checked:

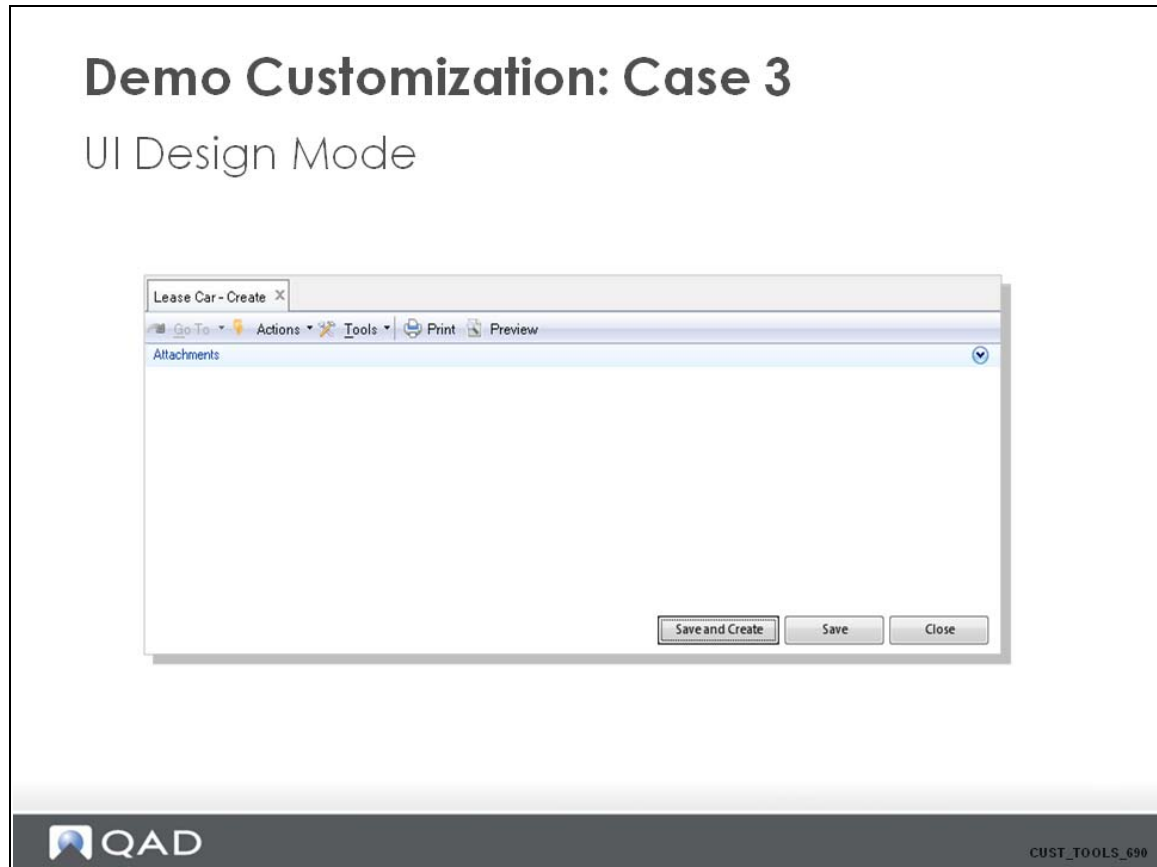
- Secured Items on menu
- Customer Management
- Supply Chain
- Manufacturing
- Financials
  - General Ledger (25)
  - Multiple Currency (26)
  - Accounts Receivable (27)
  - Accounts Payable (28)
  - Tax Management (29)
  - Cost Management (30)
    - Cost Set Maintenance (30.1) [Domain]
    - Cost Set Browse (30.2) [Domain]
    - Cost Set Copy to Cost Set (30.3) [Domain]
    - Lease Cars (30.4)
      - Lease Car Create (30.4.1) [Entity]
      - Lease Car Modify (30.4.2) [Entity]
      - Lease Car Delete (30.4.3) [Entity]
      - Lease Car View (30.4.4) [Entity]
    - Item Cost Set Browse (30.6) [Domain]

Buttons for "Save" and "Close" are at the bottom right of the dialog.

 CUST\_TOOLS\_680

Edit permissions for role “SuperUser” and select the “Lease Cars” box.

## Demo Customization: Case 3 - UI Design Mode



The form is completely empty the first time you run the object form.

## Demo Customization: Case 3 - UI Design Mode



In the Field List box, the BusinessFieldsLabel tab shows two entries:

- Lease Car
- Lease Car Usage

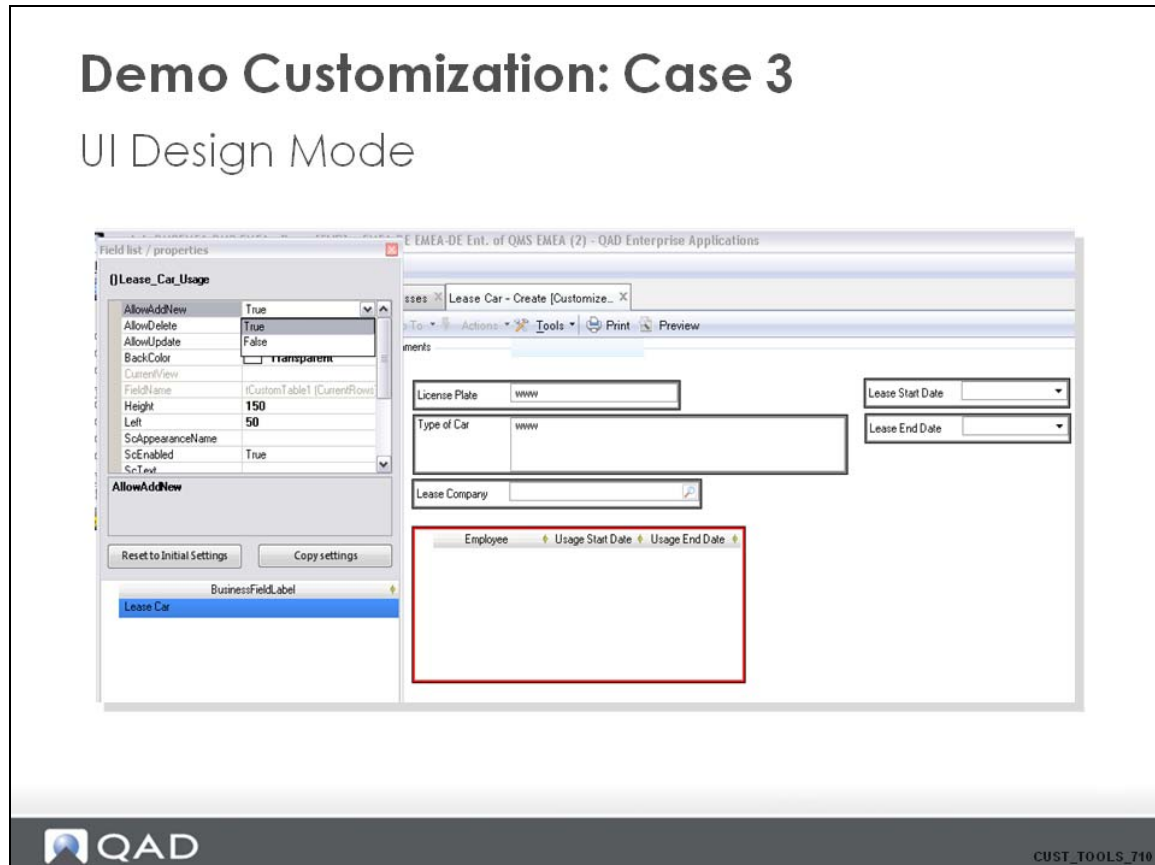
Expand the first tab (Lease Car), then drag and drop all fields under this tab to the maintenance form.

Drag and drop the entire second tab (Lease Car Usage) on to the maintenance form.

This creates a grid control. Right-click on the grid control and select “Columns”.

Select all columns available on the grid.

## Demo Customization: Case 3 - UI Design Mode



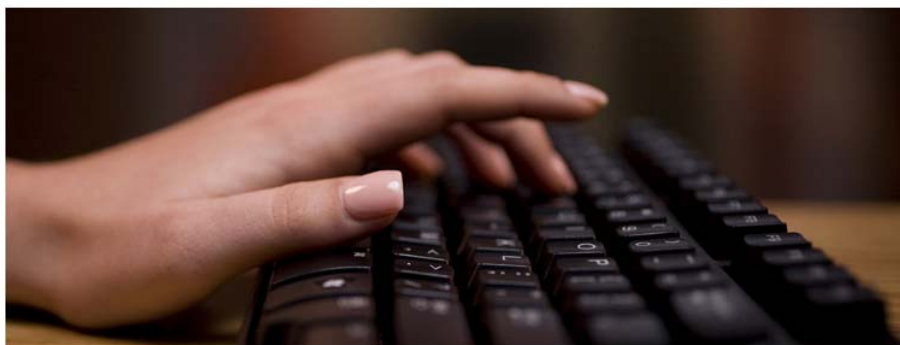
By default, you cannot insert new lines in your grid.

Do not forget to set the “AllowAddNew” property of your grid to “True”.

Select “Design Mode” again to end the design. Make sure to select all of the activities and to save your customization on the “general” level.

## Hands-on Exercise (13)

# Hands-on Exercise (13)



## Additional Validation on Lease Car

## Report Customization

Report Customization



## Report Framework for Financials Reports

### Report Framework for Financials Reports

Three main parts:

- **Business Reporting Component:** Populates the selection criteria fields and runs the queries to create the dataset
- **UI:** Displays the selection criteria, interacts with the Business Reporting Component, and runs CR or QRF to process the report layout
- **Crystal Reports:** A reporting application from a company called Business Objects, or **QAD Reporting Framework**, both used to design the report layout, and display the report at runtime



CUST\_REP\_020

## Report Framework for Financials Reports

### Report Framework for Financials Reports

Development work for reports:

- Creating the Business Component
- Setting up any required translations
- Creating the QRF file or Crystal Reports file (.rpt)



CUST\_REP\_030

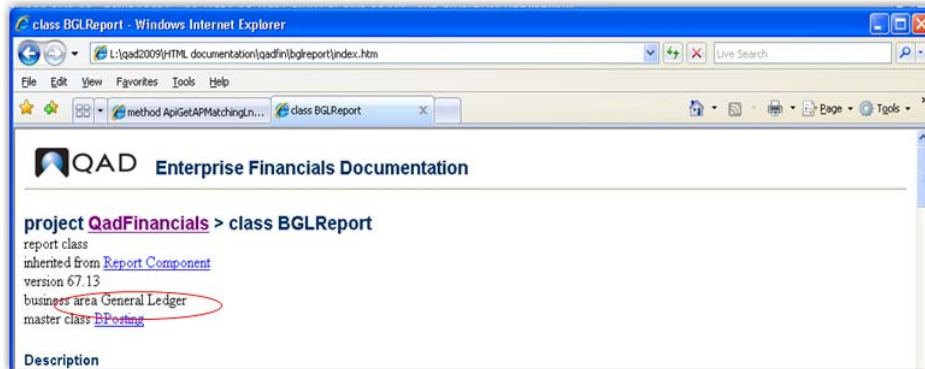
## Report Framework for Financials Reports

# Report Framework for Financials Reports

Report business component development

Inheritance from abstract foundation component

Report Component



## Report Framework for Financials Reports

### Report Framework for Financials Reports

Report business component development

- A report component typically hosts logic for multiple reports. For each report:
  - A report execution method is developed
  - An activity is defined
  - A report dataset is defined
- All of the above reflect the report name in their name
  - The method name is the report name.  
For example, `bglreport.balancebycurrency`
  - The activity name is the report name.  
For example, `bglreport.balancebycurrency`
  - The report dataset is “dcr” + the report name.  
For example, `dcrbalancebycurrency`



CUST\_REP\_050

Report Framework for Financials Reports

# Report Framework for Financials Reports

A report execution method

method BalanceByCurrency - Windows Internet Explorer

QAD Enterprise Financials Documentation

project QadFinancials > class BGLReport > method BalanceByCurrency

report procedure

**Description**  
BalanceByCurrency

**Parameters**

icLanguageCode	input	character	
ifFilter	input	temp-table	
dcrBalanceByCurrency	output	dataset	
oiReturnStatus	output	integer	Return status of the method.

Report method with fixed set of parameters

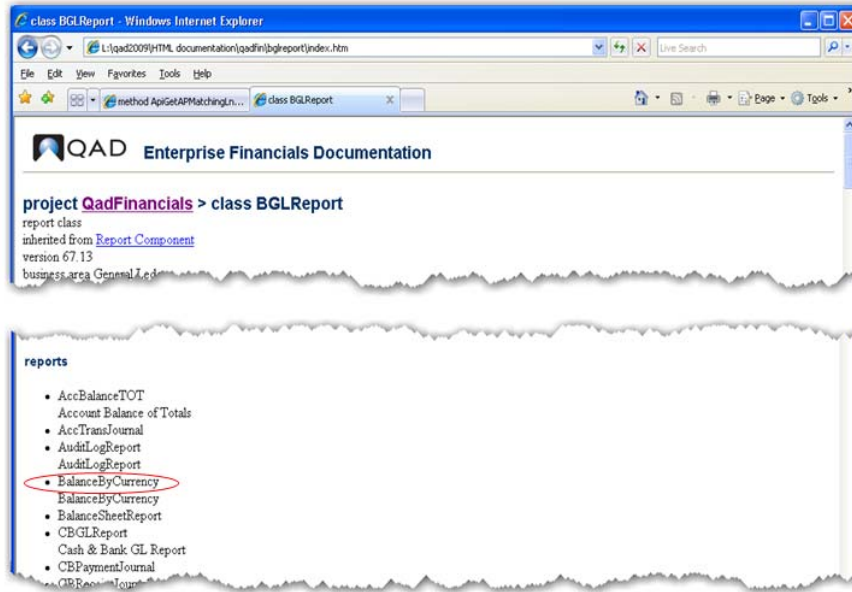


CUST\_REP\_060

Report Framework for Financials Reports

# Report Framework for Financials Reports

An activity



CUST\_REP\_070

## Report Framework for Financials Reports

### Report Framework for Financials Reports

Report business component development.

Implementation of methods:

- `GetBusinessFields`
- `SetDataItemsBasedonFilterTT`
- Report execution method



CUST\_REP\_000

Report Framework for Financials Reports

# Report Framework for Financials Reports

GetBusinessFields

Records in tBusinessFields are created for all possible filter fields of reports

project QadFinancials > class BGLReport > method GetBusinessFields

**Description**  
This method is used to set filters for reports on the UI

**Parameters**

icReference	input	character	-<classname> for receiving business field information of a business c
tBusinessFields	output	temp-table	-<classname> <method-name> for receiving business field informatio

```

if can-do ("GLAccountSheet,TrialBalance,GLHistory,HistoryReports":U, icReference)
then do:
  create tBusinessFields.
  assign tBusinessFields.tcSideLabel      = #T-52'Check If History Is Up To Da
tBusinessFields.tcFcDescription = #T-53'Check If History Is Up To Da
tBusinessFields.tcFcFieldName = "CheckHistory":U
tBusinessFields.tcDataType = "I":U
tBusinessFields.tcControlType = "Bool":U
tBusinessFields.tcInitialValue = "true":U
tBusinessFields.tcDisplayFormat = "yes/no":U
tBusinessFields.tcFcFieldType = "F":U.
end.
    
```

Unique name within the report component

Type always "F"

The developer does not have to create records with tcFcFieldType = "B" (describing the report resultset).



CUST\_REP\_090



## Report Framework for Financials Reports

## Report Framework for Financials Reports

SetDataItemsBasedOnFilterTT

Read the records from the `tqFilter` and fill in the correct data items (variables)

project `QadFinancials` > class `BGLReport` > method `SetDataItemsBasedOnFilterTT`

Description

```
for each tqFilter:  
  case tqFilter.tcBusinessFieldName:  
    when "SortBy":U  
      then do:  
        if tqFilter.tcParameterValue = "Posting Date":U  
          then assign vcSortByFilter = "1" no-error.  
        if tqFilter.tcParameterValue = "Sequence Number":U  
          then assign vcSortByFilter = "2" no-error.  
        end.  
    when "AccYear":U  
      then assign viFromAccYearFilter = int(tqFilter.tcParameterValue) no-error.  
    when "AccPeriod":U  
      then assign viFromAccPeriodFilter = int(tqFilter.tcParameterValue) no-error.
```



CUST\_REP\_100

Report Framework for Financials Reports

# Report Framework for Financials Reports

Report execution method

- Called when the report is executed from the UI or from the report service (in batch)
- Language passed at runtime (using the report options)

project QadFinancials > class BGLReport > method BalanceByCurrency  
report procedure

**Description**  
BalanceByCurrency

**Parameters**

icLanguageCode	input	character
iFilter	input	temp-table
derBalanceByCurrency	output	dataset
oiReturnStatus	output	integer

Return status of the method



## Report Framework for Financials Reports

## Report Framework for Financials Reports

Report execution method

Typically is code that calls one or more queries and based on that data populates the report dataset

```

for each ttSelectedPeriod
  by ttSelectedPeriod.tiCompany_ID by ttSelectedPeriod.tiPer
  <Q-2 run BalanceByCurrencyMR (all) (Read) (NoCache)
  (input vGLCurrentCompany_ID, (CompanyId)
  input vLayerCandoFilter, (LayerCode)
  input vcDivisionFilter, (DivisionCode)
  input vcFromGLFilter, (FromGLCode)
  input vcToGLFilter, (ToGLCode)
  input vGLIsActiveFilter, (GLIsActive)
  input vGLIsBalanceAccountFilter, (GLIsBalanceAccount)
  input vcCurrencyCandoFilter, (CurrencyCode)
  input ttSelectedPeriod.tiPeriod_ID, (Period_ID)
  output dataset tqBalanceByCurrencyMR) in BGLReport >

  for each tqBalanceByCurrencyMR:
    create tBalanceByCurrency.
    buffer-copy tqBalanceByCurrencyMR to tBalanceByCurrency.
    assign tBalanceByCurrency.tiCompanyCode = vGLCurrentCompany.
  end.
end. /* for each ttSelectedPeriod */

```

Query call

Create record in report dataset

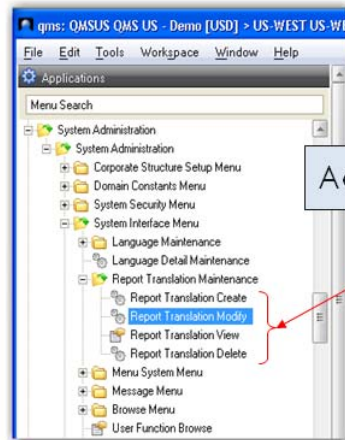
## Report Framework for Financials Reports

## Report Framework for Financials Reports

### Languages and Translations

#### BReportTranslation component

- This component is used to maintain the labels and translations on specific reports
- Menu items:



Activities on the menu

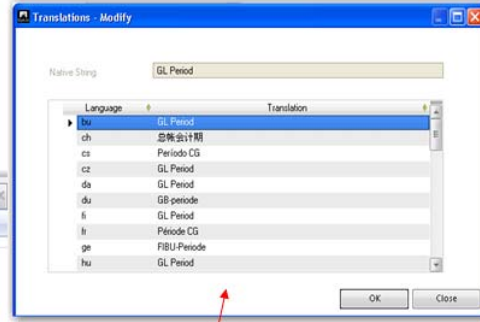
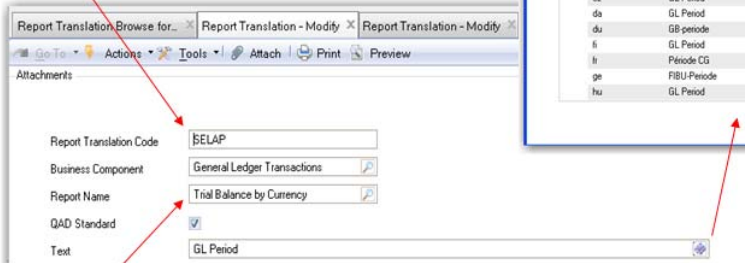
## Report Framework for Financials Reports

# Report Framework for Financials Reports

Languages and Translations

Report translation screen:

Code, referenced from within the CR Report design



Report name reference  
If empty available for all reports in the component



CUST\_REP\_140

## Report Framework for Financials Reports

### Report Framework for Financials Reports

Languages and Translations

TReportStrings

- Contains the official fixed text labels over all reports in the application (for example “filter”, “date”, “total”, “user”, and so on)
- The strings are all in source code (no maintenance from the application screen, only via non-intrusive customization)



CUST\_REP\_150

## Report Framework for Financials Reports

# Report Framework for Financials Reports

## Languages and Translations

- `TReportStrings.MainBlock` method contains the code
- Translation is fixed and comes with the QAD application installation
- Strings are referenced from Crystal reports in a similar way as the specific report labels

```
create tReportStrings.  
assign tReportStrings.tcStringText = #T-3'Date':10(358)t-3#  
tReportStrings.tcStringCode = "0-DATE":U.  
  
create tReportStrings.  
assign tReportStrings.tcStringText = #T-4'DB':2(359)t-4#  
tReportStrings.tcStringCode = "0-DB":U.  
  
create tReportStrings.  
assign tReportStrings.tcStringText = #T-5'End of Report':20(360)t-5#  
tReportStrings.tcStringCode = "0-EOR":U.  
  
create tReportStrings.  
assign tReportStrings.tcStringText = #T-6'Filter Values for this Report':100(361)T-6#  
tReportStrings.tcStringCode = "0-FILTER":U.  
  
create tReportStrings.  
assign tReportStrings.tcStringText = #T-7'No Data':20(362)t-7#  
tReportStrings.tcStringCode = "0-ND":U.  
  
create tReportStrings.  
assign tReportStrings.tcStringText = #T-8'Page':10(363)t-8#
```



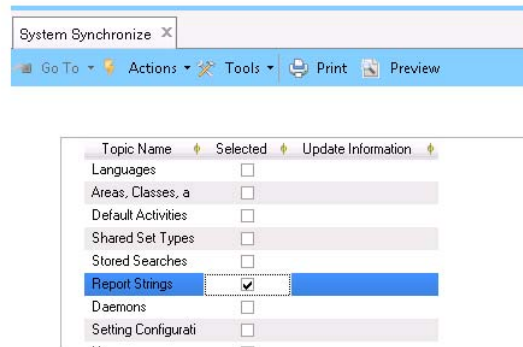
CUST\_REP\_160

## Report Framework for Financials Reports

# Report Framework for Financials Reports

For all of the strings defined in `TReportStrings` component, do the following:

1. System Synchronize (36.24.3.2) for Report Strings
2. Log out from the application and log in again

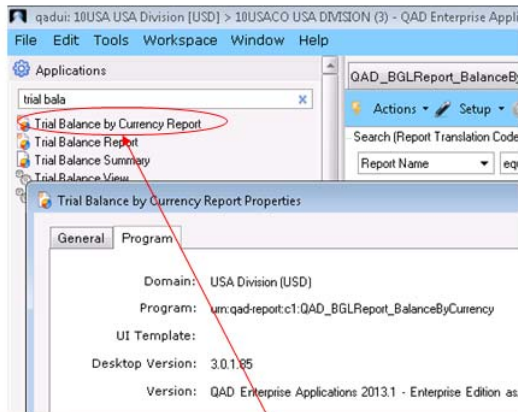


Report Framework for Financials Reports

# Report Framework for Financials Reports

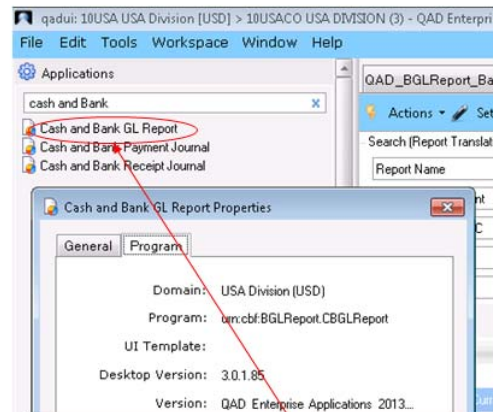
Report execution, report activities

## Reporting Framework



Report activity on the menu.  
 Standard QRF URN:  
 "urn:qad-report:c 1:<report\_name>"

## Crystal Reports



Report activity on the menu.  
 Standard Financials URN:  
 "urn:cbf:<component>.activity"



CUST\_REP\_180

Report Framework for Financials Reports

# Report Framework for Financials Reports

## Generic UI report execution interface

The screenshot shows a web-based interface for configuring a report. At the top, there's a title bar with the text "QAD\_BGLReport\_BalanceByC...". Below it is a menu bar with options like "Filter", "Viewer", "New Filter", "Open", "Save", "Save As", "Delete", "Settings", "Layout", and "Run". The main area is titled "Search Conditions" and contains a list of fields with dropdown menus and input boxes. The fields include: "Amounts in SC", "Balance/P/L", "Currency", "Entity" (set to "10USACD (10USA)"), "GL Account", "GL Cal Year", "GL Period", "Only Accounts with A", and "Subtotals". Each field has a dropdown menu and a search icon. At the bottom of the interface, there is a blue box with the text "Reporting Framework".

The screenshot shows a web-based interface for configuring a report. At the top, there's a title bar with the text "Cash and Bank GL Report". Below it is a menu bar with options like "Tools", "Print", "Preview", and "Variant". The main area is titled "Report Variants" and contains a list of filter options. The filters include: "Entity" (with a list of options like "10CORPCONS (10USA)", "11CANCO (11CAN)", "11NACONS (11EAN)", "12MEXCO (12MEX)"), "GL Account", "Posting Date" (set to "10/29/2013"), "Include Sub-Account", "Sub-Account", "Include Cost Center", "Cost Center", "Currency", "Print with Labels", "Alternate Account", "COA Cross Reference", "Include Project", and "Layer". At the bottom of the interface, there is a blue box with the text "Crystal Reports".



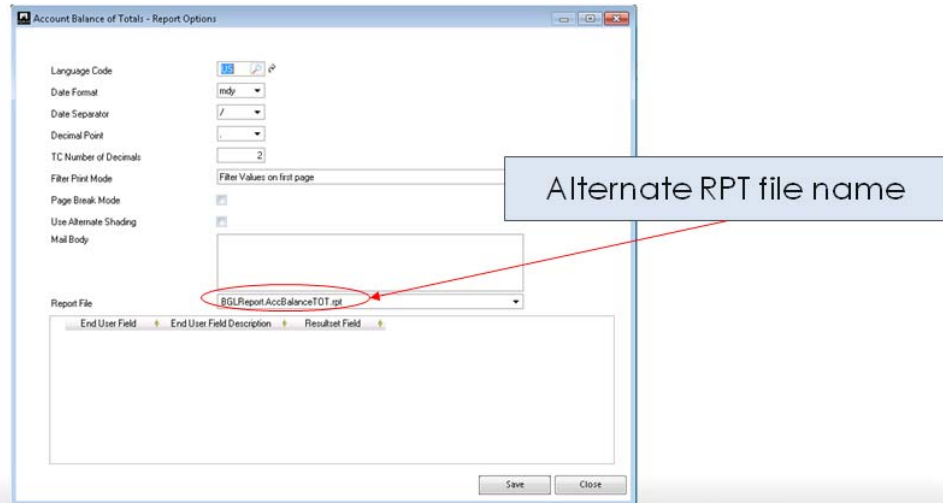
CUST\_REP\_190

## Report Framework for Financials Reports

# Report Framework for Financials Reports

Generic UI report execution interface **CR**

Use Tools->Report options... to pass settings for the report execution

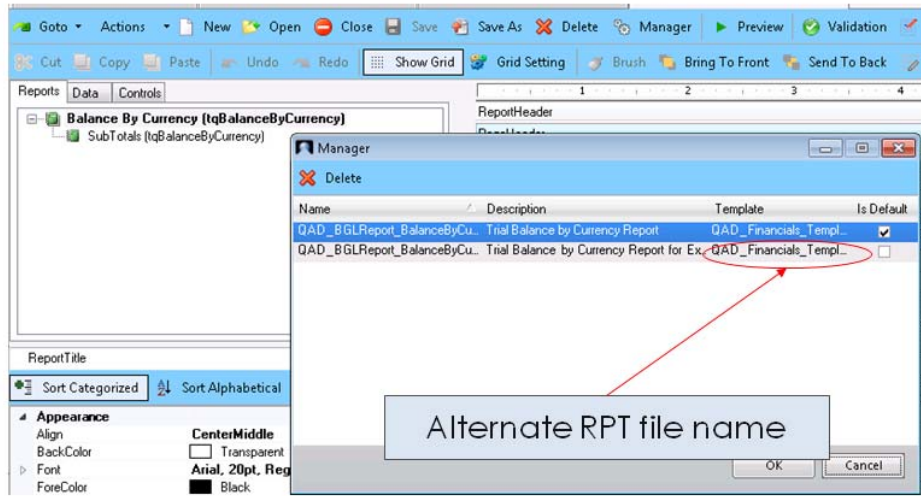


Report Framework for Financials Reports

# Report Framework for Financials Reports

Generic UI report execution interface **QRF**

Use Manager option to select a default layout



## Report Framework for Financials Reports

# Report Framework for Financials Reports

Generic UI report execution interface

- Settings can be saved as variants.  
(Variant->Save as...)  
Saved data: Report options and filter field settings
- System comes with factory default variant for each report

Account Balance of Totals - Save Variant As

Name: FACTORYDEFAULT

Description: LASTUSED\_Jacdel

Level: System

Role Name: [Dropdown]

Entity Dependent:

Customer Default:

Save Close

## Report Framework for Financials Reports

# Report Framework for Financials Reports

## Deployment

- Reports (RPT files) deployed on server
  - `crreports` folder
  - Client transfers reports when necessary (new or changed report)
- You can run reports in real-time mode or batch
  - This is specified in the generic interface for starting a report
  - Reports can also be scheduled



CUST\_REP\_220

Locate the following folder on the server in the training environment:

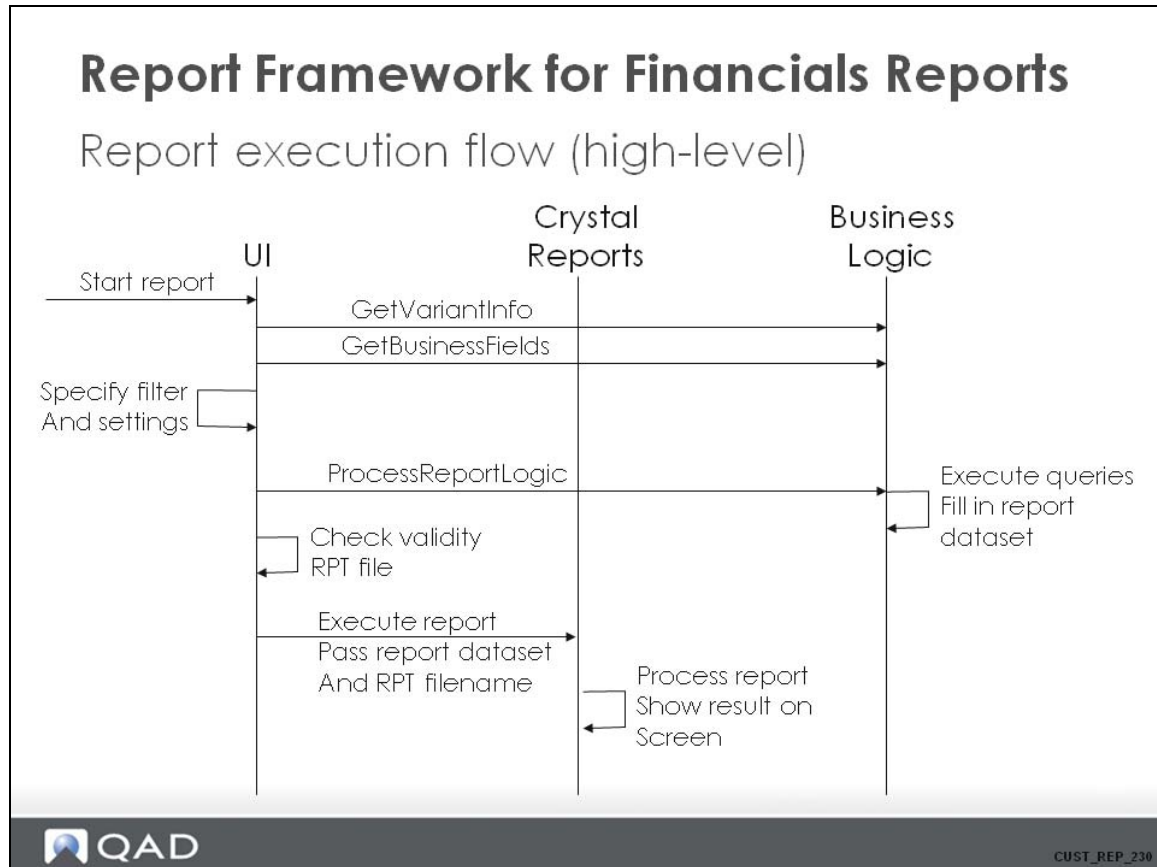
`“/dr01/qad2009/qms/fin/crreports”`.

Reports are named `“<ReportComponent>.<ReportName>.rpt”`.

Place customized reports in a different folder. This folder is specified in the `server.xml` file, under the `“<customreports>”` tag. In the training environment, it is set to `$(ENVROOT)/customreports`. Define `$(ENVROOT)` in this `server.xml` file.

The client transfers the RPT files only when they are to be executed. These files are cached in a client-side folder (`plugins\qad.plugin.Financials\Reports`).

Report Framework for Financials Reports



See “Customization of Component-Based QAD Applications – Architecture” for more details on the Business Logic flows for starting and executing reports.

## Report Customizations

### Report Customizations

- Customize the backend BL
- Use the normal NI customization in 4GL to do one of the following:
  1. Implement
 

```
<ReportComponent>.GetTableCustomFields  
After()
```

This is done if user-defined fields of related components must be available for selection on the report
  2. Implement
 

```
<ReportComponent>.ProcessReportLogicAfter()  
or <ReportComponent>.<NameOfReport>.After()
```

Do this if you must implement Customization hard-coded assignments of user-defined fields



CUST\_REP\_240

There are two approaches for customizing the report backend. One approach is to write code to expose one or more UDFs of one or more business components to the report so that they are available for selection when the report is executed. The other approach is to write code that automatically fills in UDFs in the report result set with “calculated” values, which might be fields from other tables.

The user-defined fields for the report are selected in the report options form when the report is executed.

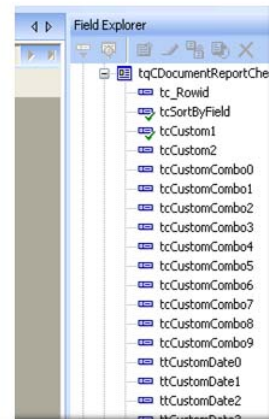
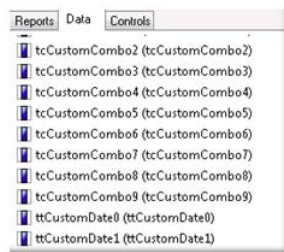
## Report Customization - User-Defined Fields

### Reports Customization

User-defined fields (Custom Fields)

- The custom fields are just used as normal data fields in reports
- On BL side, ensure the correct values are assigned for custom fields

Reporting Framework



Crystal Reports



CUST\_REP\_250

## Report Customization - Debugging

### Reports Customization

#### Debugging

- Output the report data to XML
- Add the following sentence into the file  
    `"..\Client\container\homeserver.config":`  
    `<add key="UIDebugEnabling" value="on" />`
- Log in to the application and run any report.  
    The file `lastreport.xml` containing the report data generated under the directory where you start the application
- CT Log

**Hands-on Exercise (14 + 15 + 16)**

## Hands-on Exercise (14 + 15 + 16)



### Customer Aging Analysis Report



CUST\_REP\_270

Chapter 4

# **Integration with Enterprise Financials: Backend**

## Integrating with Financials Backend

### Integrating with Financials Backend

- You can do integration:
  - Through QXtend  
*Currently only available for DataSync*
  - Using public API methods

- Purpose of this training

Following this training, you will be able to write an integration to Enterprise Financials in Progress 4GL using 4GL proxies provided by QAD



CUST\_JIT\_030

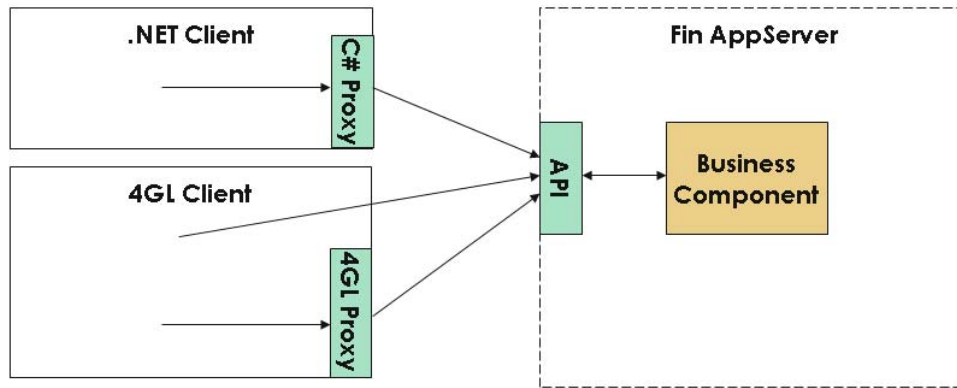
Refer to the QXtend training for information on how to set up and configure data sync for Financials data using QXtend.

## Calling the API

### Calling the API

Ways to call a component API:

- Using the 4GL proxy layer
- Using direct API calls to the AppServer
- Calls in C# through the C# proxy layer



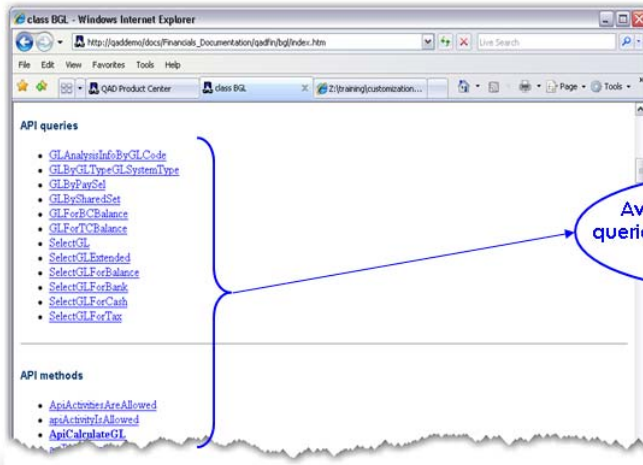
Refer to *Calling API Documentation* for more detailed information and sample code.

## Calling the API

## Calling the API

Available API methods:

- Use HTML documentation
- Look for API queries and methods on the main page of a business component



## Calling the API

## Calling the API

Documentation of API methods

- The documentation shows the parameters used internally. For API calls, more parameters are necessary

- For the proxy implementation:

```
define input parameter ic_CompanyCode as character
<parameters from documentation>
define output parameter dataset for tFcMessages
define output parameter oiReturnStatus as integer no-undo
```

- For the direct API implementation:

```
define input parameter icApiLogin as character
define input parameter icApiPassword as character
define input parameter icApiExtra as character
define input parameter iiApiSessionId as integer
<parameters from documentation>
define output parameter dataset for tFcMessages
define output parameter oiReturnStatus as integer no-undo
```



CUST\_IIT\_060

## Calling the API

## Calling the API

Using 4GL proxy layer – Set up

Configure connection in `cbserver.xml` in the client  
PROPATH

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file contains the configuration of the business layer of the application -->
- <serverConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="serverConfiguration.xsd">
  <!-- sessionInfoGetterProcedure can be used to override the default value of env.p -->
  <sessionInfoGetterProcedure>env.p</sessionInfoGetterProcedure>
  <!-- disable authentication for time being -->
  <!-- <DisableAuthentication/> -->
  <!-- A valid appserver needs to be specified -->
  - <appServerConnection>
    - <appService name="dev">
      <appServiceURL>appserver://qaddemo:5162/qadfinlive</appServiceURL>
    </appService>
    <DebugLevel />
  </appServerConnection>
</serverConfiguration>
```



CUST\_IIT\_070

## Calling the API

## Calling the API

Using 4GL proxy layer – Set up

- Implement the `infogetter` procedure  
By default called `env.p` in the client `PROPATH`

```
define output parameter ocGlobalSessionId as character NO-UNDO INIT "".
define output parameter ocUserLogin as character no-undo.
define output parameter ocPassword as character no-undo.
define output parameter ocCompany as character no-undo.

ASSIGN ocUserLogin = "mfg"
       ocPassword = ""
       ocCompany = "US-EAST".
```

- For the client session make sure the `PROPATH` points to the directory or procedure library containing the `proxy` subfolder



CUST\_INIT\_000

In the training environment, the `PROPATH` should point to “/dr01/qadapps/qea/fin/proxysrc”.

## Calling the API

## Calling the API

Using 4GL proxy layer – Implementation

- Include the correct definitions

```
{ proxy/bgldf.i }
```

- Prepare input data

```
CREATE tFilter.  
ASSIGN tFilter.tcBusinessFieldName = "tgl.GLCode"  
       tFilter.tcDataType = "c"  
       tFilter.tcOperator = "matches"  
       tFilter.tcParameterValue = "1".
```

- Start proxy persistently

```
run proxy/bg1.p persistent set vhProxyComponent.
```

## Calling the API

## Calling the API

Using 4GL proxy layer – Implementation

- Run the API method

```
run SelectGl in vhProxyComponent ("", /* Company code */
                                "A", /* range */
                                "", /* start from rowid */
                                0, /* start from row number */
                                50, /* number of rows to retrieve */
                                "", /* sort columns */
                                FALSE, /* only counting */
                                TRUE, /* forward read */
                                0, /* maximum rows to count */
                                DATASET tFilter,
                                OUTPUT viCount,
                                OUTPUT viEq,
                                OUTPUT DATASET tqSelectGl,
                                OUTPUT DATASET tFcMessages,
                                OUTPUT viReturn).
```

- Stop persistent proxy

```
delete procedure vhProxyComponent.
```

- Error handling

```
IF viReturn <> 0
THEN FOR EACH tFcMessages:
    MESSAGE tFcMessages.tcFcMessage
        VIEW-AS ALERT-BOX INFO BUTTONS OK.
END.
```

## Calling the API

### Calling the API

Alternative – Use the predefined includes

```
{ proxy/bg1/selectgldef.i }  
{ proxy/bg1/selectglrun.i }
```

(These do not include error handling)



CUST\_JIT\_110

## Calling the API

## Calling the API

Specific for API queries

- API queries get the filter conditions in a generic way, using the `tFilter` table in the `tFilter` dataset
- `tFilter` dataset as input can be used to pass conditions to the query execution

<code>tcBusinessFieldName</code>	character	Name of the business field (as returned by <code>GetBusinessFields(&lt;query&gt;)</code> ).
<code>tcDataType</code>	character	One-letter indication of data type (c,d,t,i,l (resp. character, decimal, date, integer or logical))
<code>tcOperator</code>	character	Operator ("=", "matches", "begins", ">", ">=", "<=", "<", "range")
<code>tcParameterValue</code>	character	ParameterValue



CUST\_IIT\_120

For example:

```

create tFilter.
assign tFilter.tcBusinessFieldName = "iiCompanyID"
      tFilter.tcDataType = "i"
      tFilter.tcOperator = "="
      tFilter.tcParameterValue = STRING(9144).

create tFilter.
assign tFilter.tcBusinessFieldName = "icGLCode"
      tFilter.tcDataType = "c"
      tFilter.tcOperator = "="
      tFilter.tcParameterValue = "2200".

create tFilter.
assign tFilter.tcBusinessFieldName = "icDomainCode"
      tFilter.tcDataType = "c"
      tFilter.tcOperator = "="
      tFilter.tcParameterValue = "domain1".

```

## Calling the API

## Calling the API

### Specific for API queries

- API queries have the same additional parameters for straight API and Proxy calls
- Other standard parameters:
  - Input `icRange`
  - Input `icRowid`
  - Input `iiRowNumber`
  - Input `iiNumberOfRows`
  - Input `icSortColumns`
  - Input `ilCountOnly`
  - Input `ilForwardRead`
  - Input `iiMaximumRowsToCount`
  - Input `dataset tFilter`
  - Output `viCount`
  - Output `vLEoq`
  - Output `dataset tq<queryName>`
  - Output `oiReturnStatus`



CUST\_HIT\_130

With:

- `icRange`: The range to read: “A” (all), “F” (first) or “L” (last).
- `icRowid`: Start reading with this rowid (reposition first before start reading) (if = " then start at beginning).
- `iiRowNumber`: The number of the row to start reading (if = 0 then start at beginning).
- `iiNumberOfRows`: The number of rows to read (if = 0 then all is read).
- `icSortColumns`: Comma-separated list of fields on which the result list needs sorting.
- `ilCountOnly`: Default false. If true, only count, so the result dataset is not filled in, and only the `viCount` is returned.
- `ilForwardRead` (default true): Read forward through the query
- `iiMaximumRowsToCount`: Stop counting after number of rows (if = 0, NO COUNTING occurs).
- `Dataset tFilter`: The condition for the query (see HTML documentation for the specific query).
- `viCount`: The number of records counted.
- `vLEoq`: End of query reached. This is true if the last record matching the conditions was read from the database.
- `Tq<QueryName>`: The result dataset.

For example:

```
run SelectG1 in vhProxyComponent ('', /* Company code */
    'A', /* range */
    '', /* start from rowid */
    0, /* start from row number */
    50, /* number of rows to retrieve */
    '', /* sort columns */
    FALSE, /* only counting */
    TRUE, /* forward read */
    0, /* maximum rows to count */
    DATASET tFilter,
    OUTPUT viCount,
    OUTPUT vlEq,
    OUTPUT DATASET tqSelectG1,
    OUTPUT DATASET tFcMessages,
    OUTPUT viReturn).
```

## Calling the API

## Calling the API

- Use the HTML documentation to find the right API method. Some guidelines:
  - For data retrieval: API Queries
  - For updates: `APIMaintainByDataset` and `APIMaintainByDatasetWithOutput`
- Example: Dump & Load UI customizations



CUST\_HIT\_140

Use the HTML documentation for the classes to find the right method or query.

`ApiMaintainByDataset` is typically used for updates. This method has the object dataset as an input parameter. It generically creates data that is not found in the database, and updates the data it finds (using the alternate key: the object dataset is not expected to have ID fields filled in). It does not delete data.

The `ApiMaintainByDatasetWithOutput` method is the same method that is used for QXtend inbound integration, and the XML daemon. This method works only if the component has the `DataLoadByInput` method implemented. This can be checked in the HTML documentation by looking at the code.

See the Solutions document for a description of what to do as preparation for API calls (`cbserver.xml` and `env.p` files).

## Calling the API

## Calling the API

Typical implementation of a call to create an object

```

{ PROXY/bcountrydef.i }
run proxy/bcountry.p persistent set vhProxyComponent.

CREATE tCountry.
ASSIGN tCountry.CountryCode = "QZI"
       tCountry.CountryDescription = "Test country for QZI"
       tCountry.tc_rowid = "777".
CREATE tCountryVatFormat.
ASSIGN tCountryVatFormat.CountryVatFormat = "QZI Det 1"
       tCountryVatFormat.tc_parentrowid = "777"
       tCountryVatFormat.tc_rowid = "778".

Run apiMaintainByDatasetWithOutput IN vhProxyComponent
(INPUT "", /* company code */
 INPUT "save", /* action */
 INPUT TRUE, /* return dataset */
 INPUT TRUE, /* partial update */
 INPUT "", /* icPartialUpdateExceptionList */
 INPUT DATASET BCountry BY-REFERENCE,
 OUTPUT ocPrimaryKey,
 OUTPUT ocRowid,
 OUTPUT oiDraftInstance,
 OUTPUT ocPrimaryKeyName,
 OUTPUT dataset-handle who,
 OUTPUT dataset tFMessages,
 OUTPUT viReturn).

```



CUST\_HIT\_150

This example uses the `apiMaintainByDatasetWithOutput` method to create a Country object on the Financials business layer.

Do the following on the client caller to implement this:

1. Include the definitions for everything required on the client side proxy call.
2. Run the proxy method persistently.
3. Create the records that hold the information of the country object that needs to be created. Remark the `tc_rowid` and `tc_parentrowid`. These fields must be filled in because this is how the backend knows what information belongs together. In this simple sample, we are creating only one object. Of course, it is possible to create a whole set of objects in one call.
4. Run the `ApiMaintainByDatasetWithOutput` method.

## Calling the API

## Calling the API

Typical implementation of a call to create an object

```

*/
DATASET tFcMessages:WRITE-XML("file",
    "C:\develop\work\Integration\foundation\api\Output\vhEx
    true,
    ?,
    ?,
    FALSE,
    NO).

IF VALID-HANDLE(vh0) THEN
    vh0:WRITE-XML("file", "C:\develop\work\Integration\foundation\api\Output\vh0.x
  
```

5. Use the information in tFcMessages to look for any exception messages coming back from the business logic.
6. Use the output dataset to look for the exact data that was written to the database for the new object.

## Calling the API

## Calling the API

Specific information about the `apiMaintainByDataset` and `apiMaintainByDatasetWithOutput`

- These methods are generically available for all business components responsible for access to one or more database tables  
BUT, these only work when `DataLoadByInput` is implemented on the component
- Biggest differences:
  - `apiMaintainByDataset` expects the full dataset as input (data and structure). This is not the case for `apiMaintainByDatasetWithOutput`
  - `apiMaintainByDataset` does not return the object created by the call



CUST\_IIT\_170

### `apiMaintainByDataset`:

- Expects an input dataset that exactly matches the structure of the object dataset.
- Automatically detects if the object needs to be created or modified based on the alternate key values.
- Expects the full object data. Automatically deletes child records that exist in the database, but do not exist in the input dataset.
- Cannot be used to delete specific detail lines (child records).
- Cannot be used to add specific detail lines.
- Does not return the created/modified/deleted objects. It only returns a result via the `oiReturnStatus`.

### `apiMaintainByDatasetWithOutput`:

- If `ilPartialUpdate` is true:
  - Expects an input dataset that can be a subset of the object dataset what concerns the structure.
  - For modification of objects, the data passed to the method does not need to be complete. The bare minimum is the alternate key fields of the main table in the object.
  - If fields are passed with value ? (unknown value / null), they are not updated in the database (an exception can be made by using `icPartialUpdateExceptionList`).

- Value of “tc\_rowid” is taken into account for deletion of child records. If the tc\_rowid is “D”, the logic tries to delete the record.
- This method returns the object dataset if ilReturnDataset is set to true.

## Calling the API

### Calling the API

Specific information about the `apiMaintainByDataset` and `apiMaintainByDatasetWithoutOutput`

- The `*WithoutOutput` method is used by QXtend inbound for data synchronization and by the XML daemon to load records in the system
- These methods are the prescribed way of passing data into the business layer (without using the UI)

For specific business components, more specialized API methods might exist



CUST\_IIT\_180

## UI Customization Method

### UI Customization Method

#### Datasets for data exchange

##### In the BLF

- tUIInput
- tUIOutput

##### In the UI (C#)

- QAD.BLF\_IF.tUIInputDataSet
- QAD.BLF\_IF.tUIOutputDataSet



You can use the UI Customization method to transfer data between the user interface and business logic layer.

## UI Customization - UI Side

## UI Customization Method – UI Side

The screenshot displays a code editor window titled "Customer Create - Custom code for Event Leave of Control debtorCodeLabelText". The code is as follows:

```

404 private void debtorCodeLabelText_Leave(object sender, System.EventArgs e)
405 {
406     // Custom Code Start
407     QAD.BLF_IF.tUIInputDataSet tUIInputDS = new QAD.BLF_IF.tUIInputDataSet();
408     QAD.BLF_IF.tUIOutputDataSet tUIOutputDS;
409     tUIInputDS.tUIInput.AddUIInputRow(
410         "ControlName",
411         "PropertyName",
412         "PropertyValue");
413
414     tUIOutputDS = this.Adapter.UICustomization(
415         "icControlName",
416         "icEventName",
417         tUIInputDS,
418         this.ProcessObject);
419
420     // Custom Code End
421 }

```

The code editor includes a "Check Syntax" button and "OK" and "Close" buttons. A warning message at the top of the editor states: "Warning: Do not change any code outside of the // Custom Code Start and // Custom Code End lines. All code changed outside of those lines will be discarded."

Create data in the UI and use it in the BLF.

## UI Customization - BLF Side

### UI Customization Method – BLF Side

```
PROCEDURE BDebtor.UICustomization.before:  
  for each tUIInput  
    where tUIInput.tcControlName = "ControlName"  
    and tUIInput.tcPropertyName = "PropertyName"  
  no-lock:  
    vcVariable = tUIInput.tcPropertyValue.  
  end.  
  
  create tUIOutput.  
  assign  
    tUIOutput.tcControlName = "ControlName"  
    tUIOutput.tcPropertyName = "PropertyName"  
    tUIOutput.tcPropertyValue = "Value to be retrieved by UI"  
  .  
END PROCEDURE.
```

Retrieve values from UI

Create values to retrieve in UI



Retrieve the data from the UI and create data to send to the UI.

## UI Customization - UI Side

## UI Customization Method – UI Side

The screenshot displays the QAD software interface for UI customization. On the left, the 'Properties' pane shows the 'Events' tab with 'Leave' selected, associated with the code 'QAD.BLF\_I'. The main window shows the 'Customer Create' form with fields for 'Customer Code', 'Business Relation', 'Active', and 'Bill-To Customer'. Below the form, a code editor window titled 'Customer Create - Custom code for Event Leave of Control debtorCodeLabelText' contains the following C# code:

```

412     "PropertyValue");
413
414     tUIOutputDS = this.Adapter.UICustomization(
415         "icControlName",
416         "icEventName",
417         tUIInputDS,
418         this.ProcessObject);
419
420     foreach (QAD.BLF_IF.tUIOutputDataSet.tUIOutputRow row1 in tUIOutputDS.tUIOutput)
421     {
422         if (row1.tcControlName == "ControlName"
423             && row1.tcPropertyName == "PropertyName")
424         {
425             VALUE = row1.tcPropertyValue;
426         }
427     }
428     // Custom Code End
429 }

```

At the bottom of the code editor window, there are buttons for 'Check Syntax', 'OK', and 'Close'. The QAD logo is visible in the bottom left corner of the screenshot.

Retrieve the data from the BLF.



# Product Information Resources

QAD offers a number of online resources to help you get more information about using QAD products.

[QAD Forums \(community.qad.com\)](http://community.qad.com)

Ask questions and share information with other members of the user community, including QAD experts.

[QAD Knowledgebase \(knowledgebase.qad.com\)\\*](http://knowledgebase.qad.com)

Search for answers, tips, or solutions related to any QAD product or topic.

[QAD Document Library \(www.qad.com/documentlibrary\)](http://www.qad.com/documentlibrary)

Get browser-based access to user guides, release notes, training guides, and so on; use powerful search features to find the document you want, then read online, or download and print PDF.

[QAD Learning Center \(learning.qad.com\)\\*](http://learning.qad.com)

Visit QAD's one-stop destination for all courses and training materials.

\*Log-in required

