QAD Enterprise Applications
Enterprise Edition

Training Guide

QAD Customization

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

Customization_TG_v2015EE.pdf/biw/mdf

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

Contents

QAD Customization Change Summary

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

Date/Version	Description	Reference
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

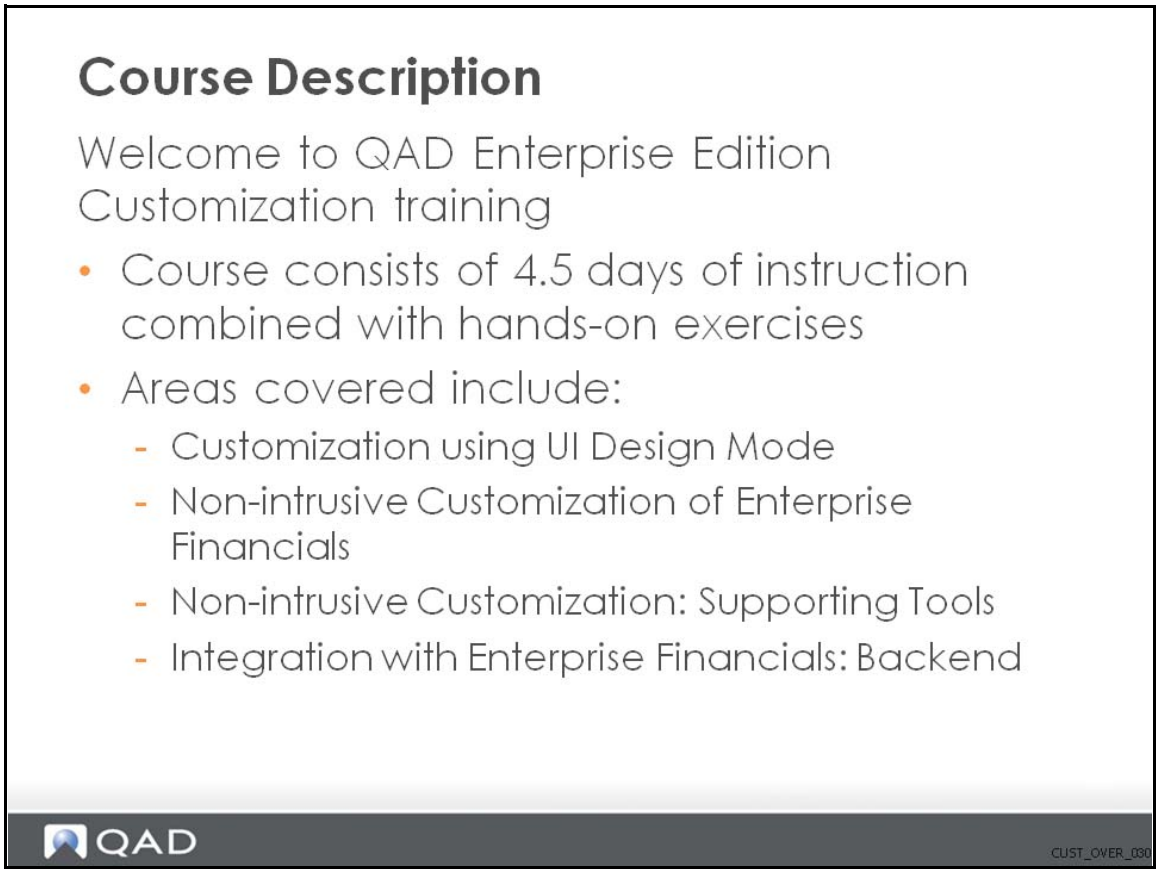
From QAD's main site, you can access QAD's Learning or Support sites.

http://www.qad.com/

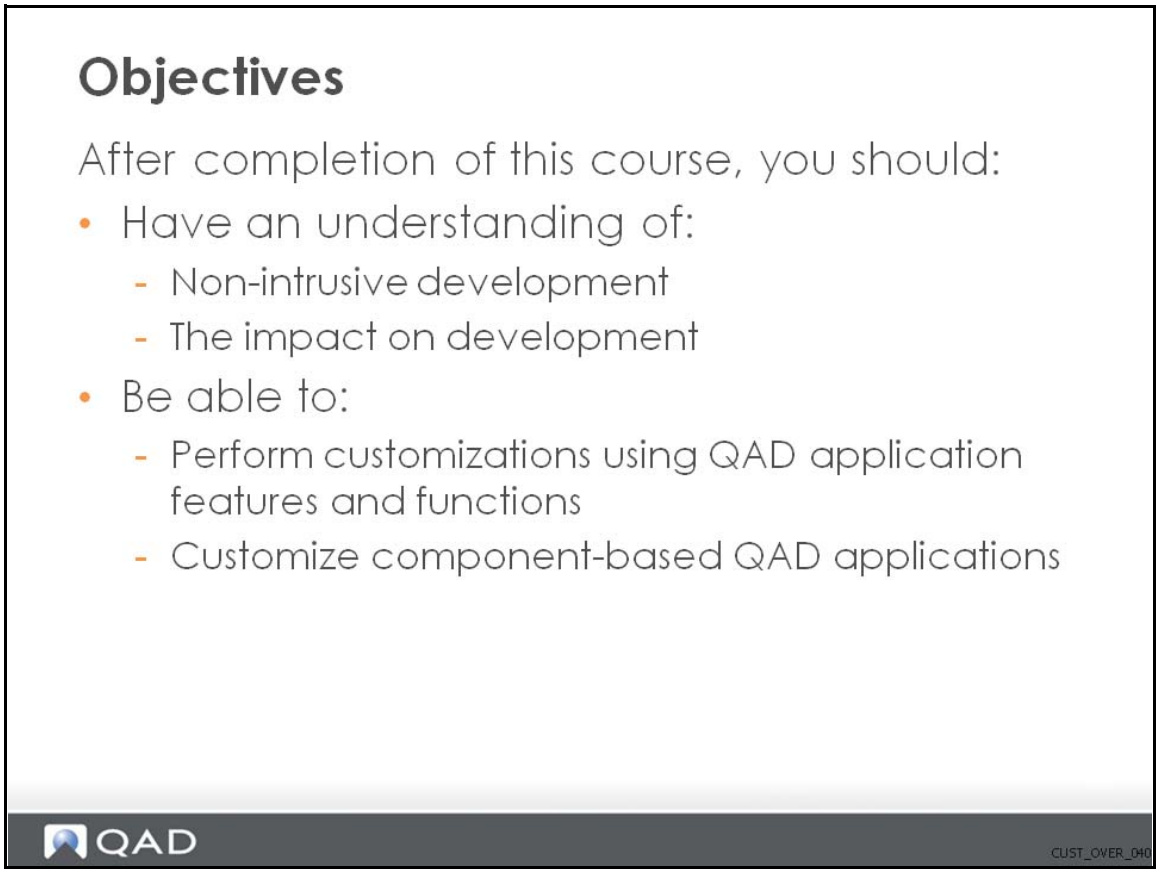
Chapter 1

Course Overview

Course Description

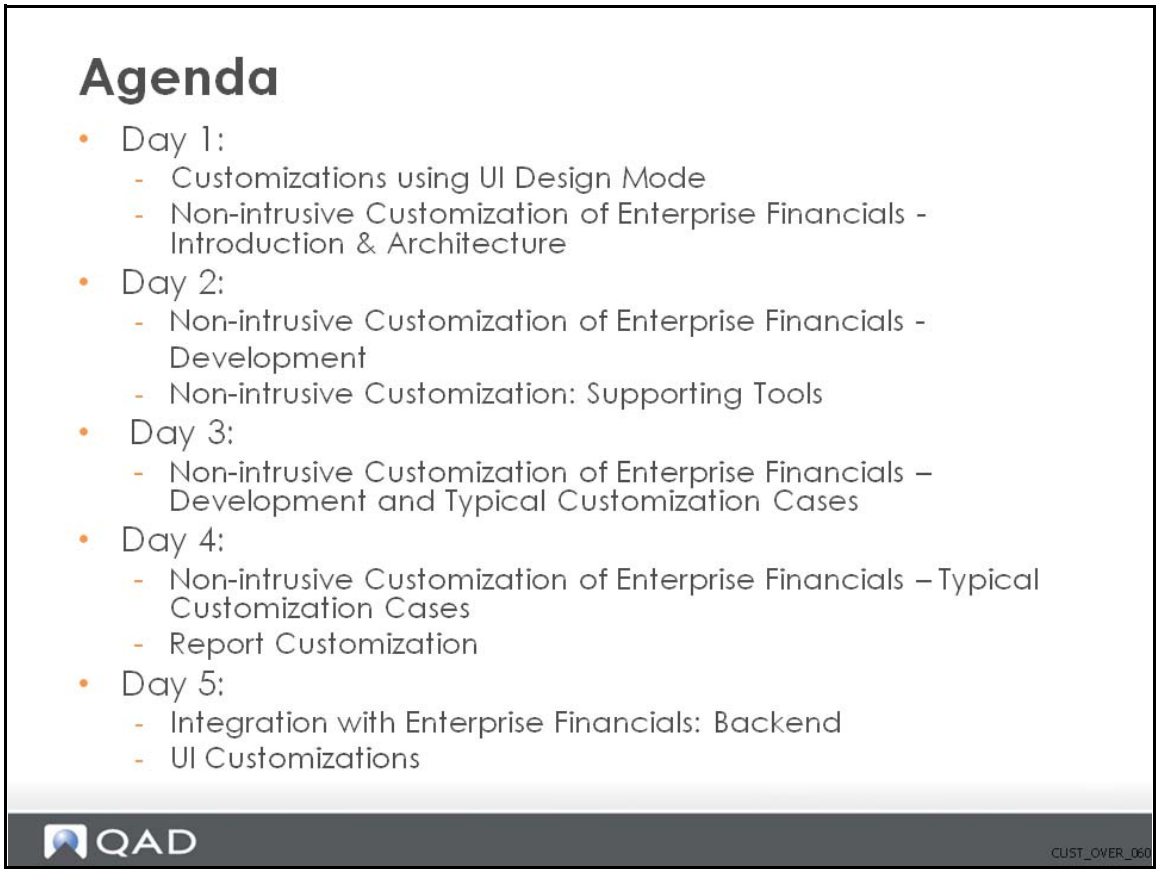


Objectives

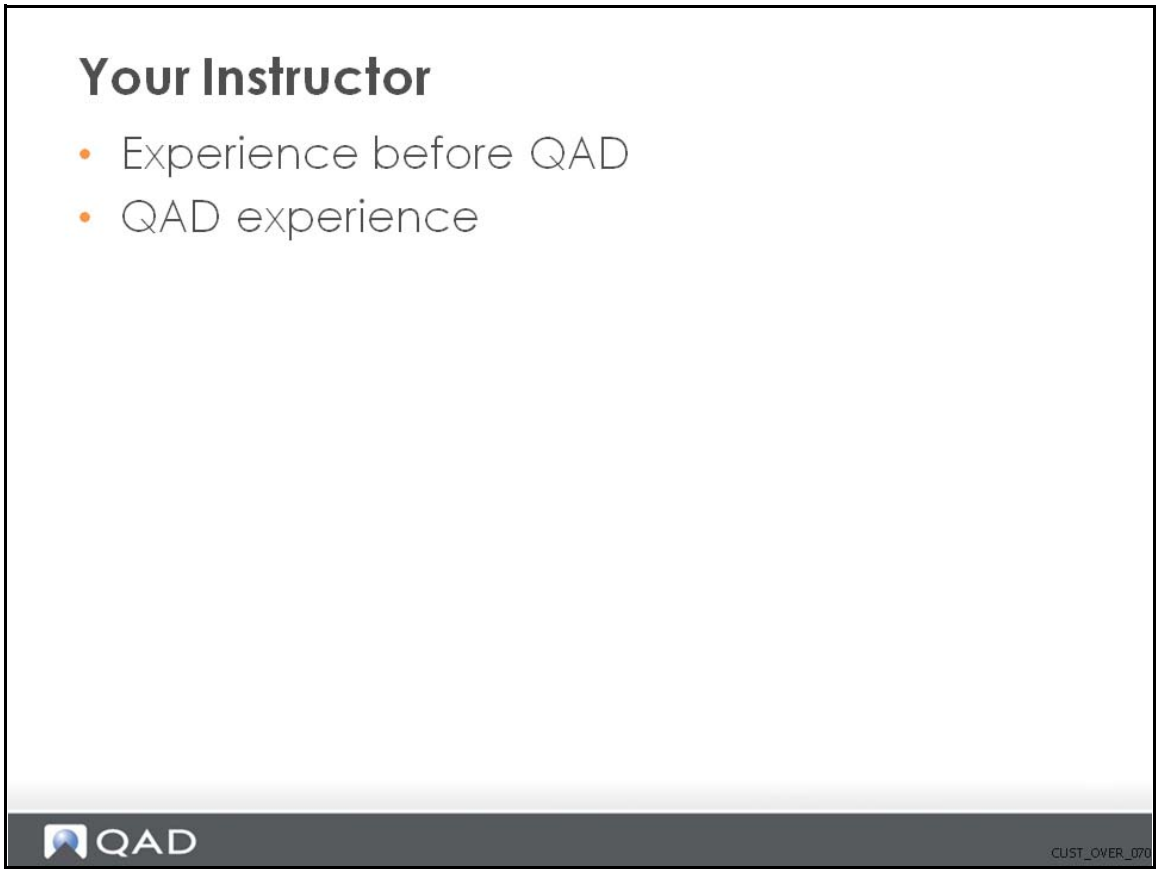

Objectives
After completion of this course, you should:
• Have an understanding of:
- Non-intrusive development
- The impact on development
• Be able to:
- Perform customizations using QAD application features and functions
- Customize component-based QAD applications
QAD
CUST_OVER_040

Daily Schedule

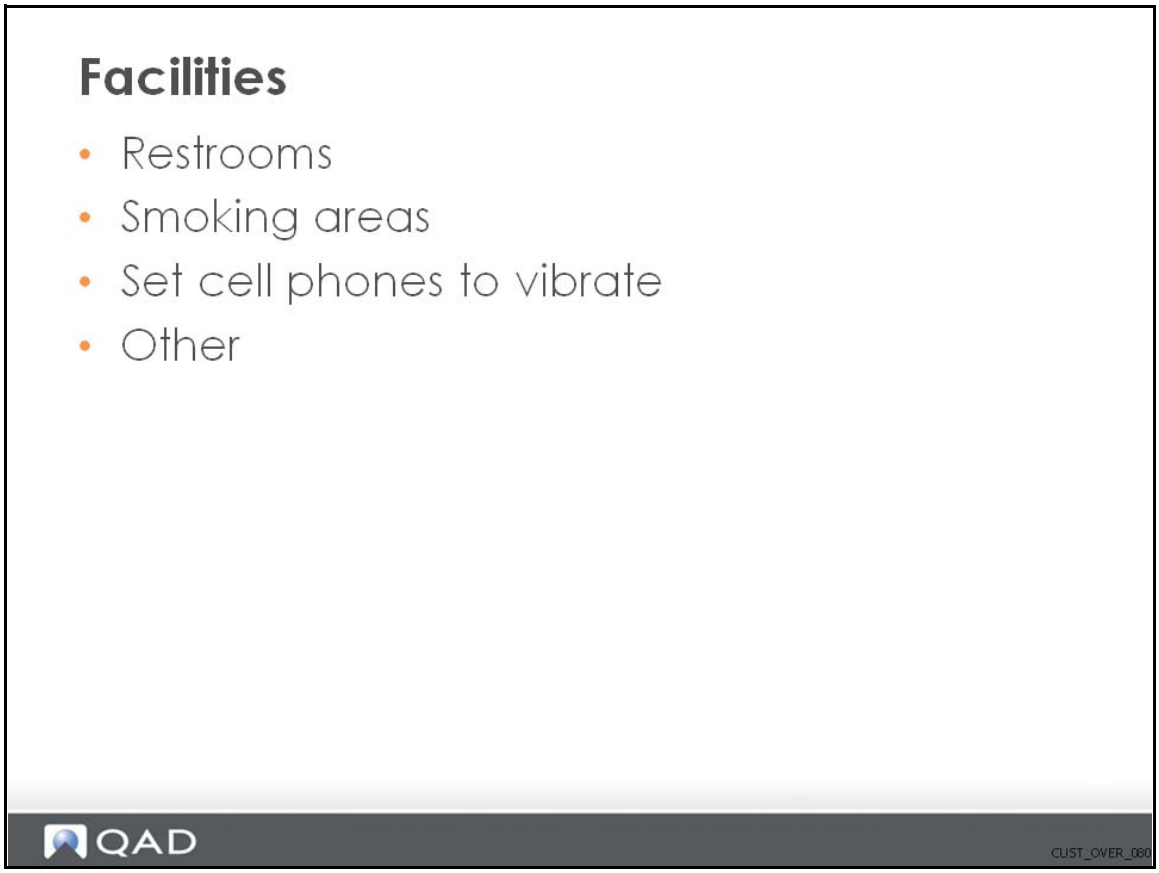
Agenda


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
QAD
CUST_OVER_060

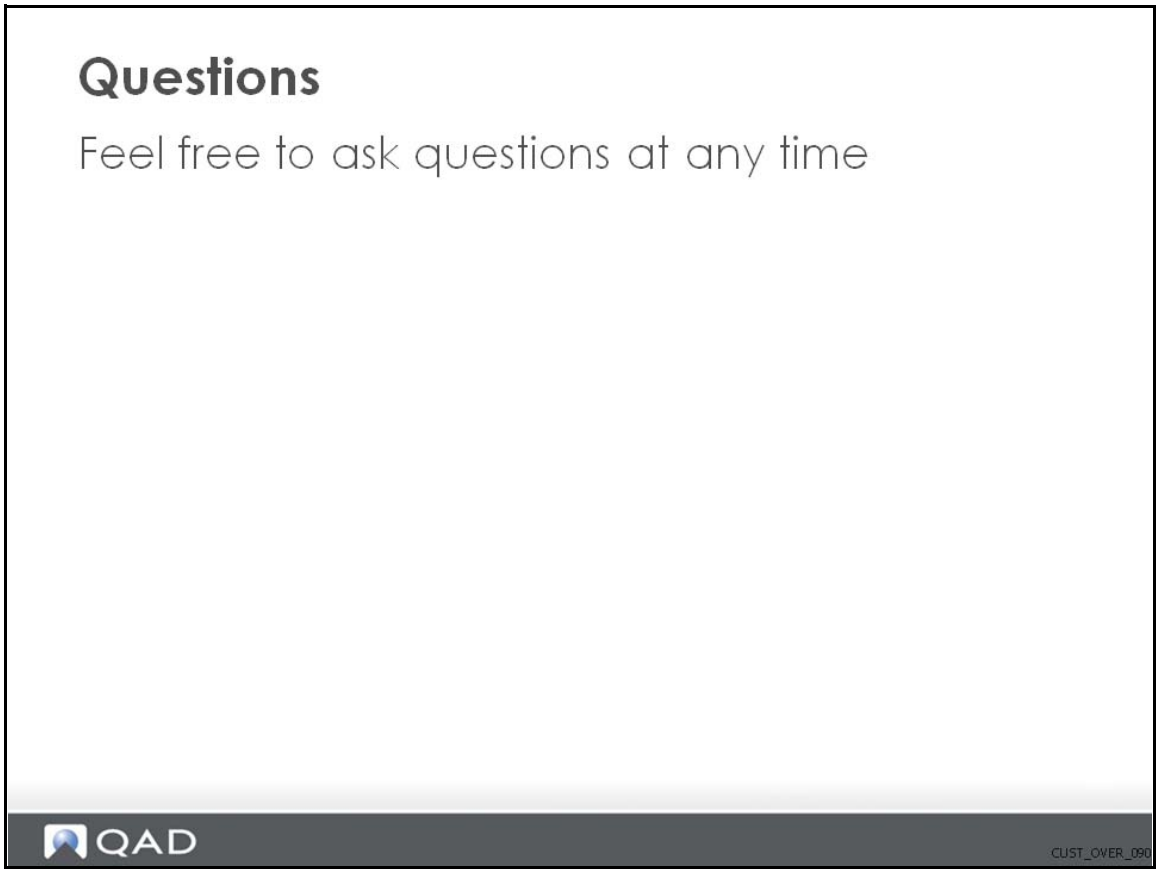
Your Instructor



Facilities


Facilities
Restrooms
Smoking areas
Set cell phones to vibrate
Other
QAD
CUST_OVER_080

Questions

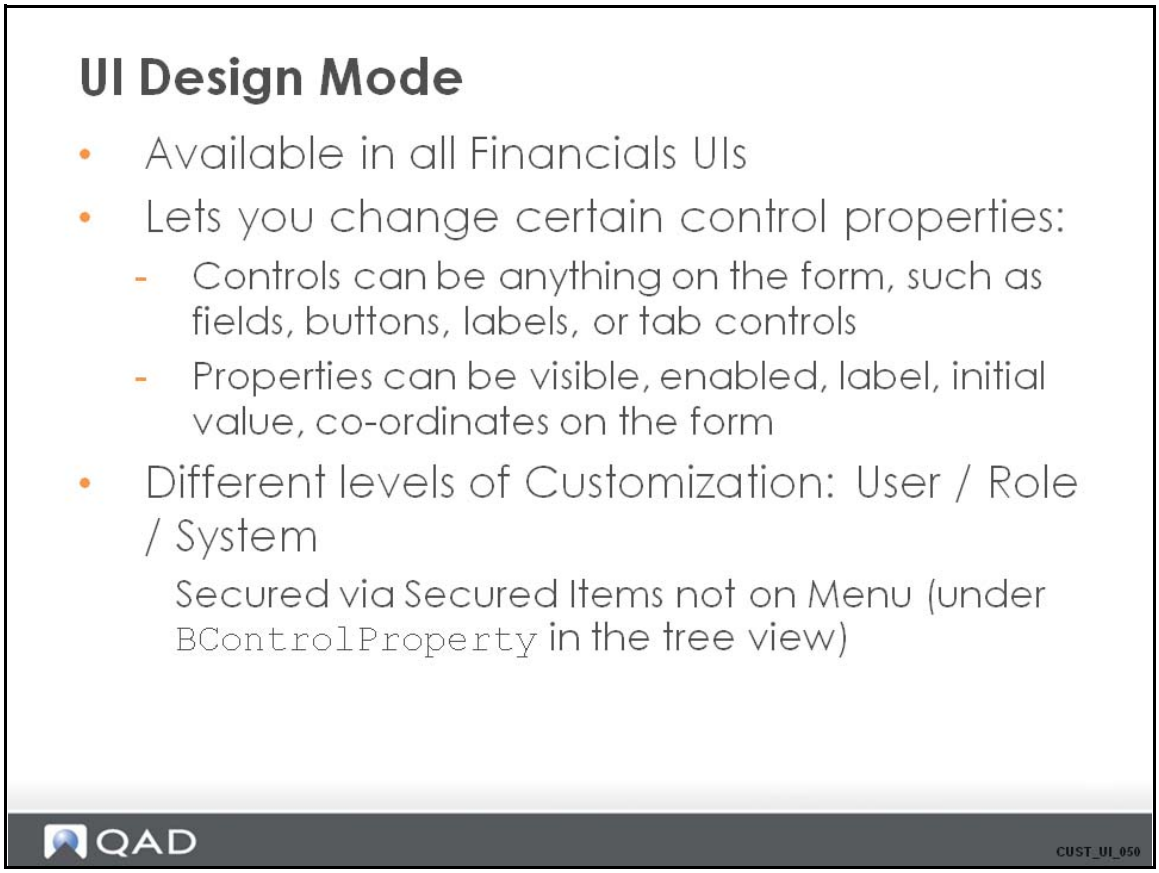

Questions
Feel free to ask questions at any time
QAD
CUST_OVER_090

Chapter 2

Non-intrusive Customization of Enterprise Financials

Non-Intrusive Customization: UI Design

UI Design Mode



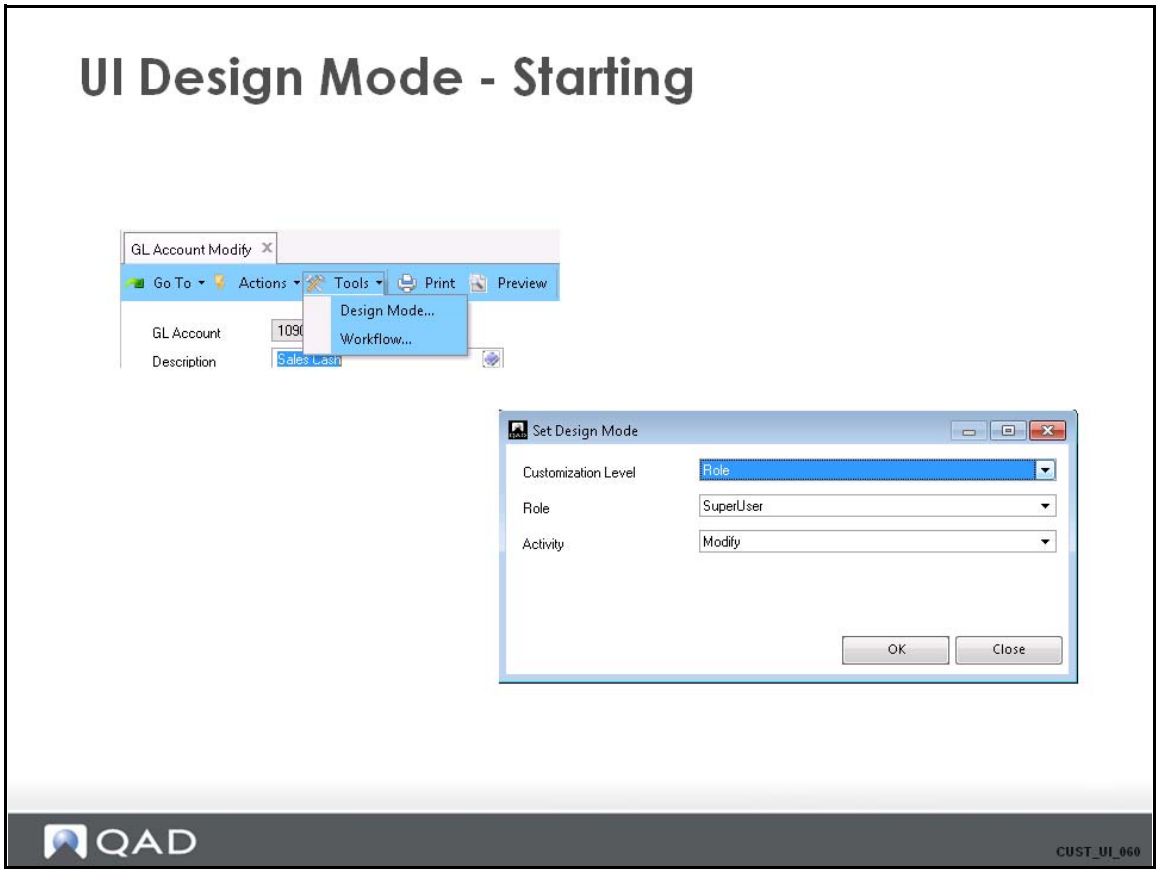
- QAD UI developers design all Financials forms. The forms are included in the application UI libraries.
- The system lets you apply changes to the standard layout/design. The system stores the applied changes (changed properties) at user/role/system level.
- UI Design Mode is a secured feature. The 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

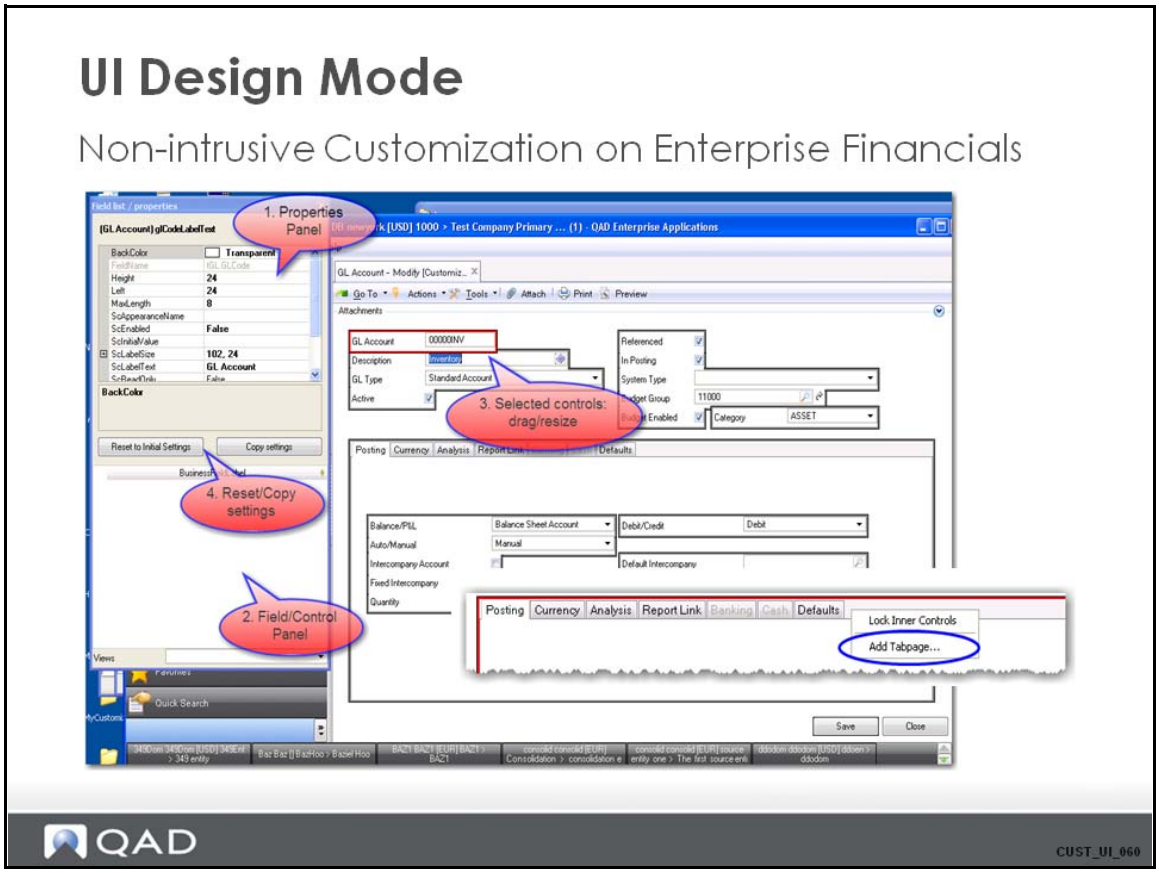


The UI Design Mode is the main way to change the applications UI. Typically, customization developers would remove fields, or make them read-only. They can also totally rearrange the form.

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.

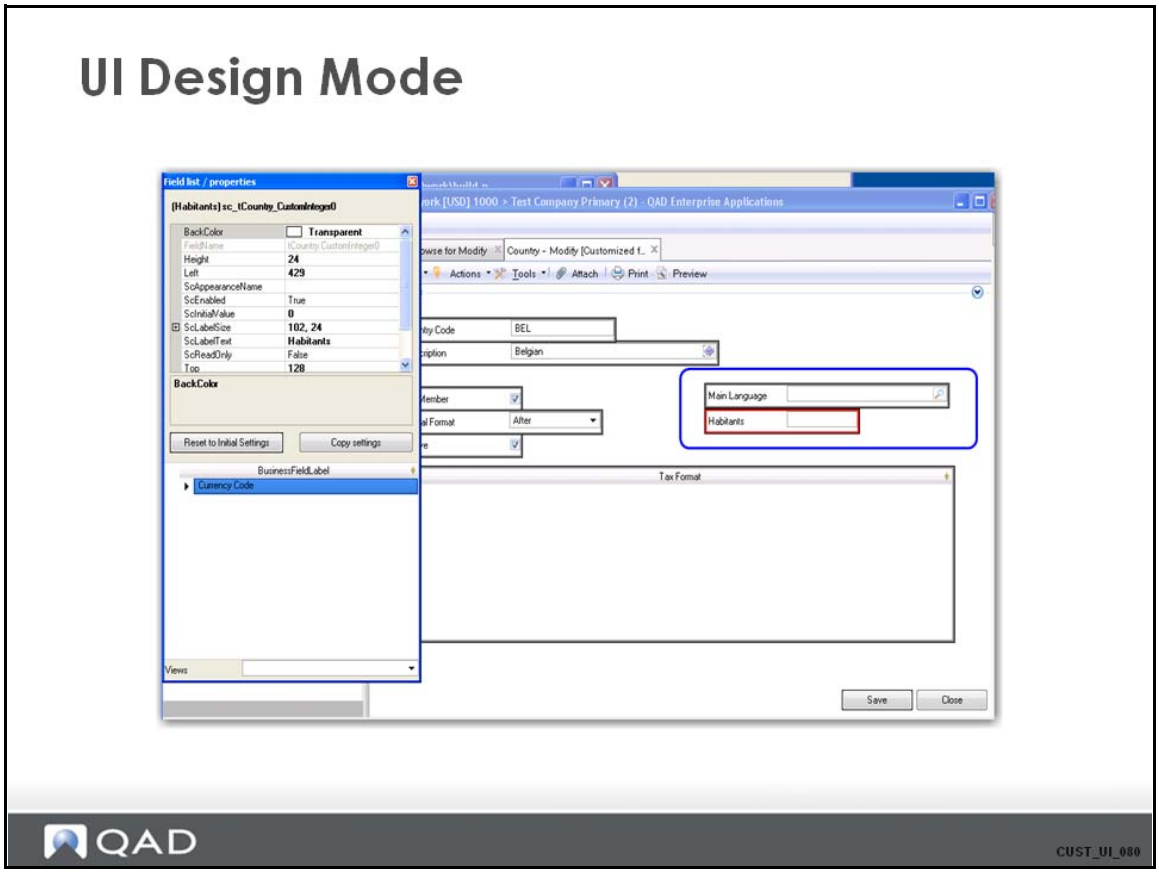
If the user wants to add a custom table on the form, the user can drag it to the form, which adds a grid to the form.

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

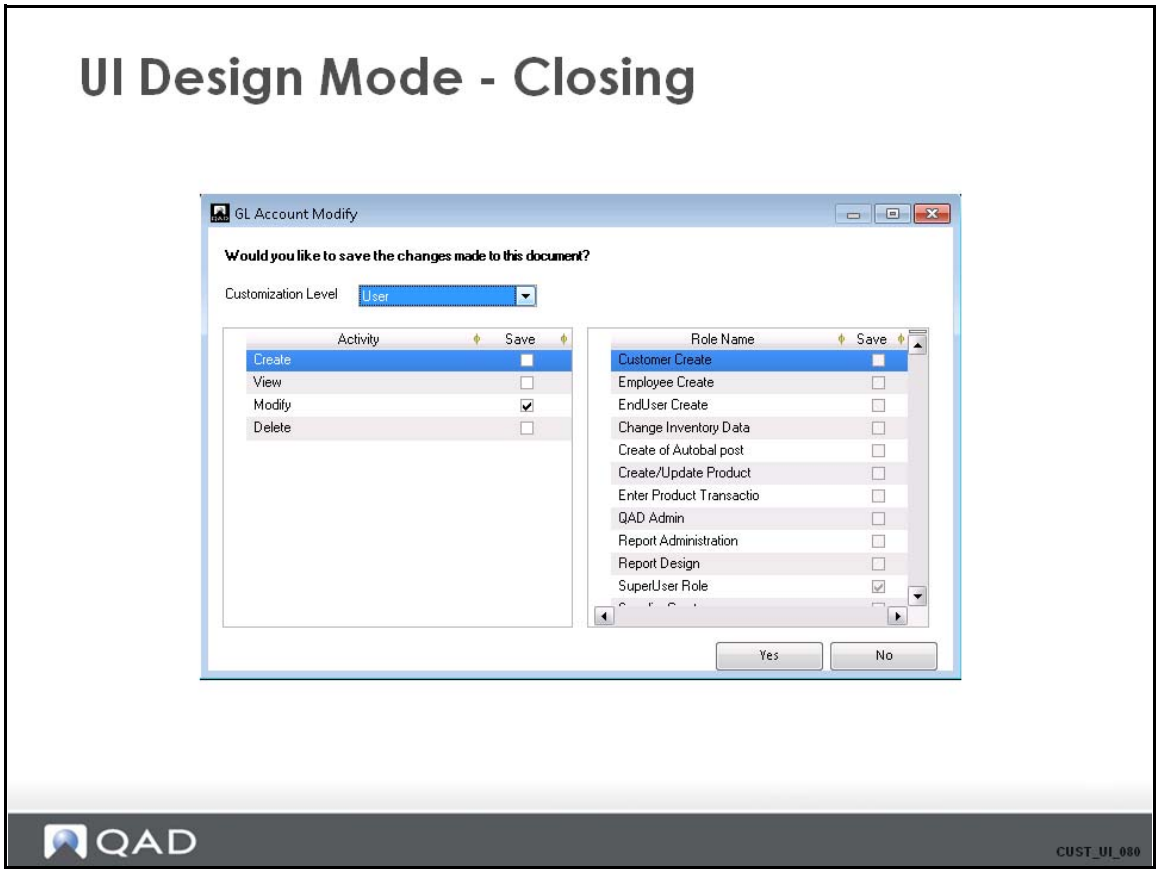
On the form design, the user can use the context menu to add a tabcontrol (max. 1 per form), and in an existing tabcontrol add tabpages. One can also add a text control this way.

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 over-writes the previous customization.

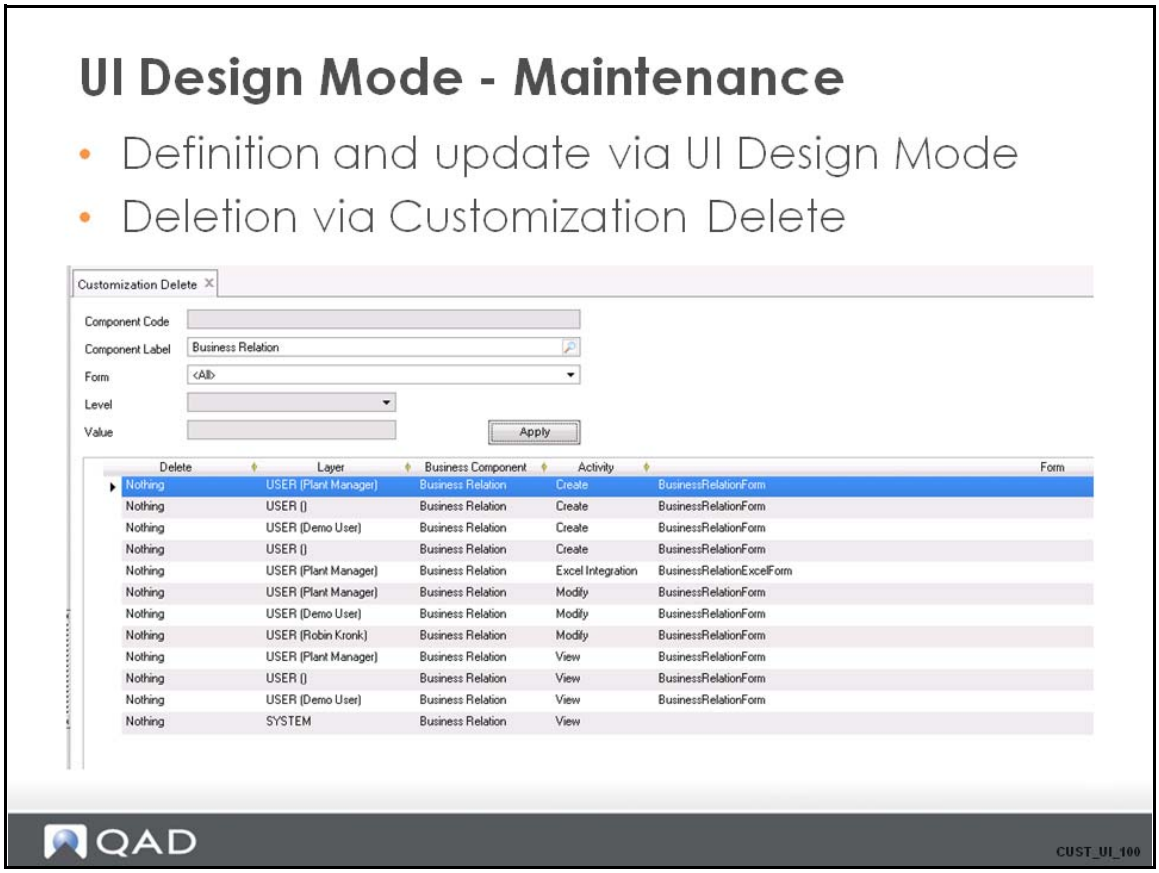
UI Design Mode



UI Design Mode



UI Design Mode

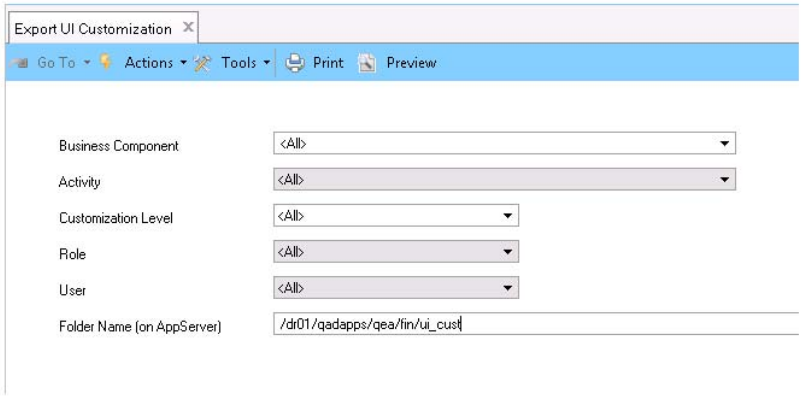


Importing and Exporting UI

Exporting UI

Exporting UI

- Export UI customization option



- API

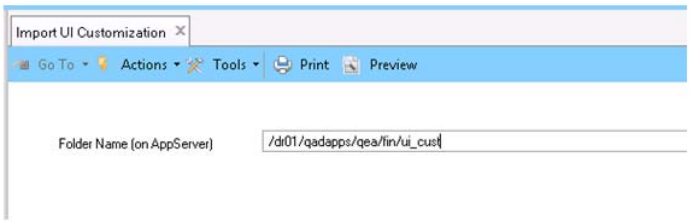
```
ApiExportCustomization(...)
```

QAD

Importing UI

Importing UI

- Import UI customization option

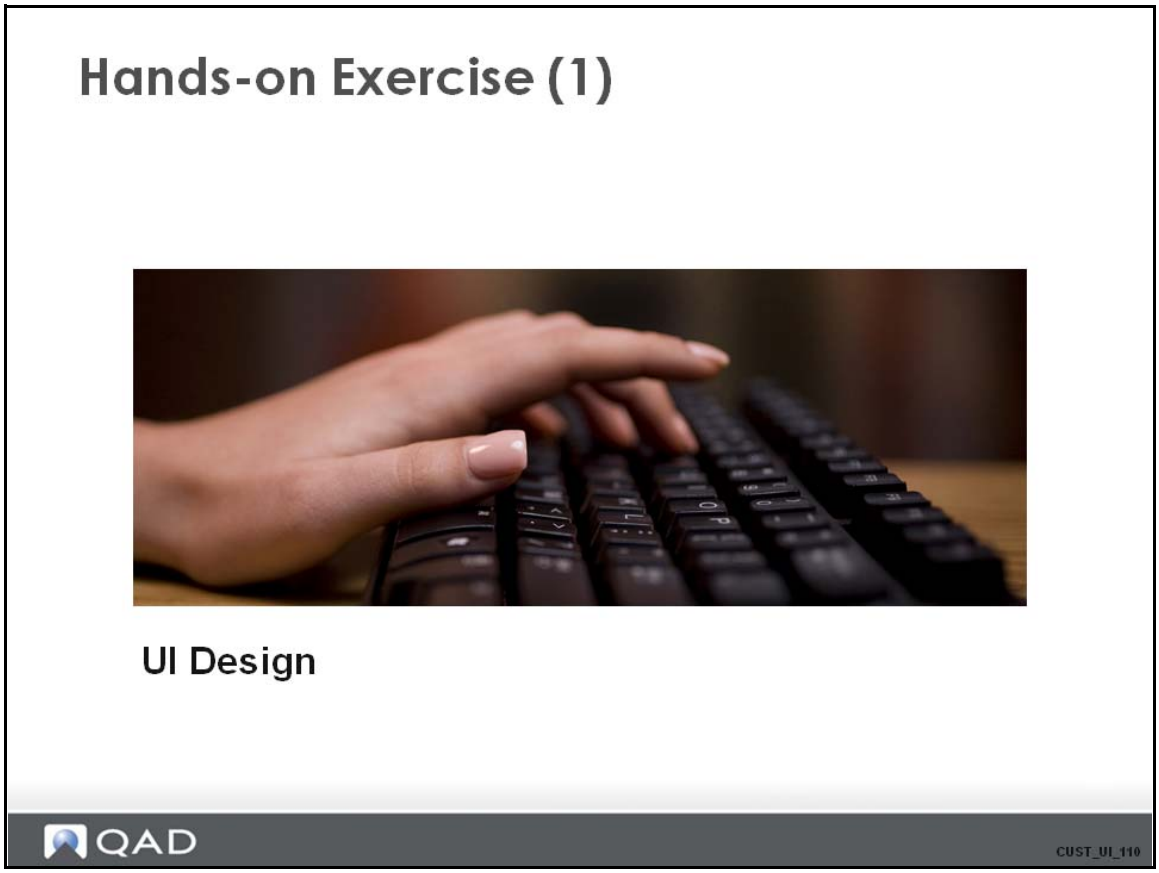


- API
  `ApiImportCustomizations(icImportFile)`

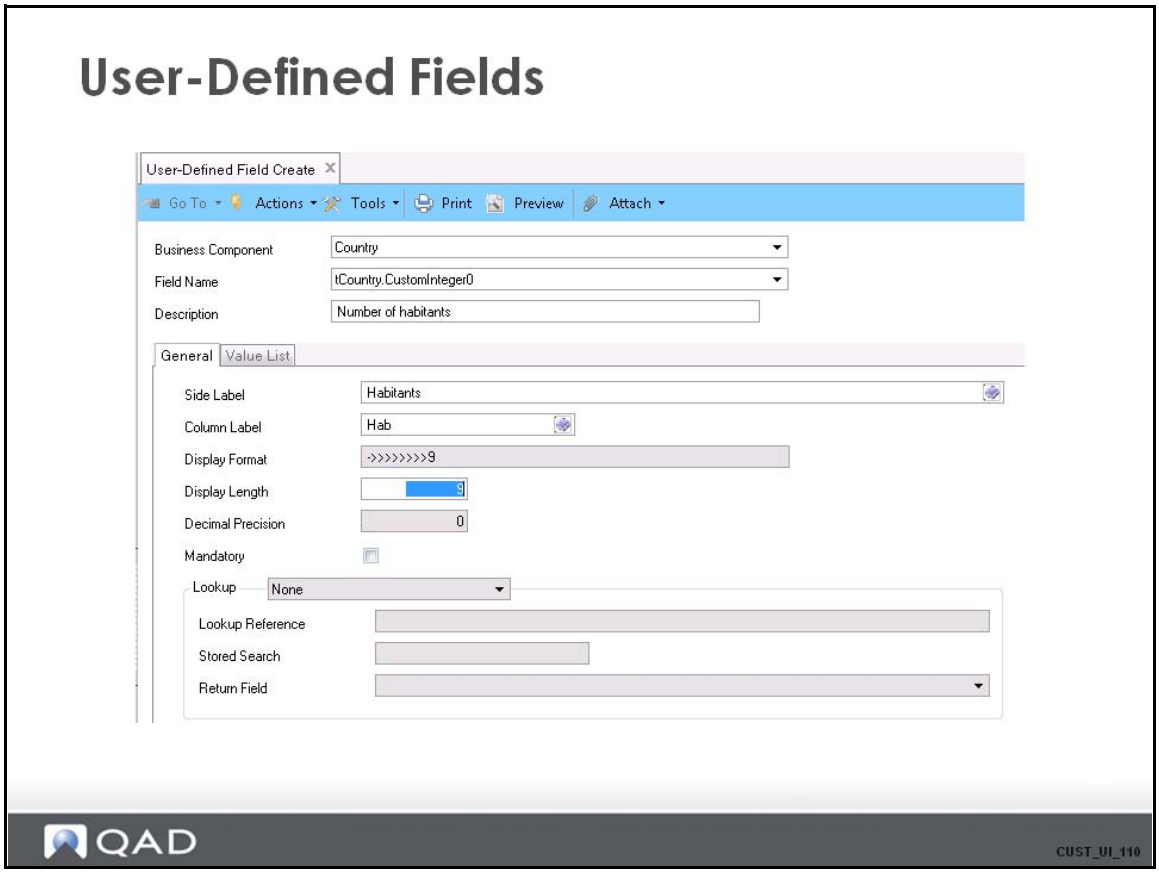
QAD

CUST_INT_120

Hands-on Exercise (1)



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.

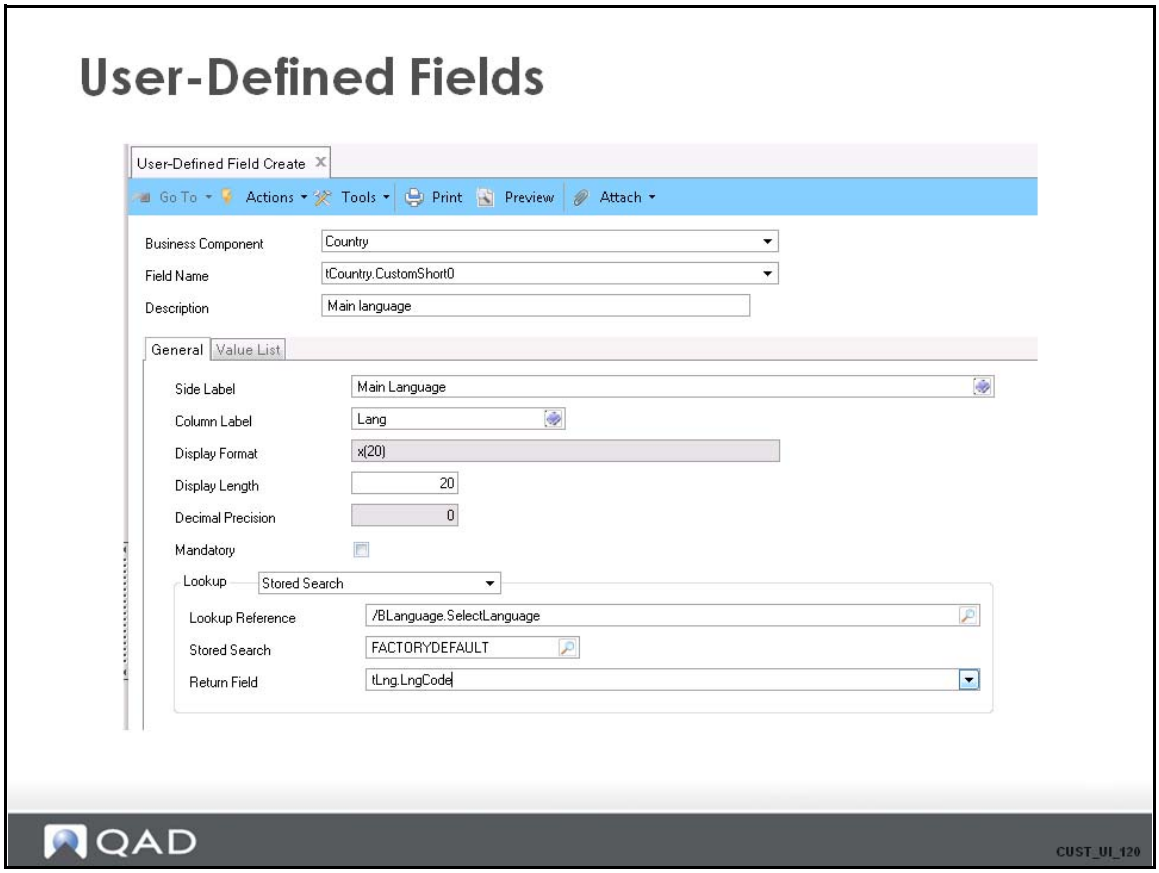
Specify the following information:

- Business Component: This is the label of the business component. For example "GL account" maps to the physical component name "BGl". 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);

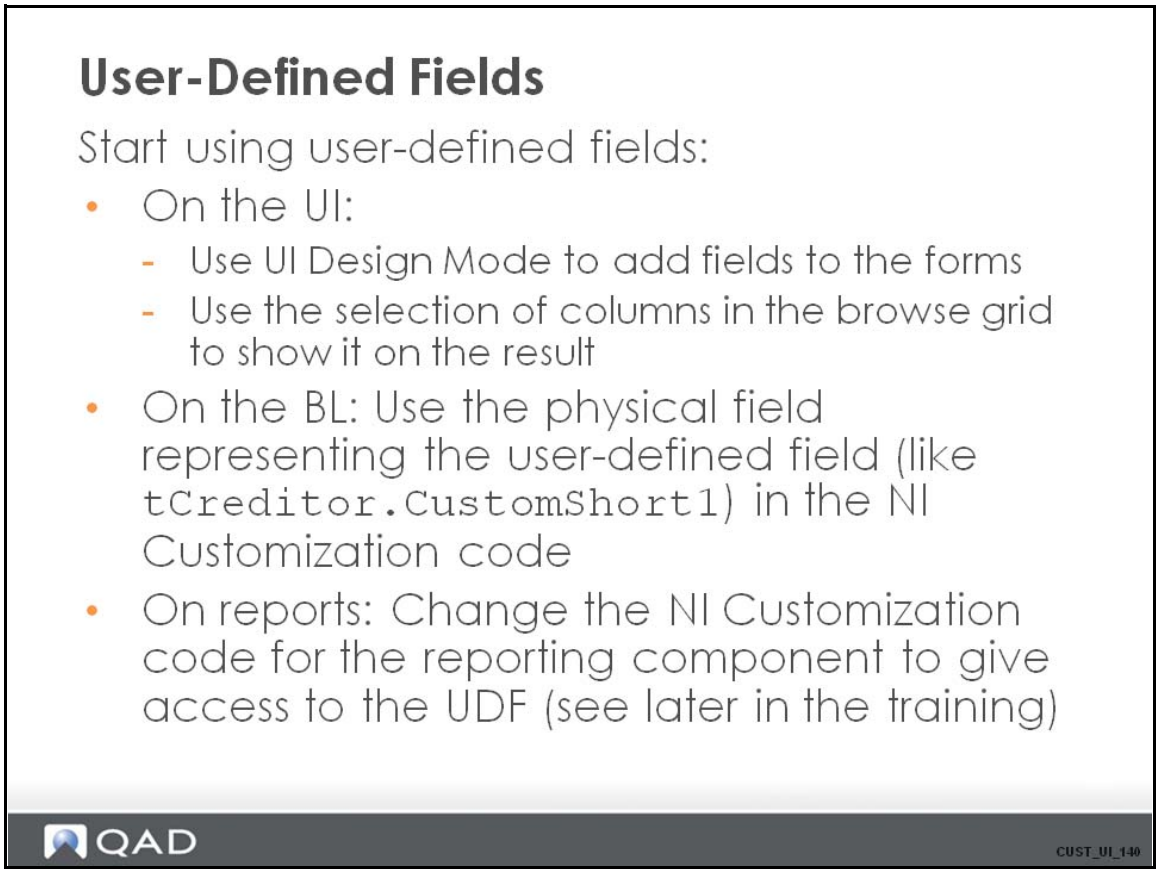
CustomDecimal0..4 (decimal type fields); CustomInteger0..4 (Integer type fields); CustomDate0..4 (date type fields); CustomNote (character type field especially for fields represented as an editor)

- 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.

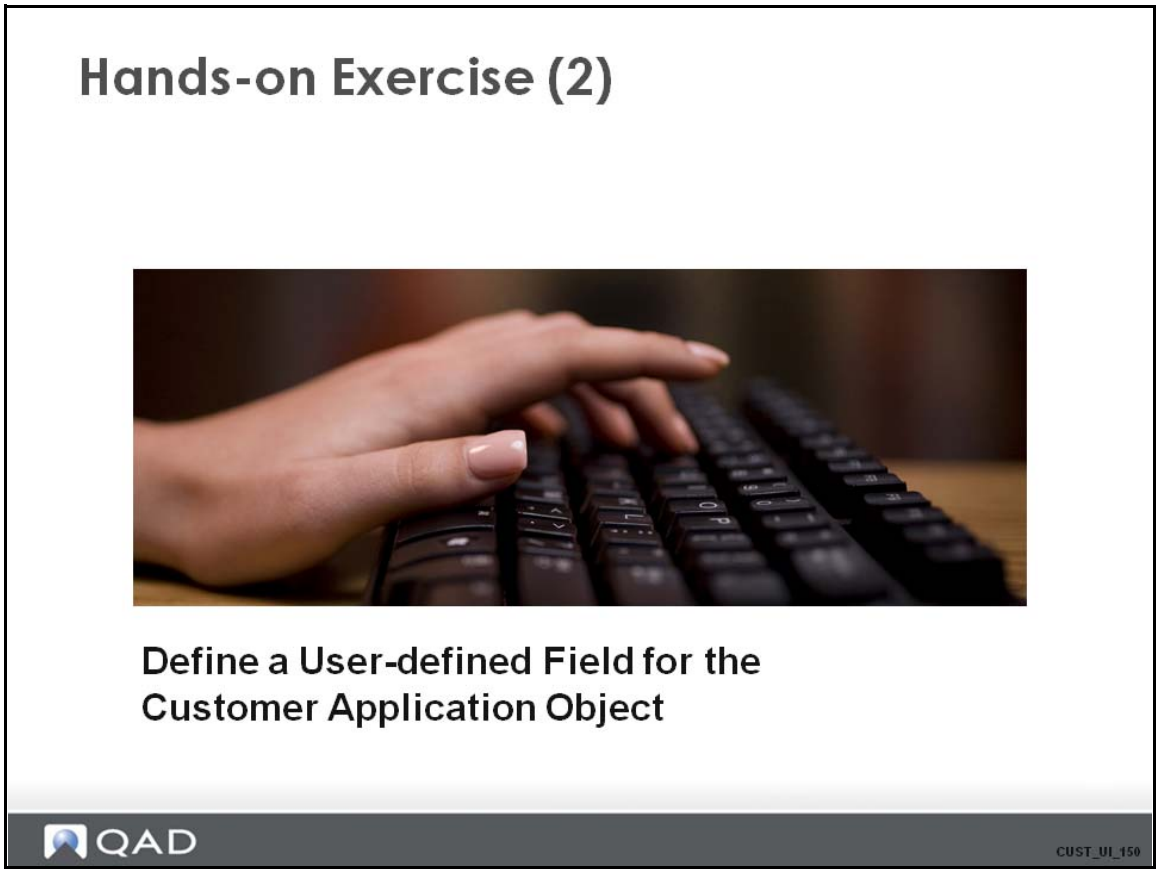
User-Defined Fields



User-Defined Fields

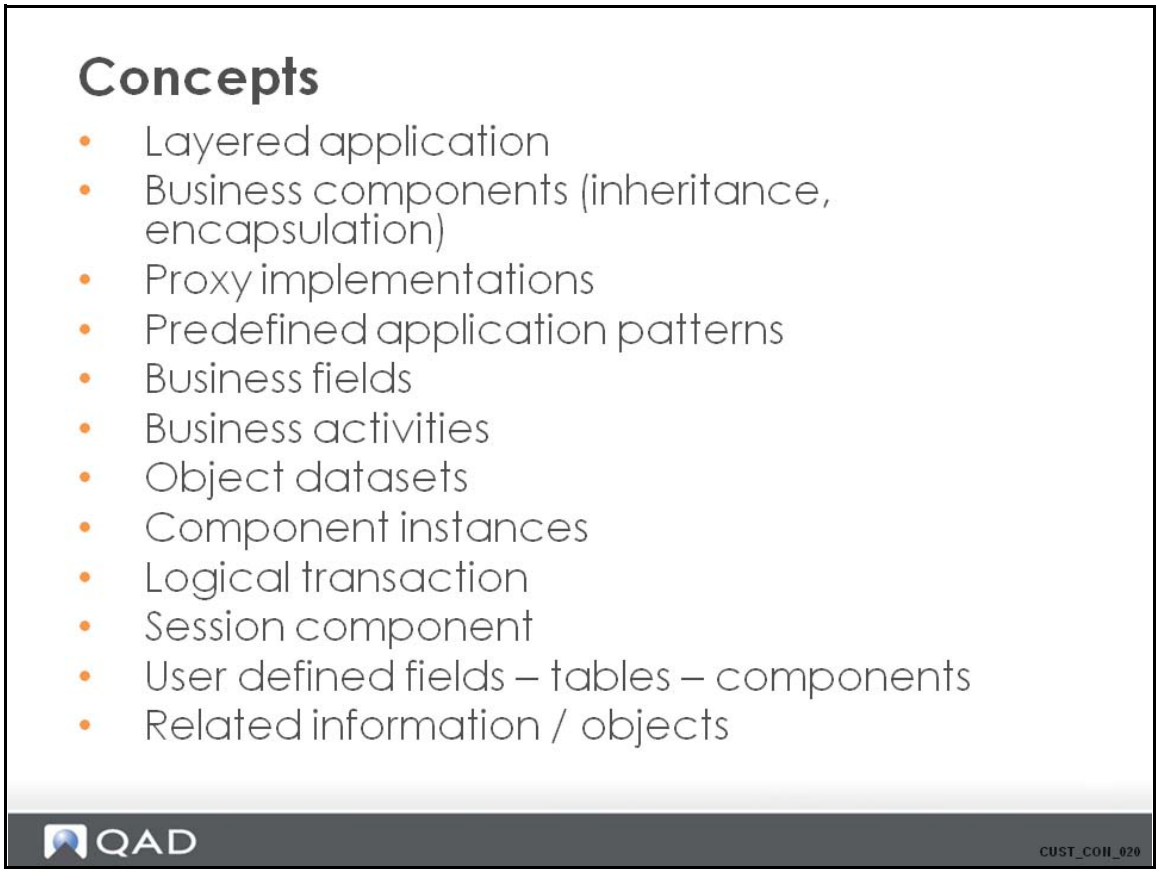


Hands-on Exercise (2)



Non-intrusive Customization: Concepts

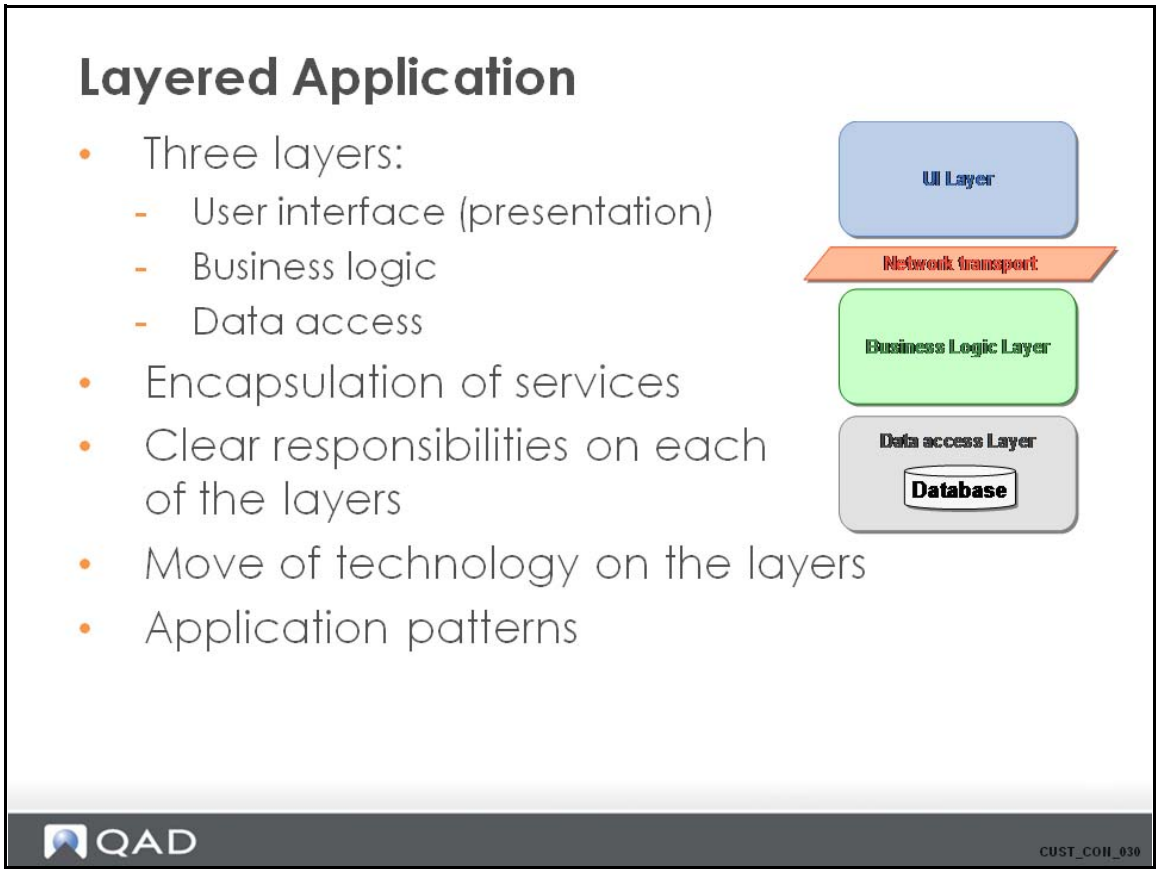
Concepts



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.

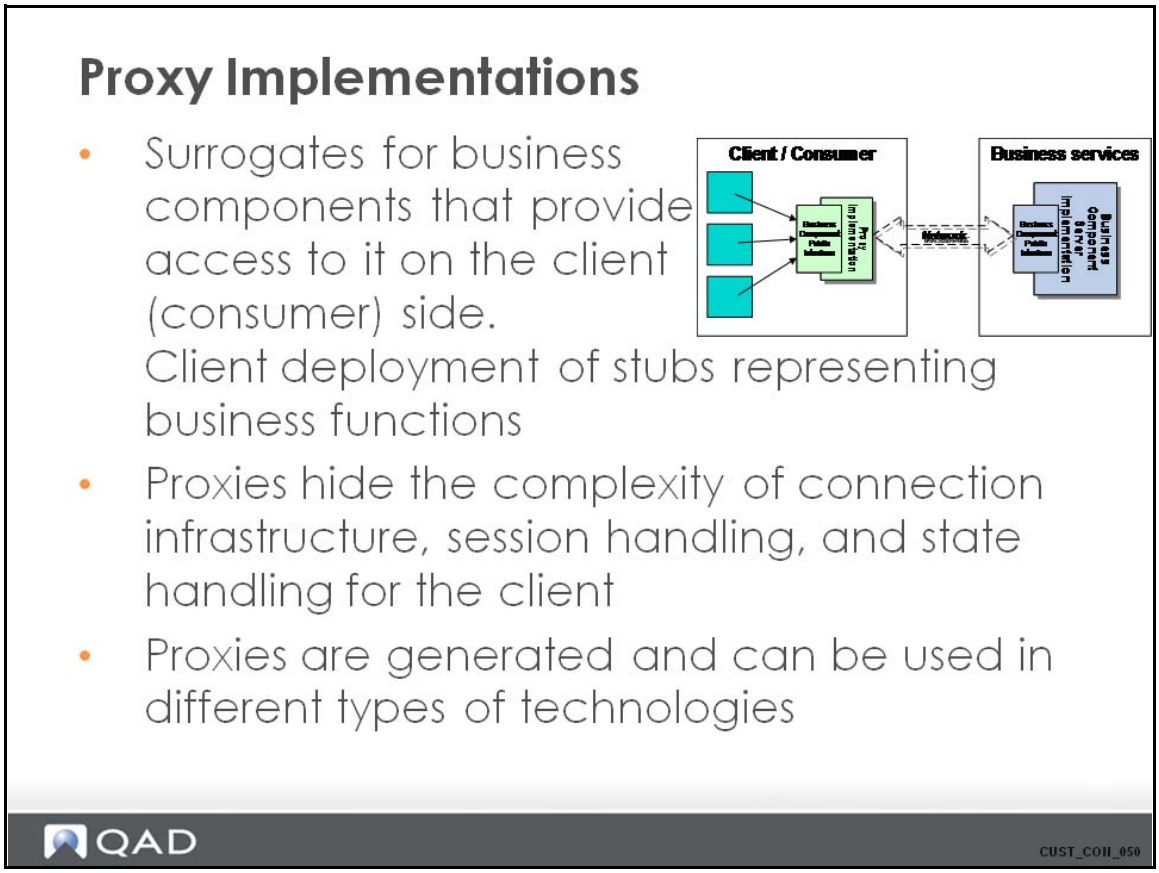
Business Components

Business components 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

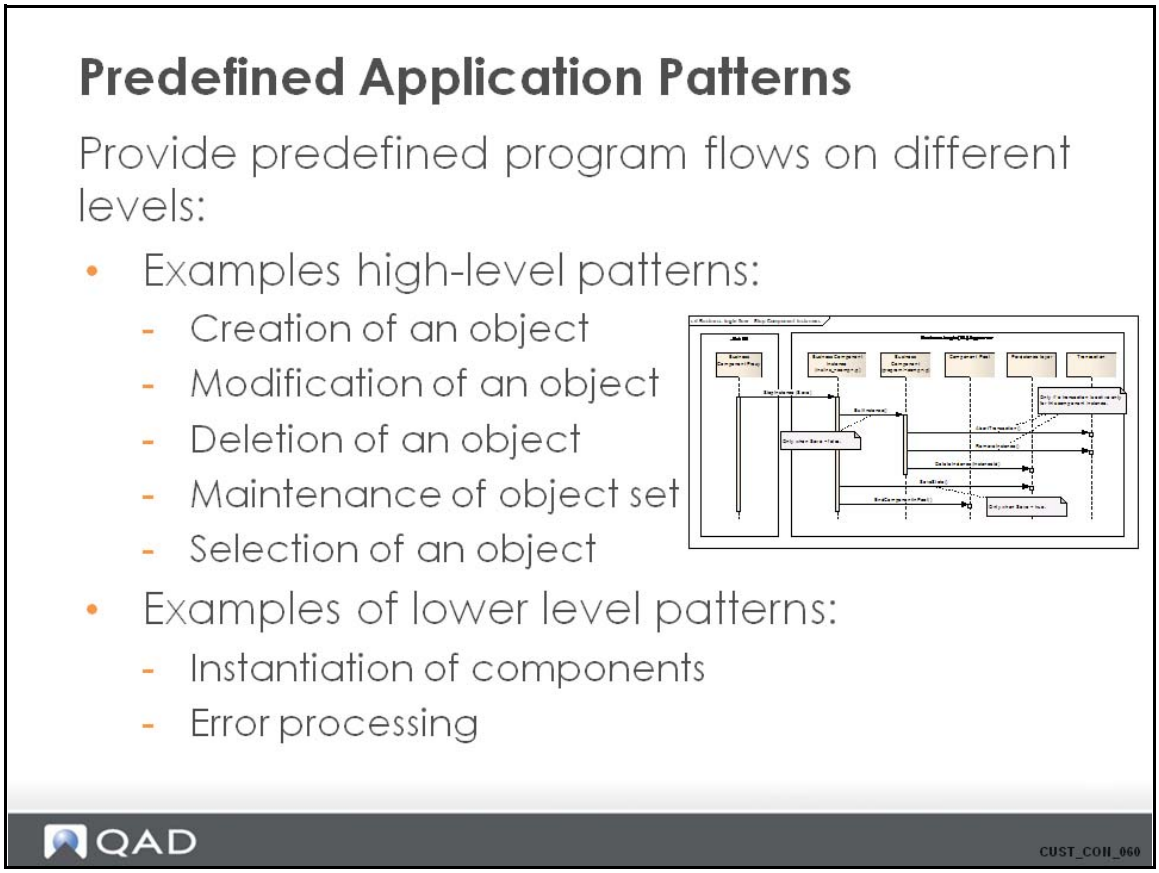


All business services are available on a central host in the system. These services are available through from a client through the network layer. In order to use the central business service, it is typically necessary to know:

- 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

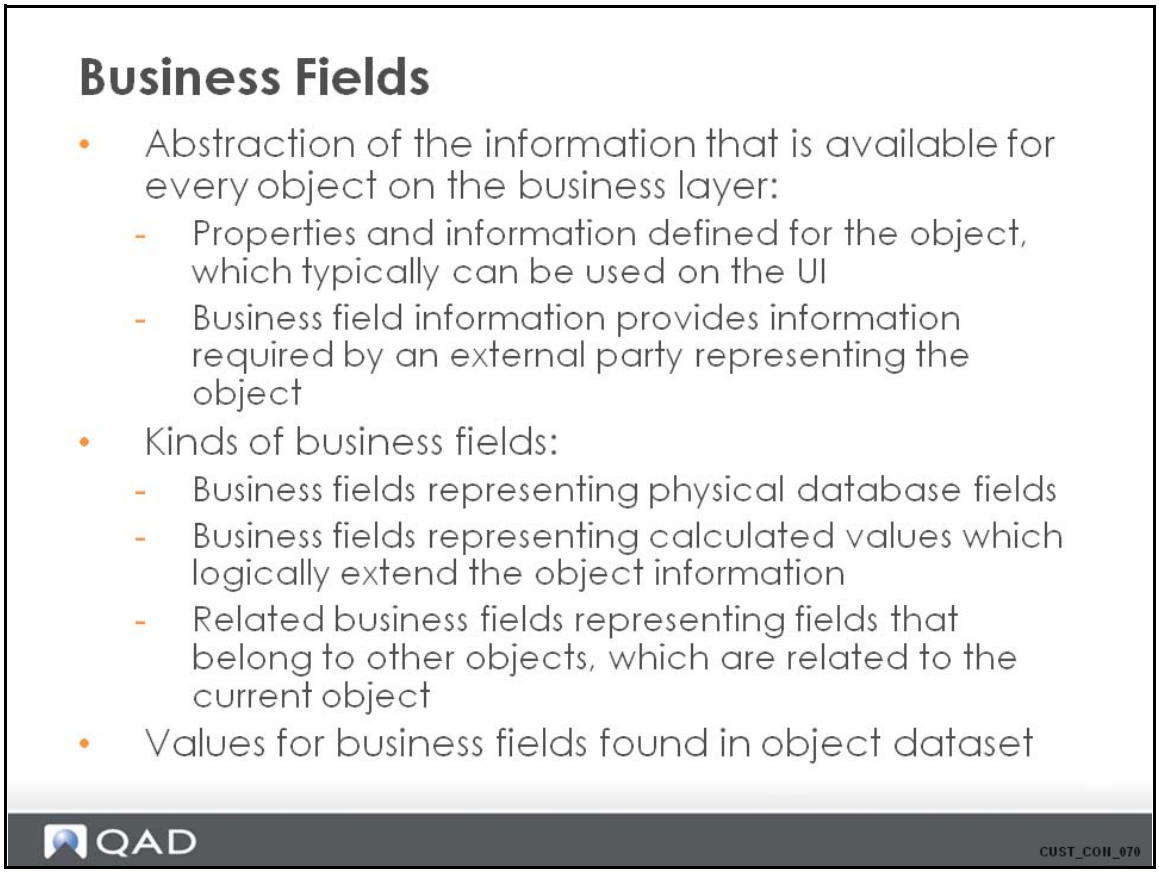


Patterns are crucial to new types of layered, component-based applications. The patterns drive the requirements for the generic coding in the infrastructure and give the guidelines for the design of the specific application components or classes.

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.

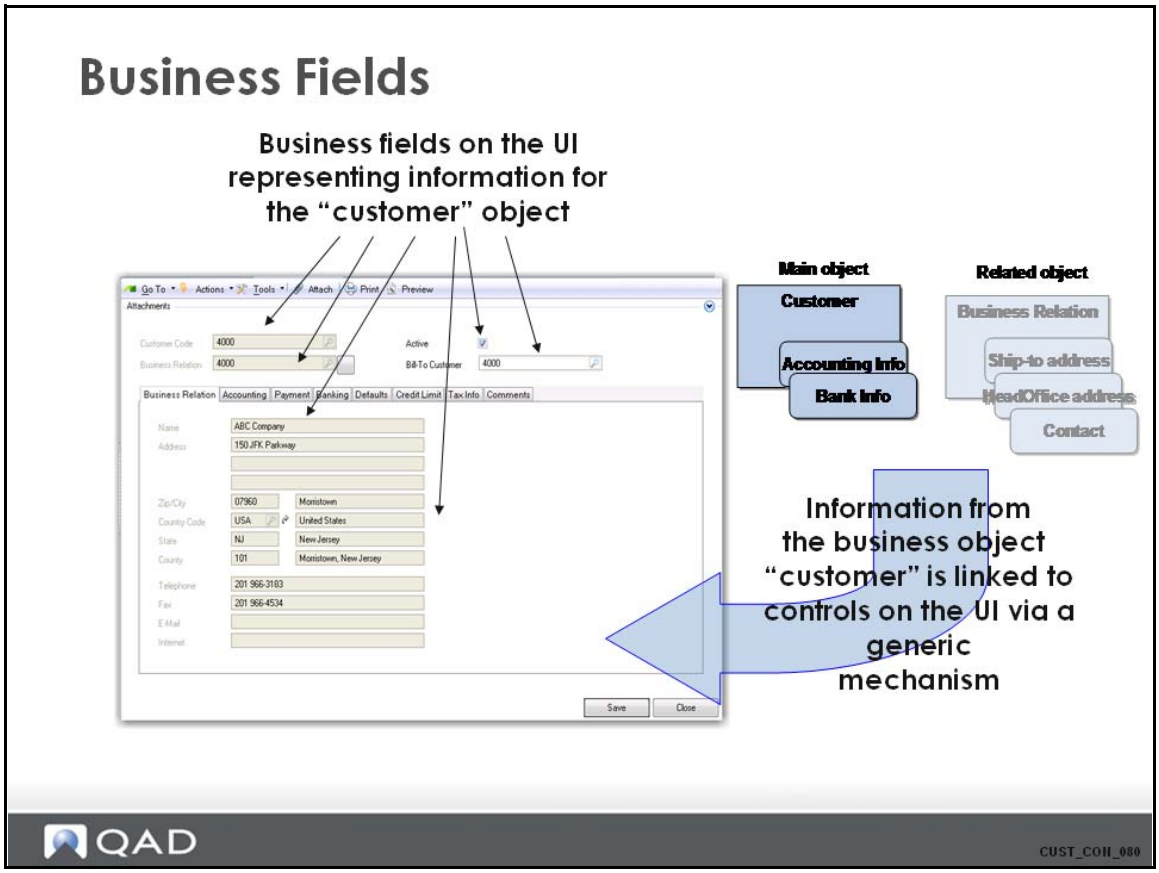
Business Fields



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

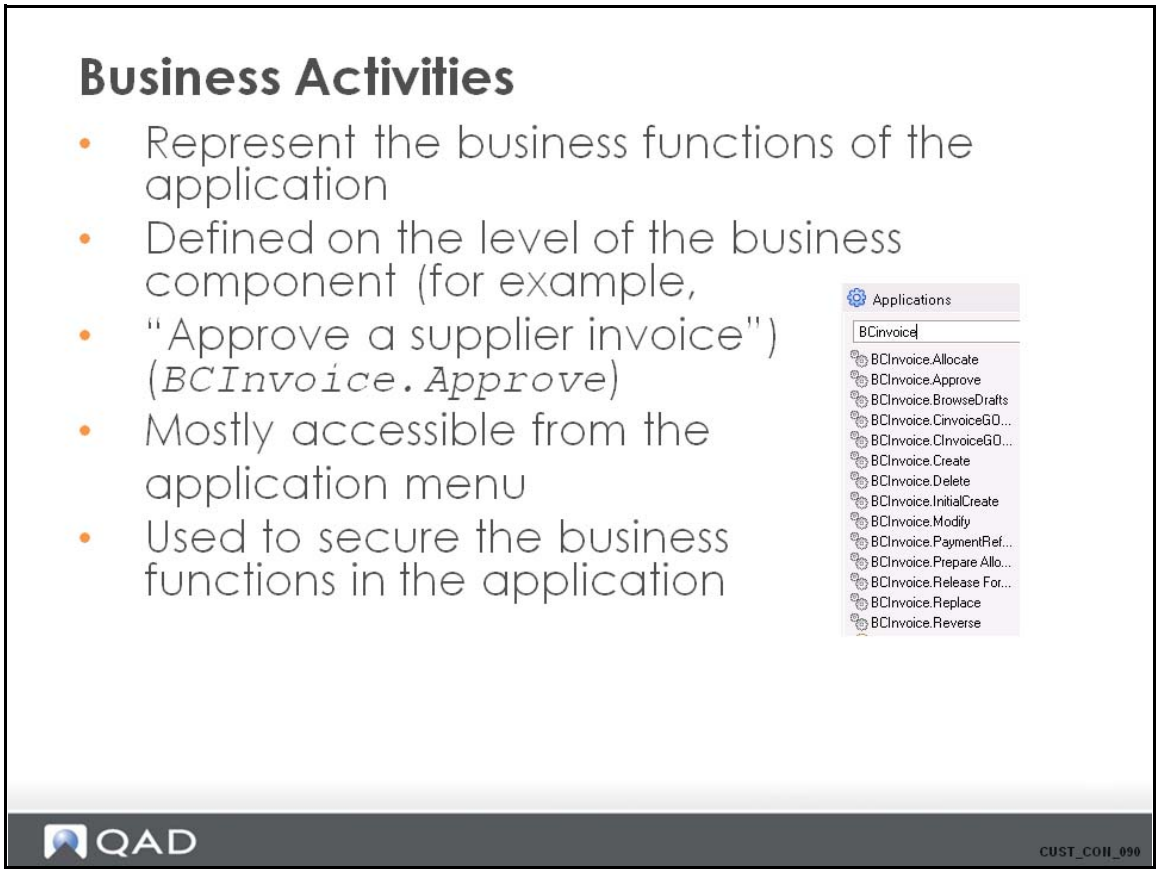
Business Fields



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

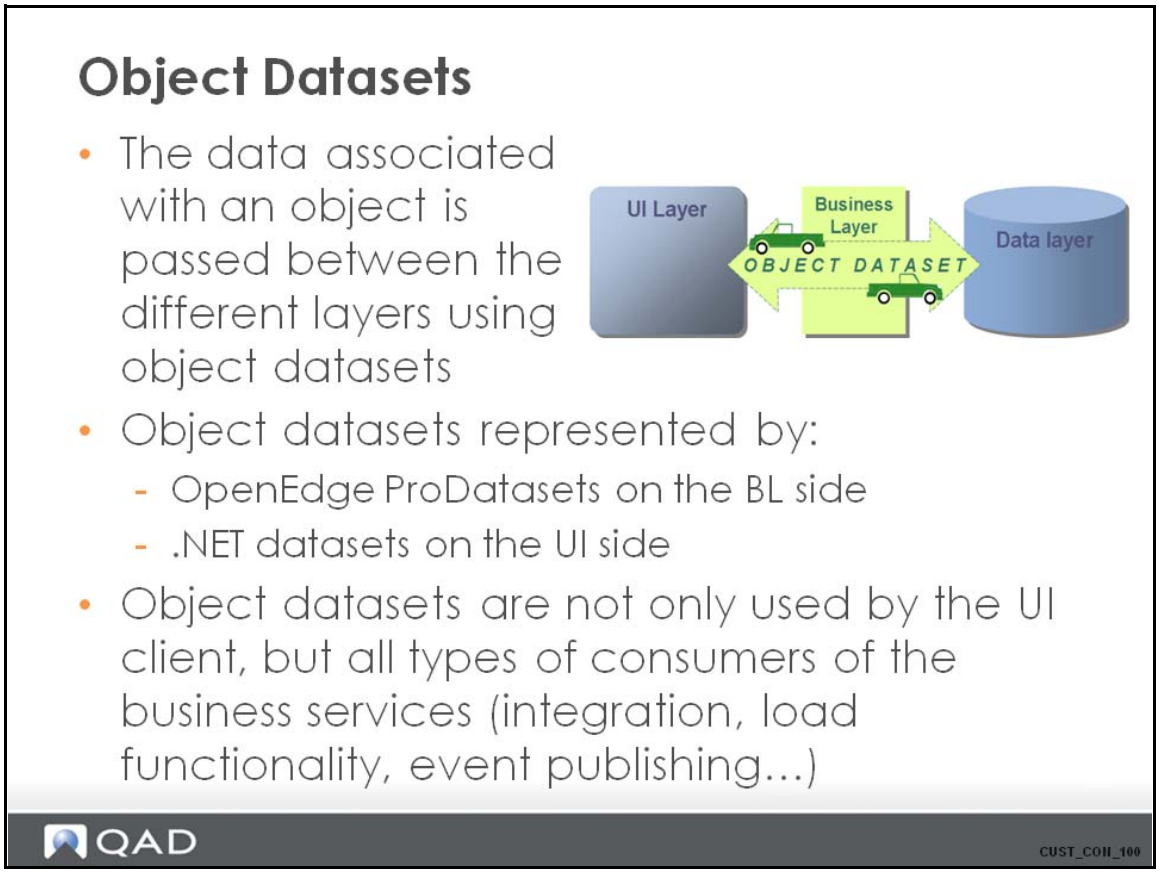


The business activities are defined on a higher level than the methods and the queries on components. Typically, the activities are exposed directly to end users. This is not the case for normal methods (even if they are public).

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

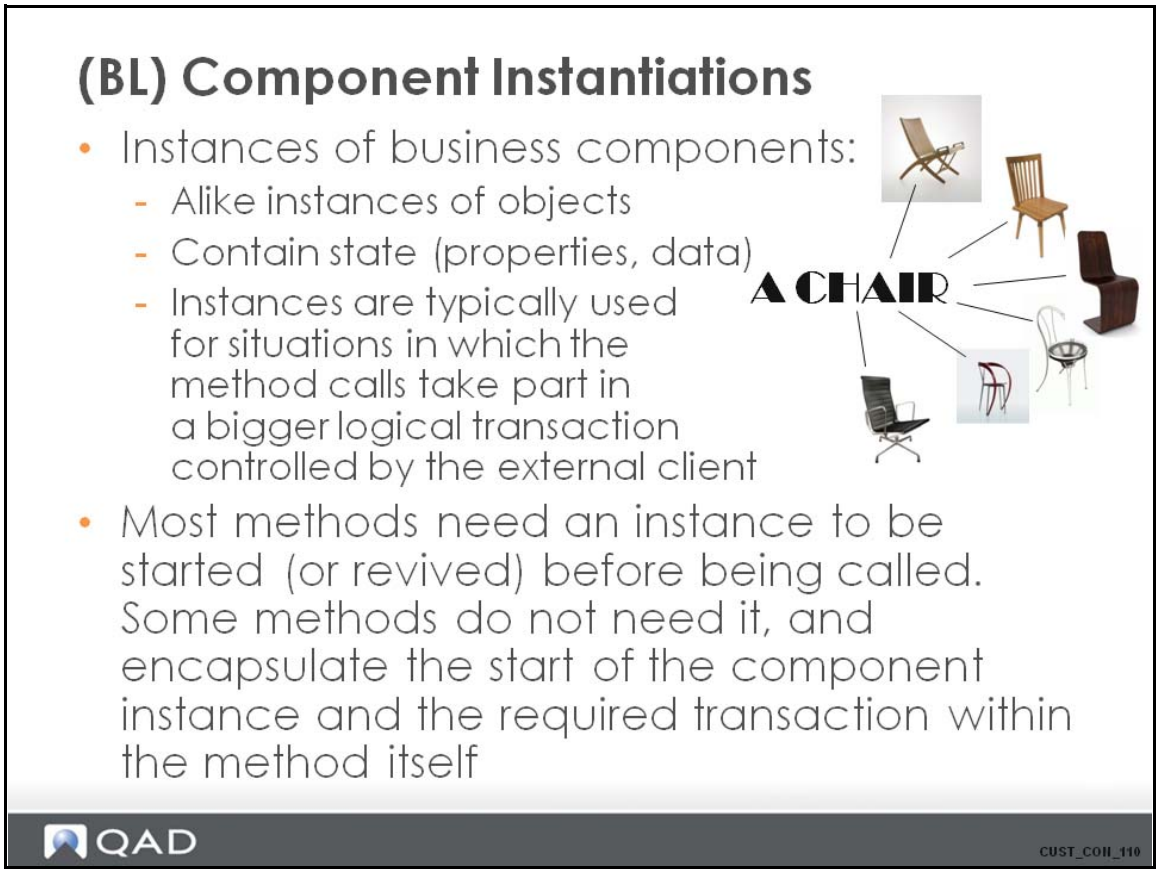


It is important to have a uniform way to pass object data between layers. As soon a business component is instantiated, and an object from the database is “loaded”, all relevant information is stored in the object dataset. The object dataset is the only way of passing the object state / information between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



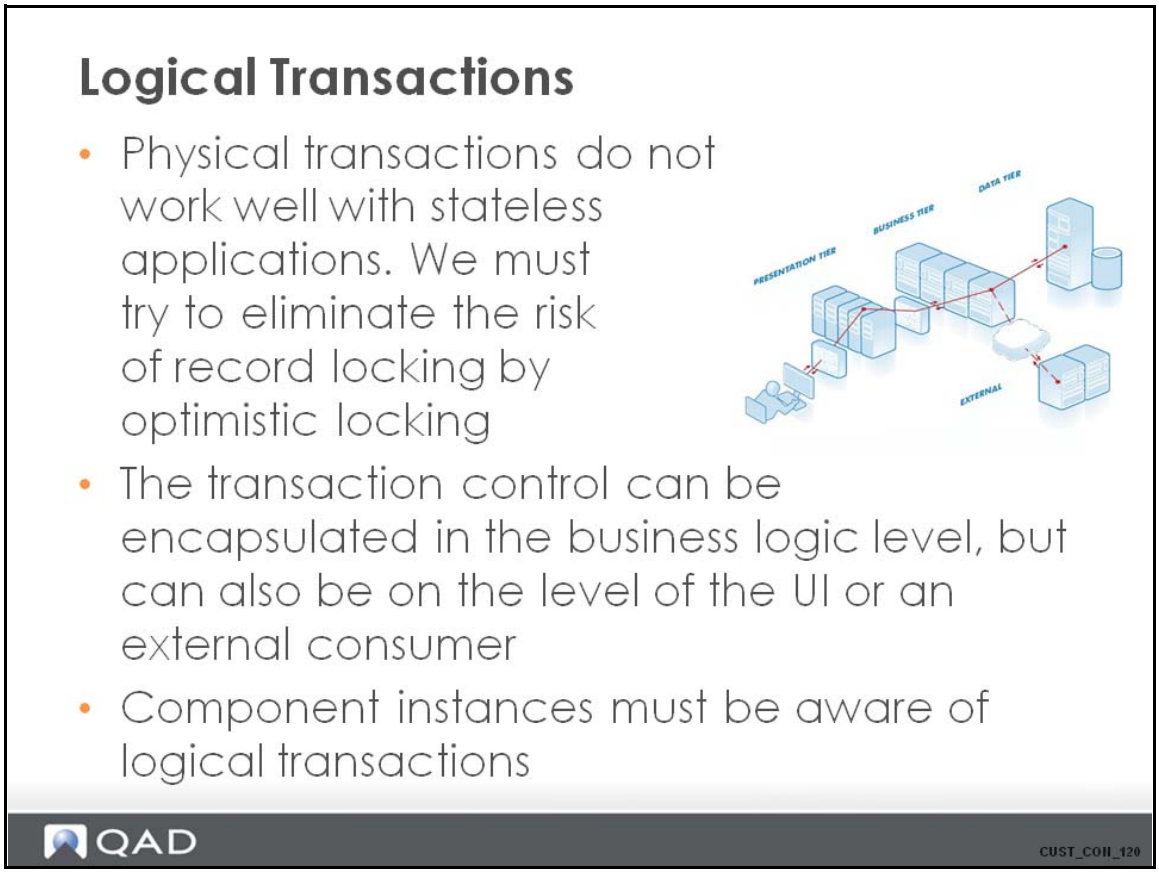
In a component-based application, the principle of “instantiation” must be very clear. Each business service consumer must instantiate a business component before being able to call a method on the business component. The consumer can be internal in the business layer, or it can be an external client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



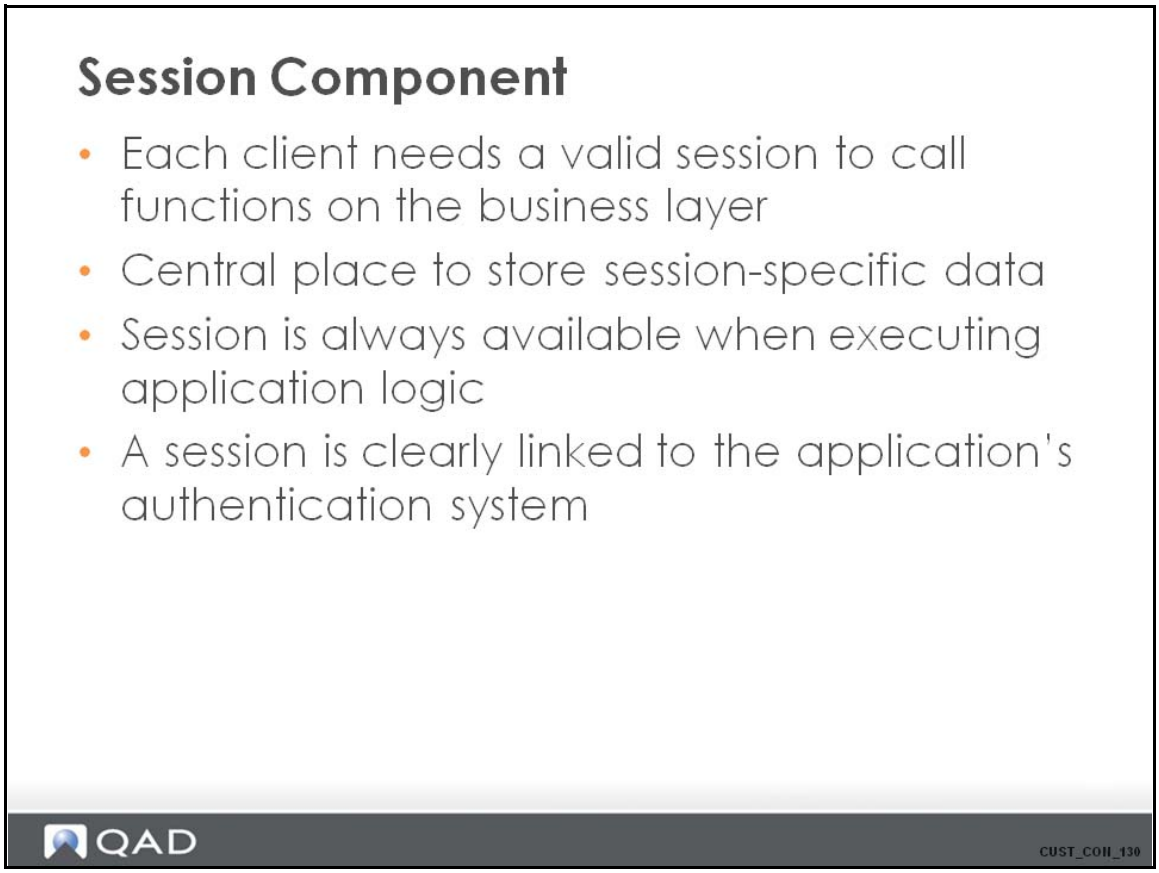
Physical transactions are de facto linked to one process, and require a stateful application model to guarantee that the same process executes the logic within a function. If you work with a multi-layer model, in which the business logic is used in a stateless way, the logical transaction is used to 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

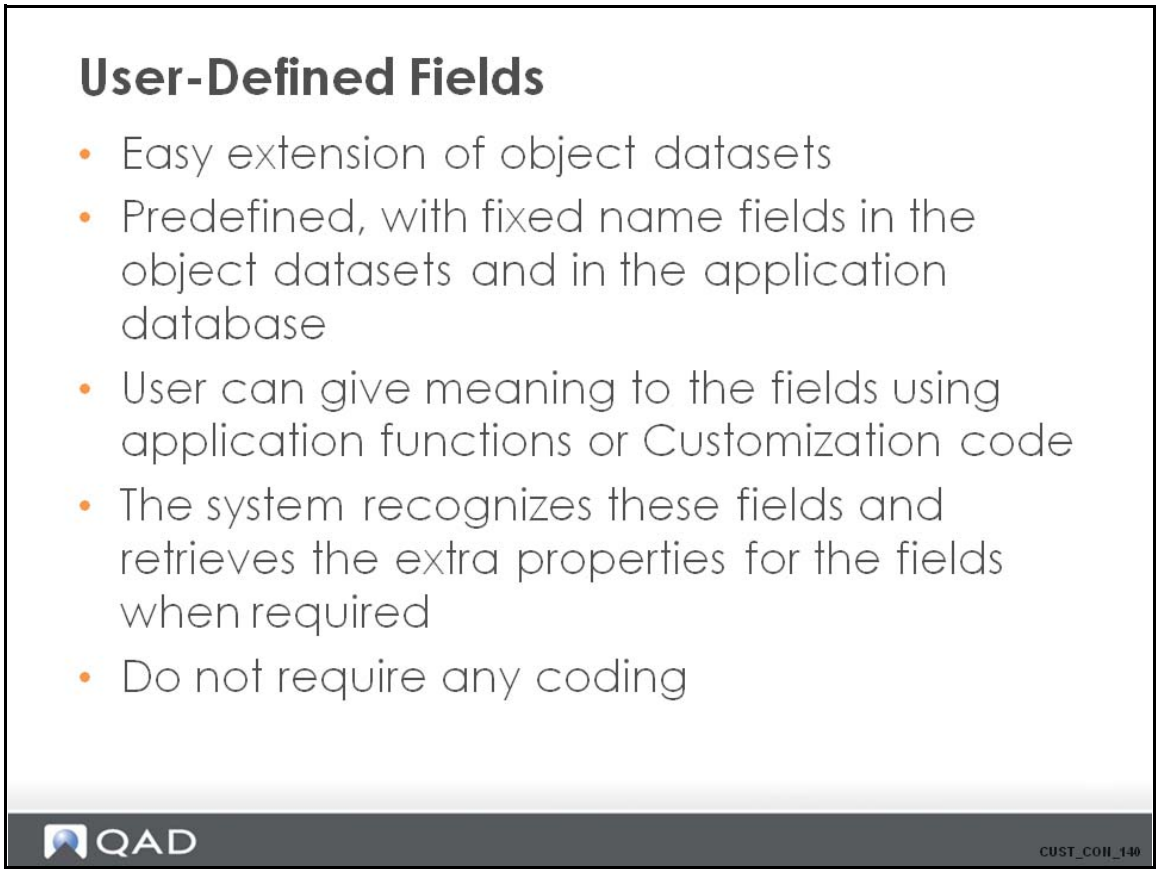


A session component is instantiated at the moment a user logs on to the system. The instance stays available throughout the user session until the user logs off or until a time-out occurs. As long as the session stays active, some data (specific for the user or specific to the context in which the user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

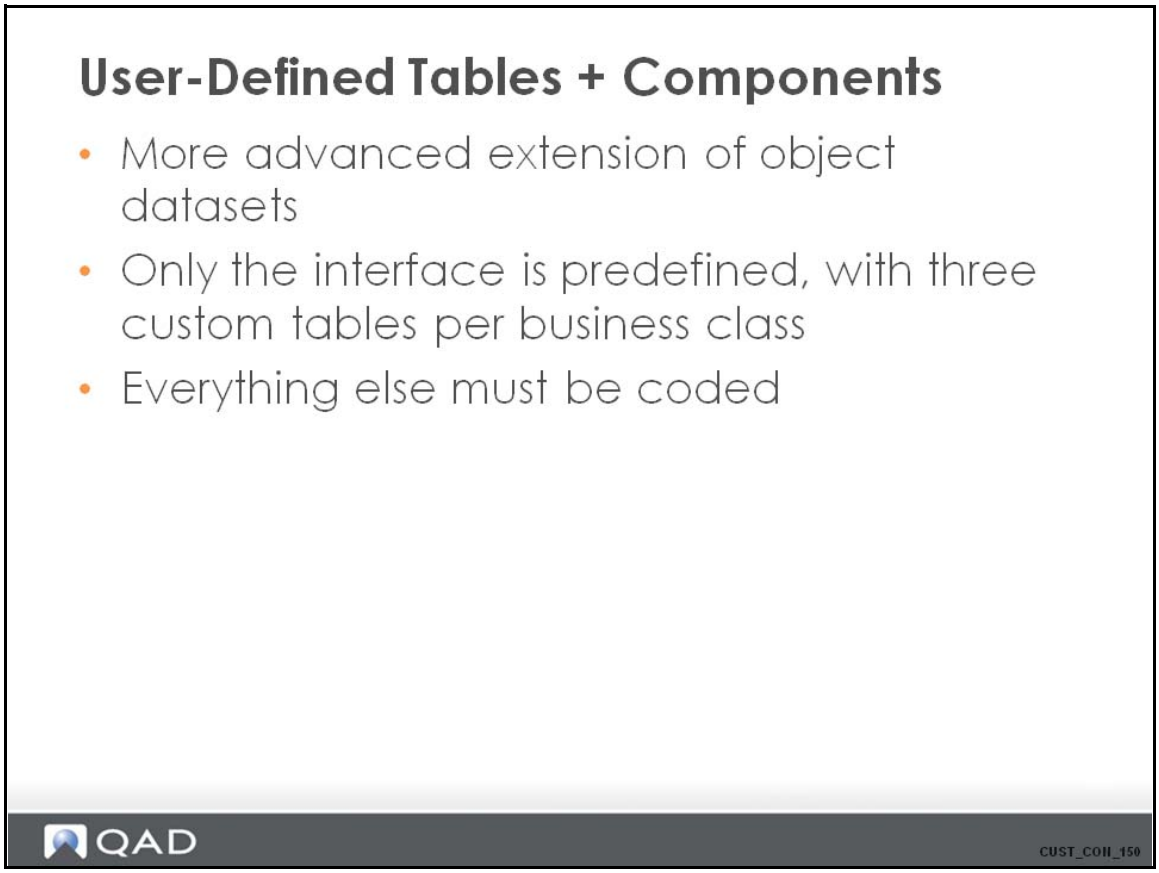
A session instance is only valid if the user is authenticated and granted access to the application.

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

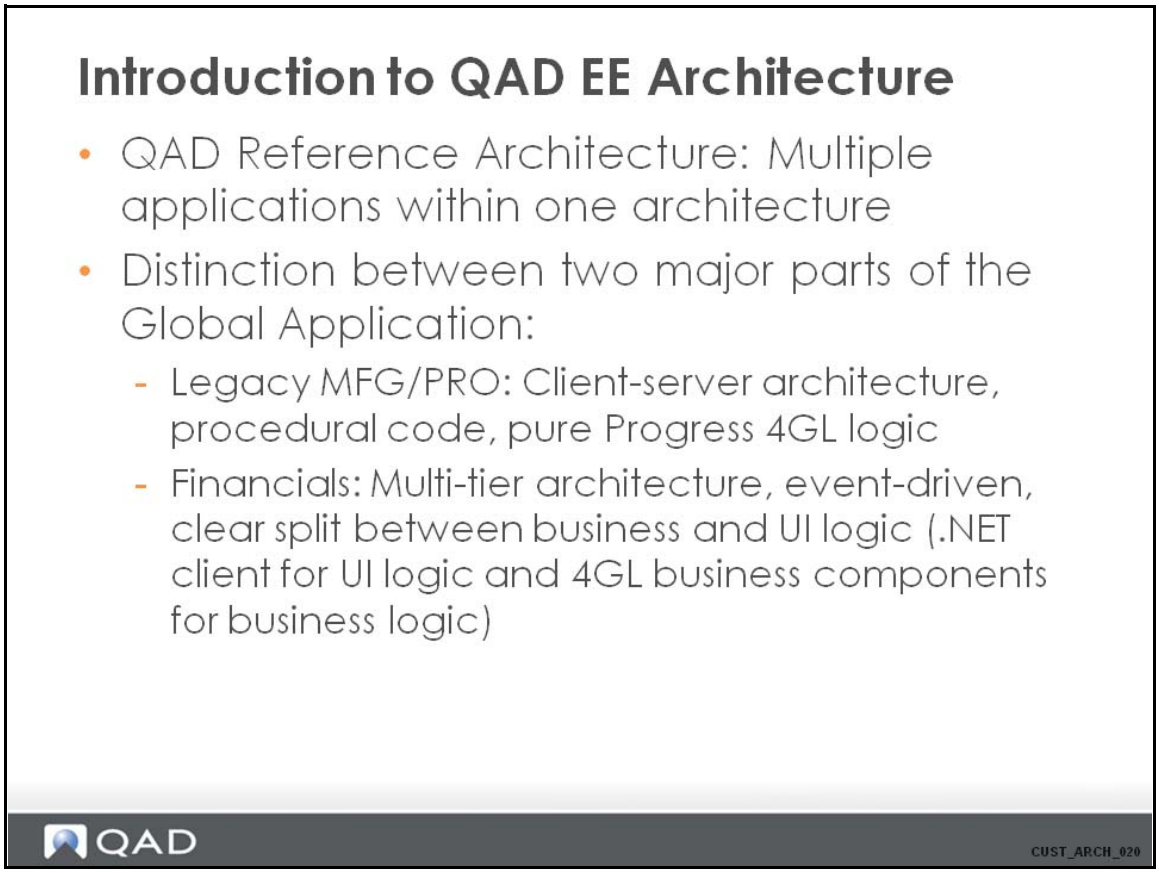


User-Defined Tables + Components



Non-intrusive Customization: Architecture

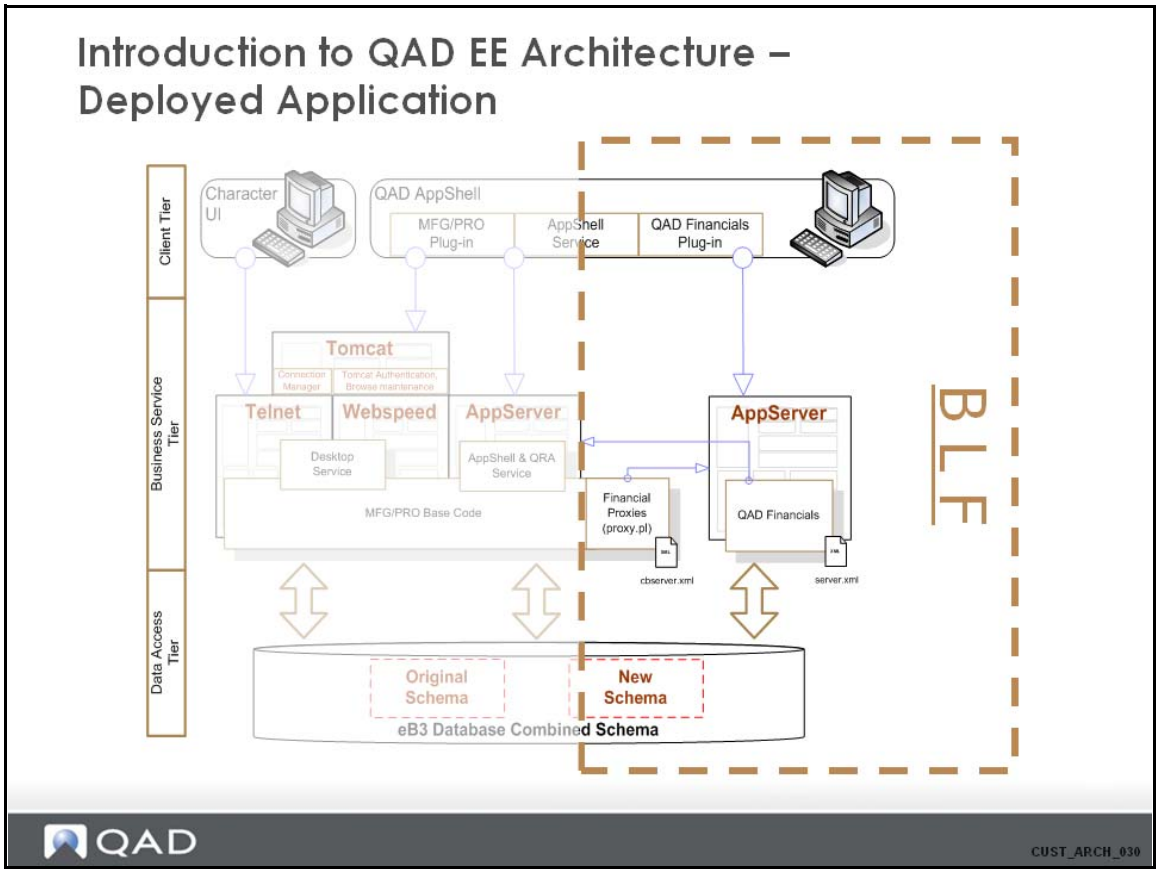
Introduction to QAD EE Architecture



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.

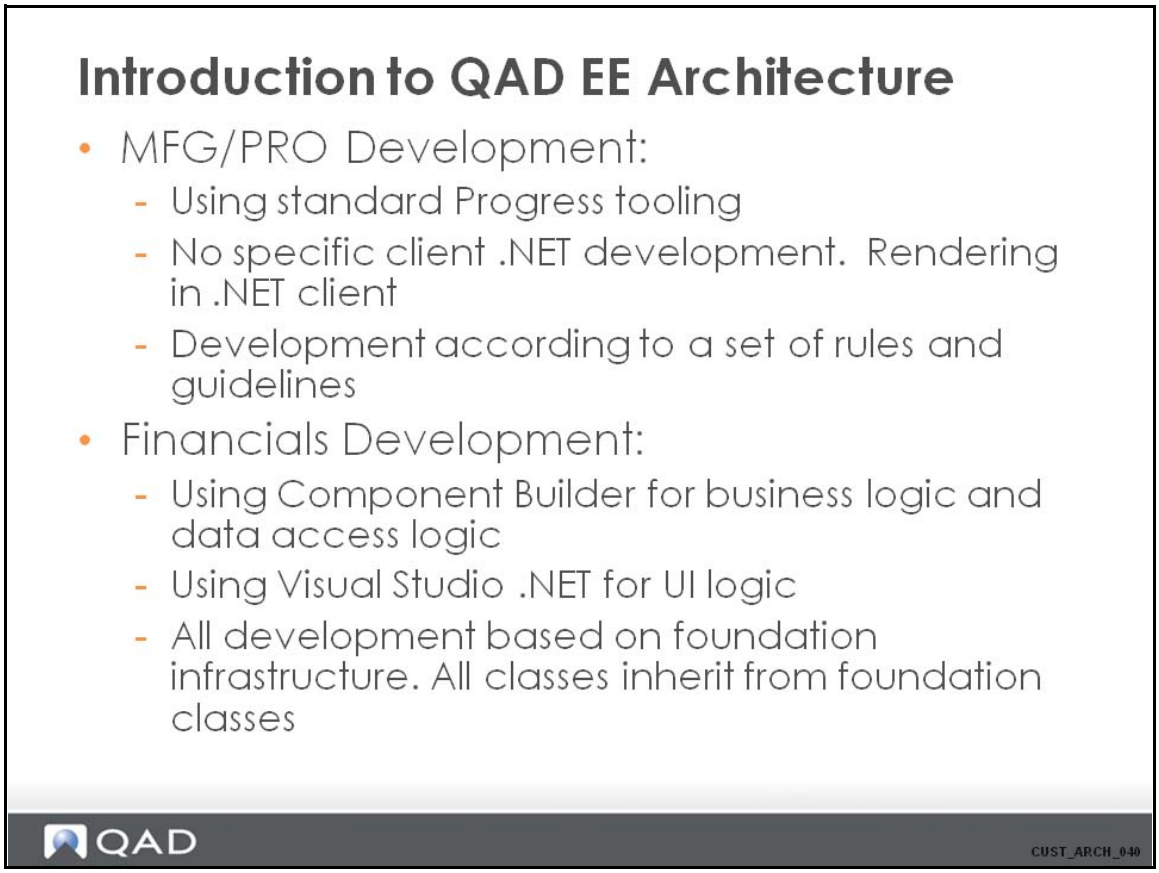
The general rule of the architecture is a clear separation between UI / BL / Data access for the future.

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



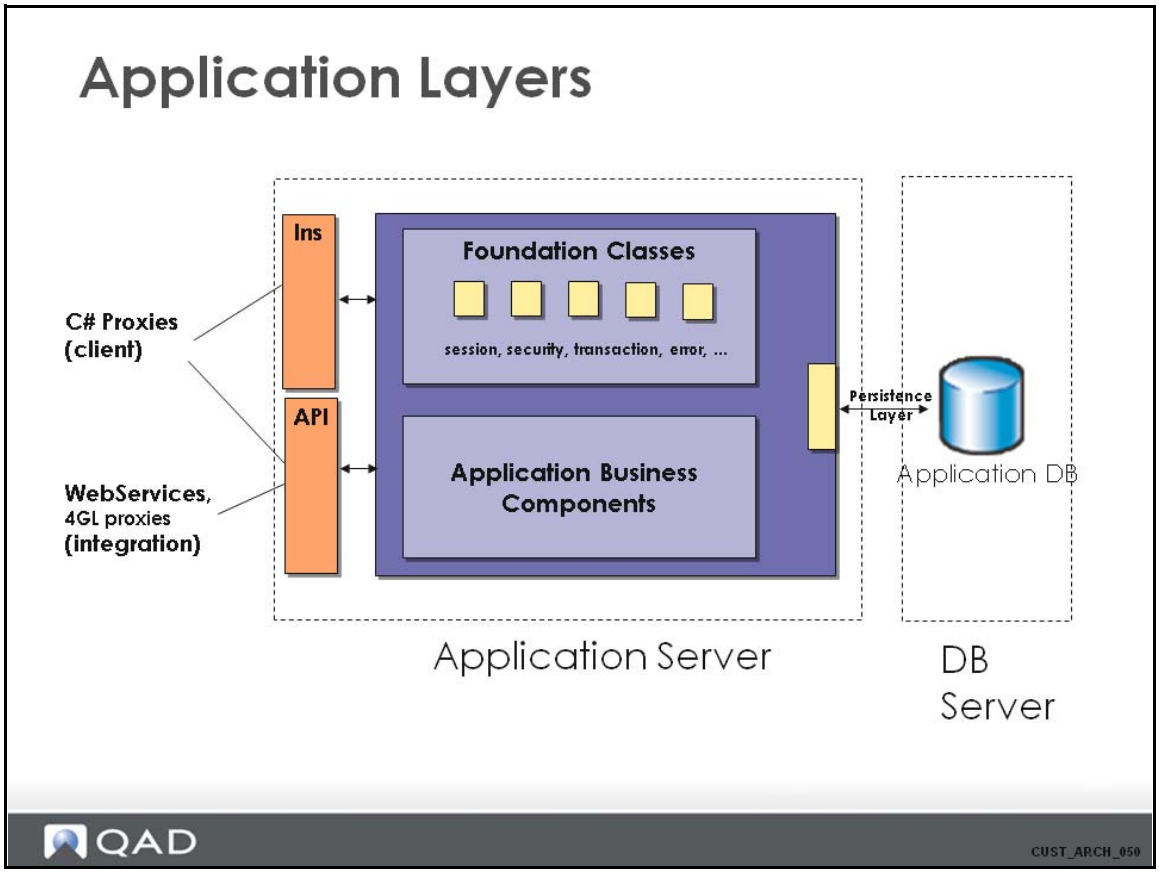
MFG/PRO code is written in the traditional style. This means that there is basically one layer containing a mixture of business, UI, and database access logic. This is the typical approach for classical, procedural-written, client-server code. The .NET UI is rendered and does not contain real application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



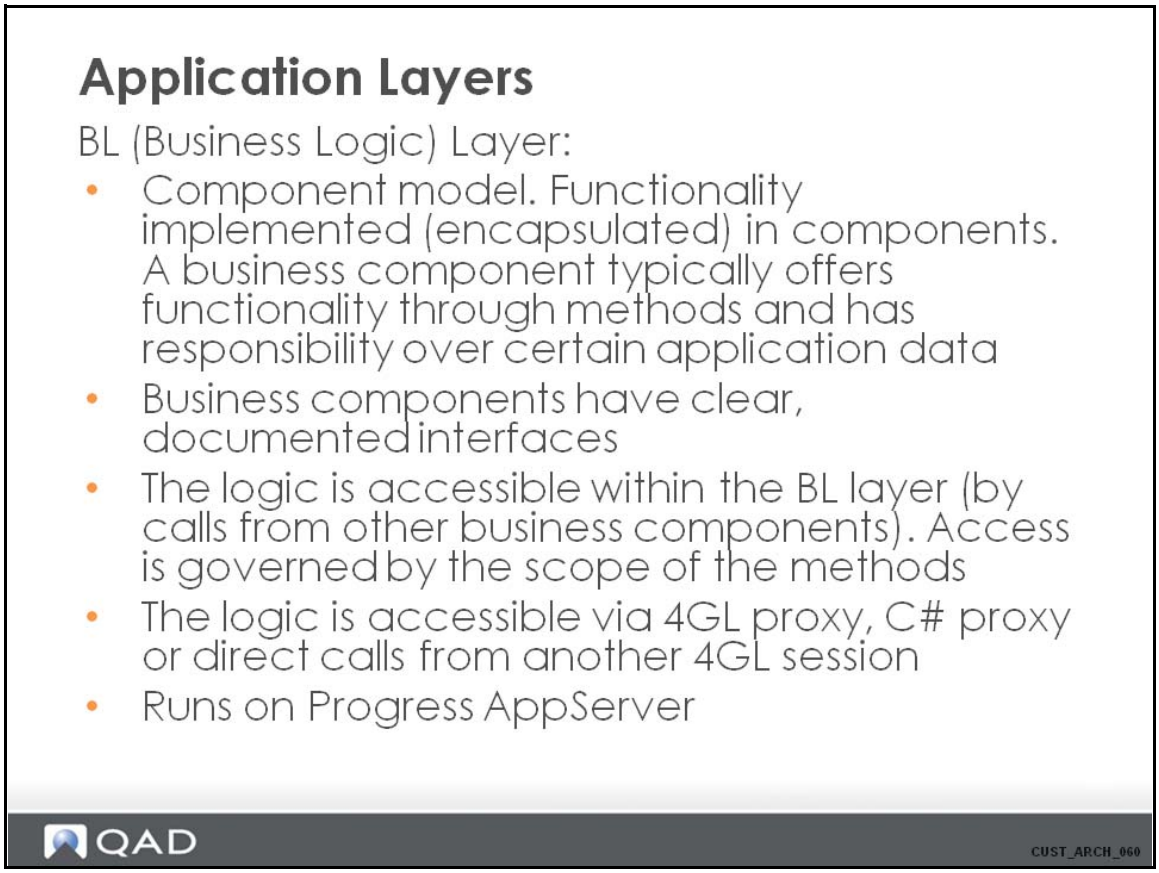
This slide shows the final result when the Enterprise Financials application is deployed.

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



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).

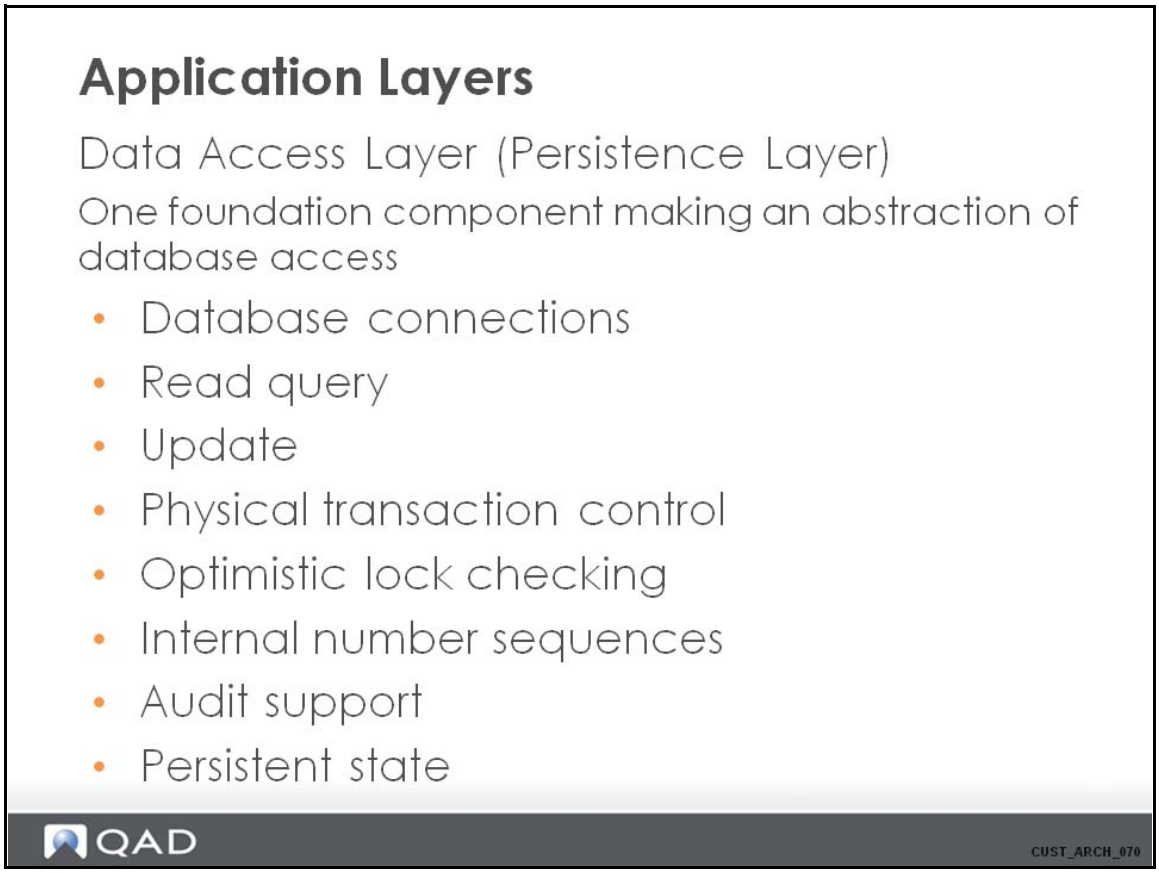
Scope of Methods

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



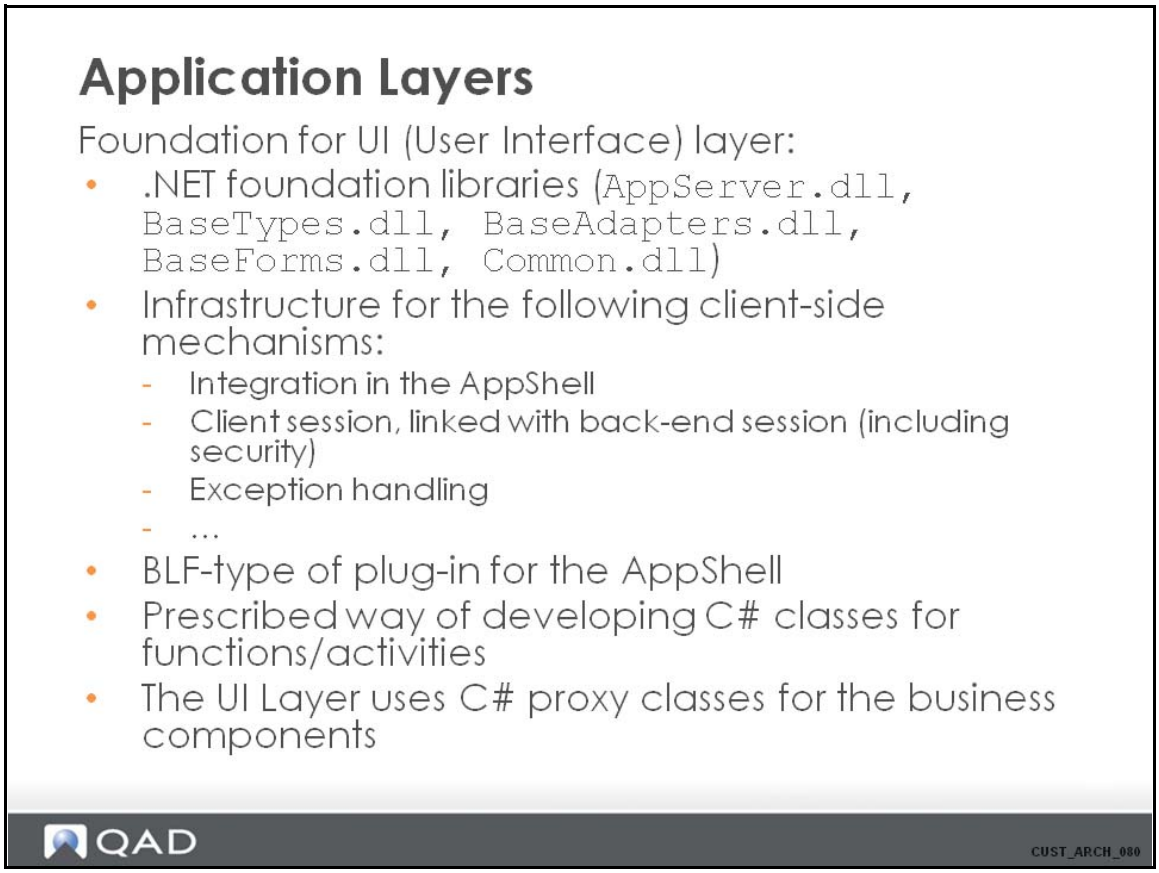
Database connections are performed at runtime. Databases do not need to be connected at the startup of the AppServer sessions.

The following sections explain logical transactions and optimistic locking:

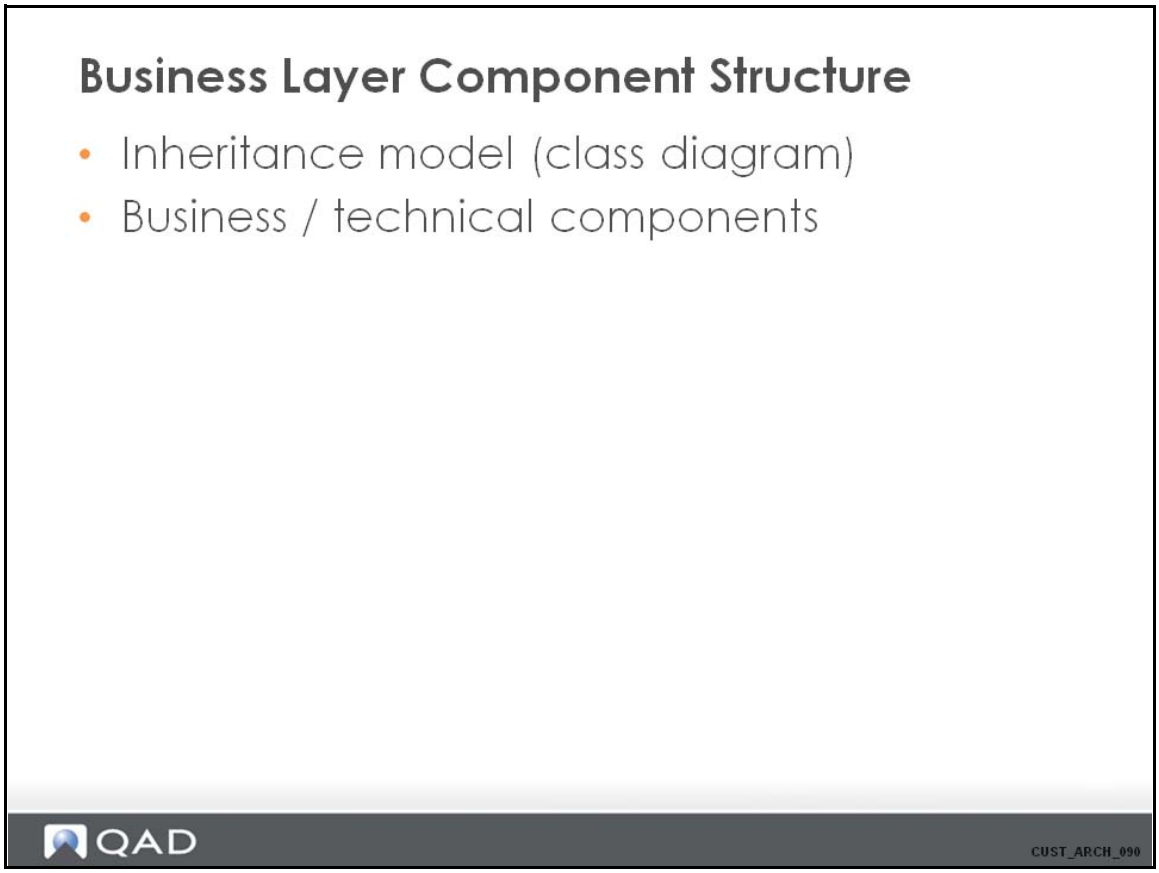
- For a typical classical transaction as used in MFG/PRO: As soon as a function is started, and data displays on the screen, a physical transaction is active. The records and data to be changed in the function are read with a find/for each with exclusive-lock (or share-lock). The updates are done directly 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.

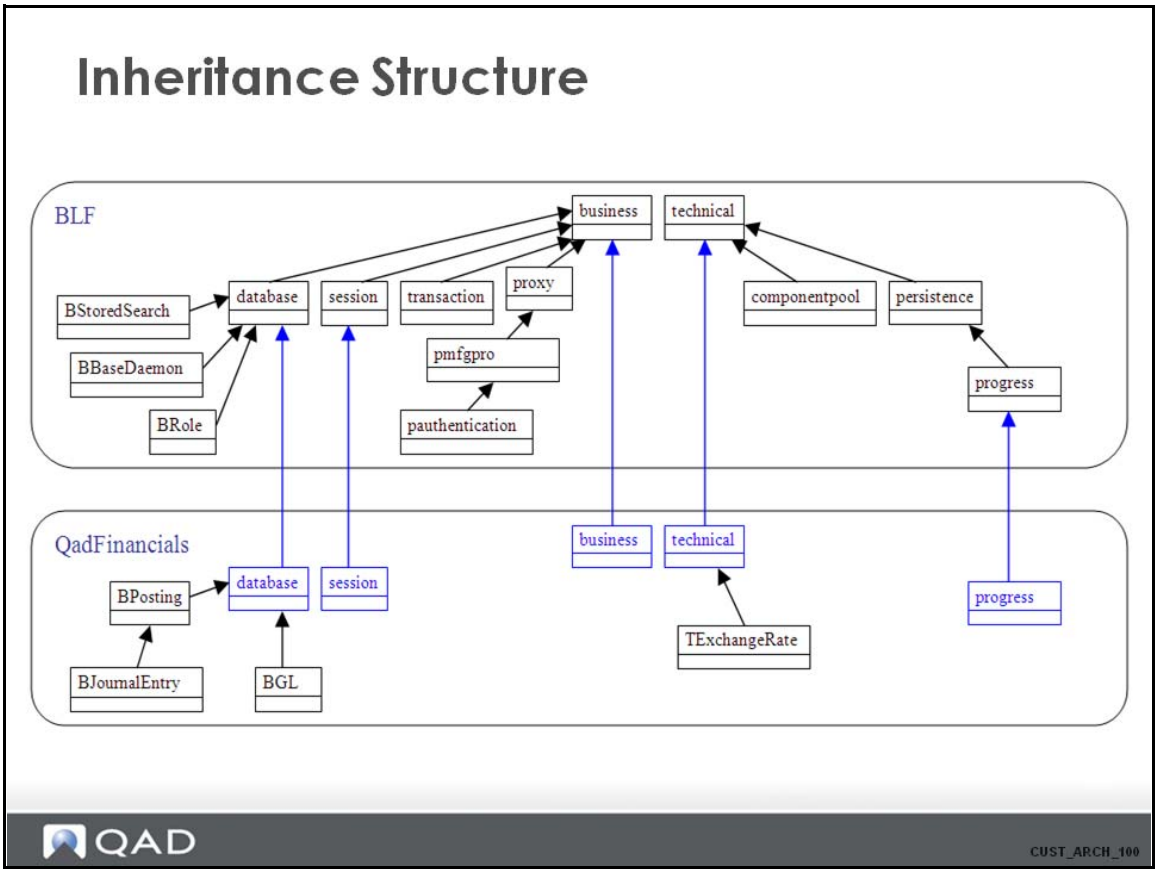
Application Layers



Business Layer Component Structure



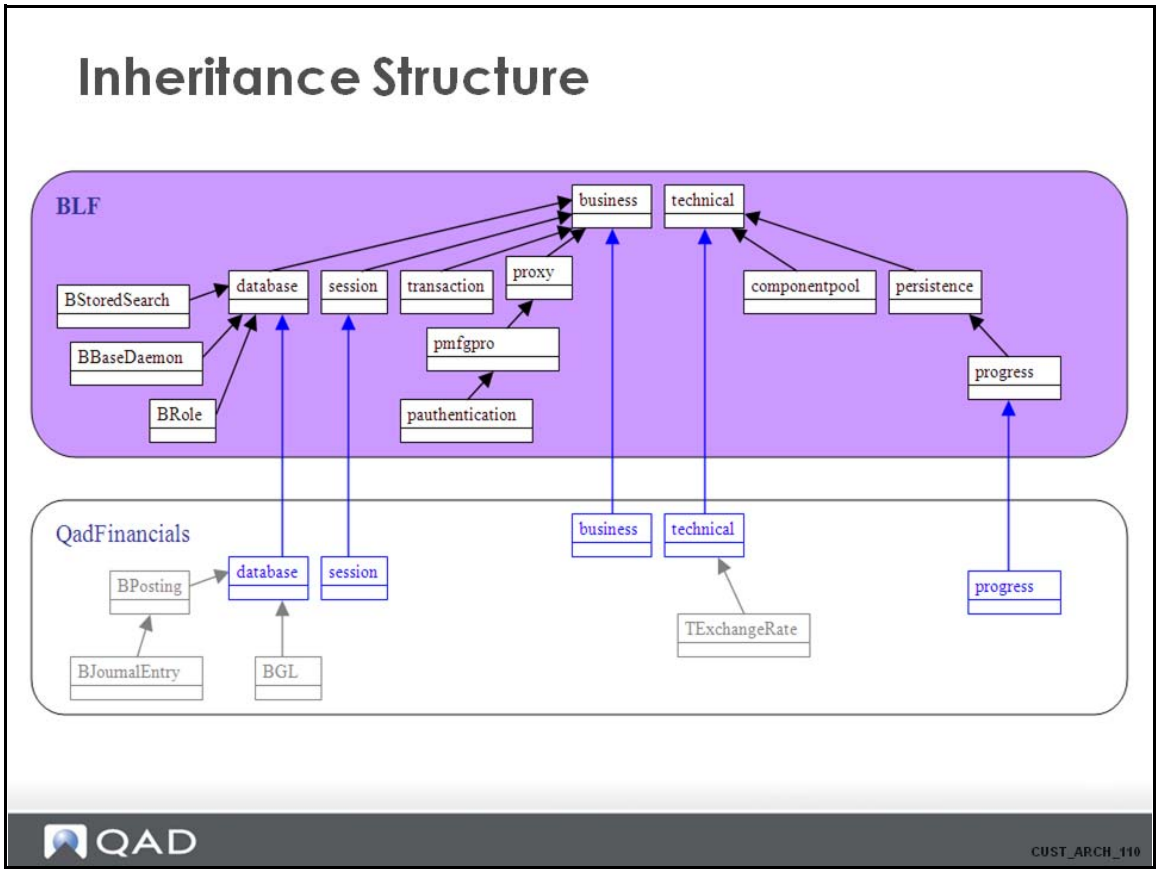
Inheritance Structure



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

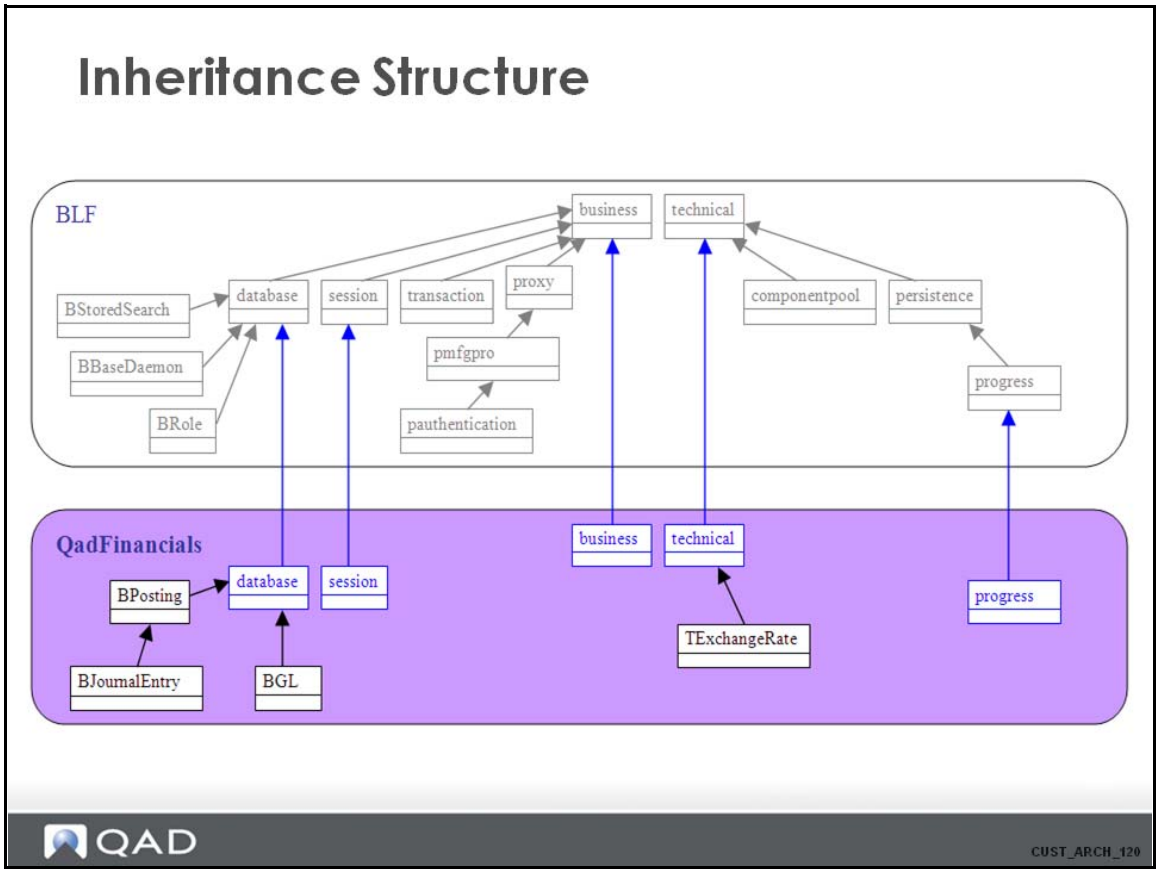


BLF:

- “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).

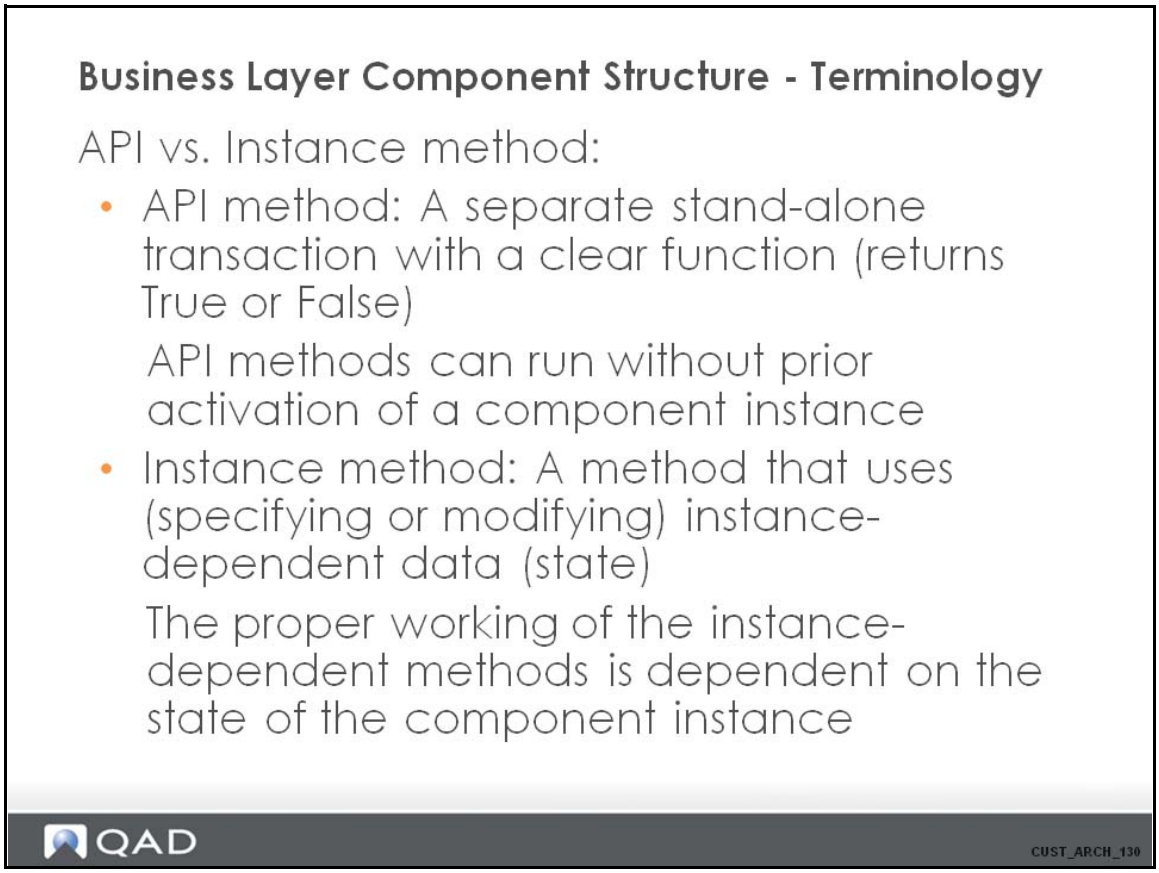
Inheritance Structure



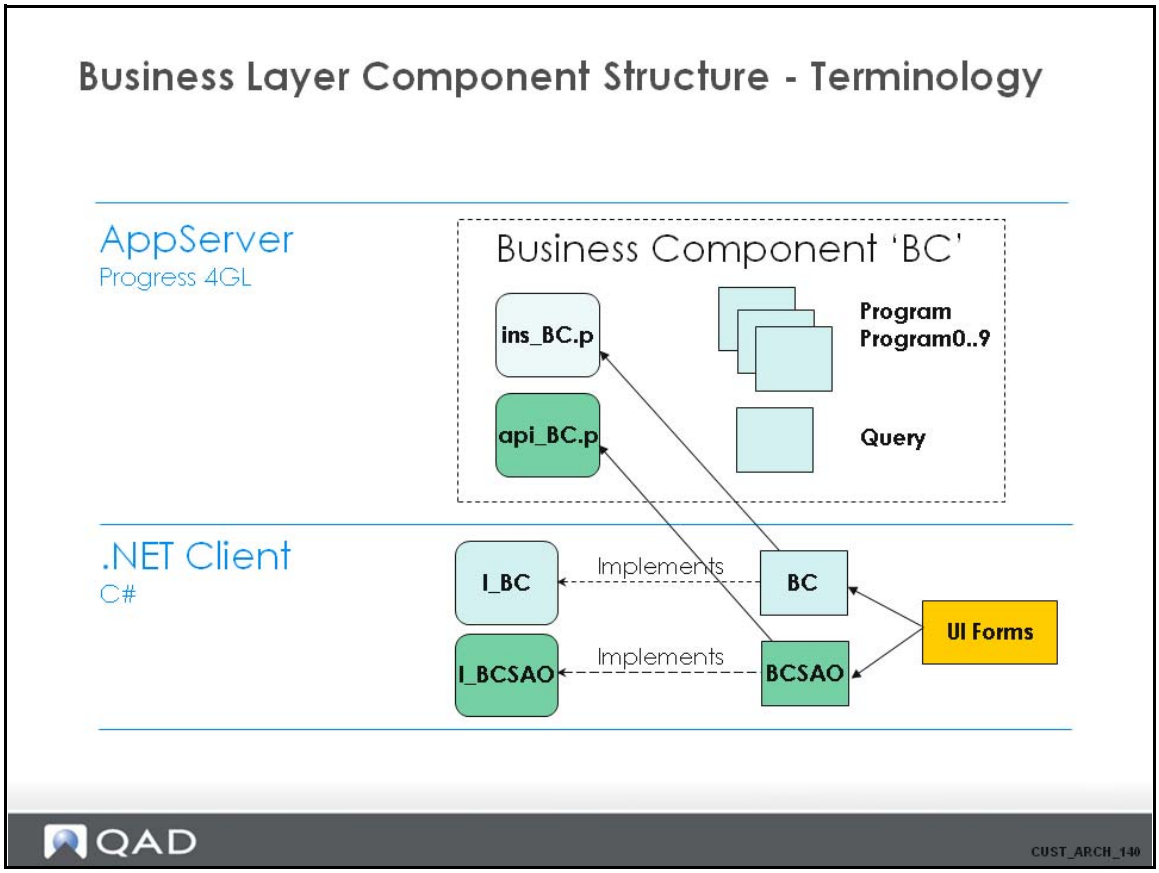
QadFinancials:

- 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.

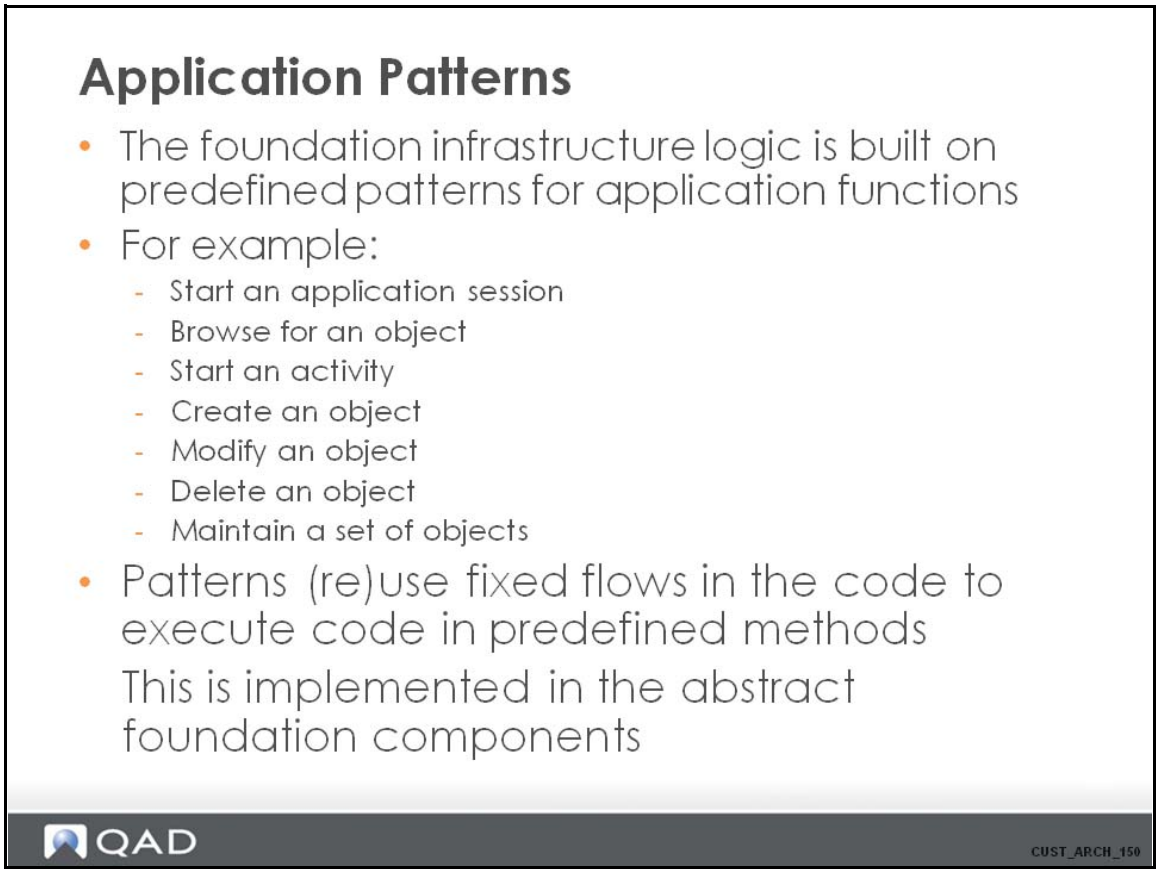
Business Layer Component Structure - Terminology



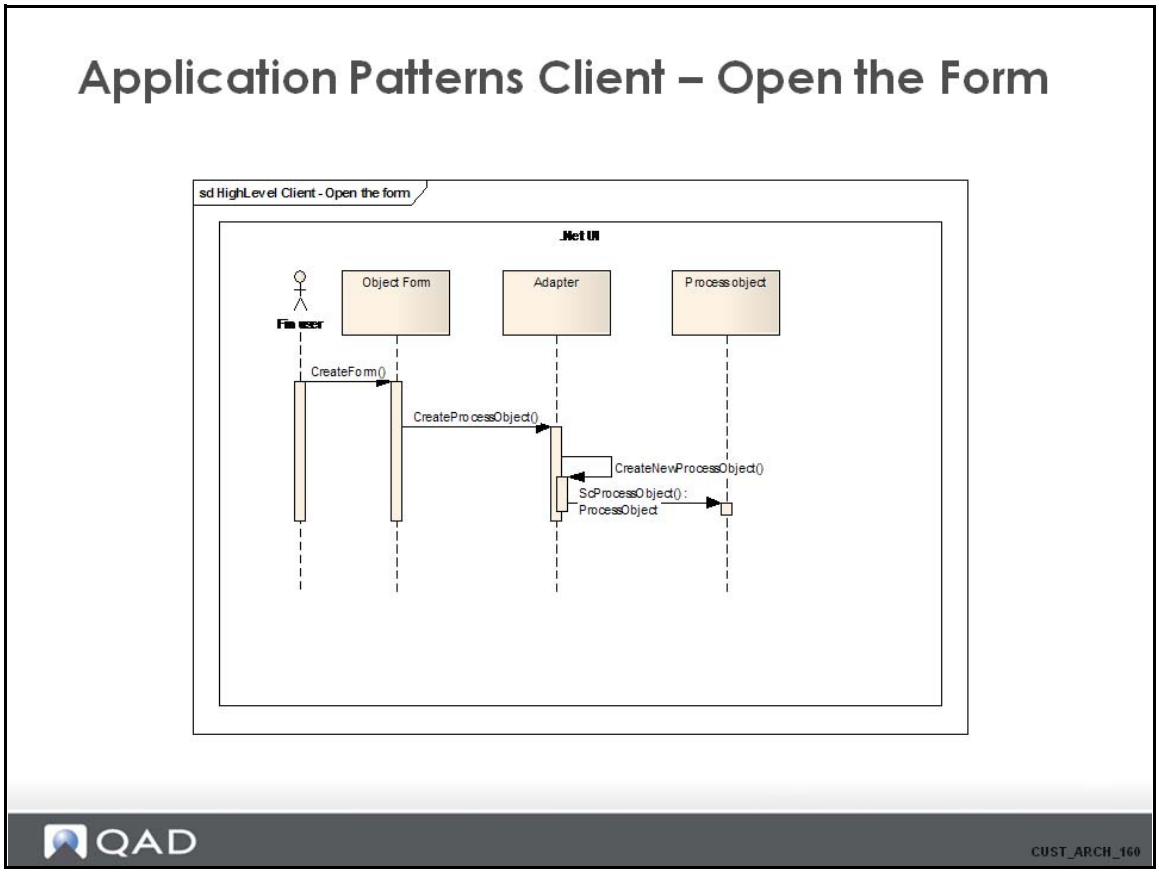
Business Layer Component Structure - Terminology



Application Patterns Overview



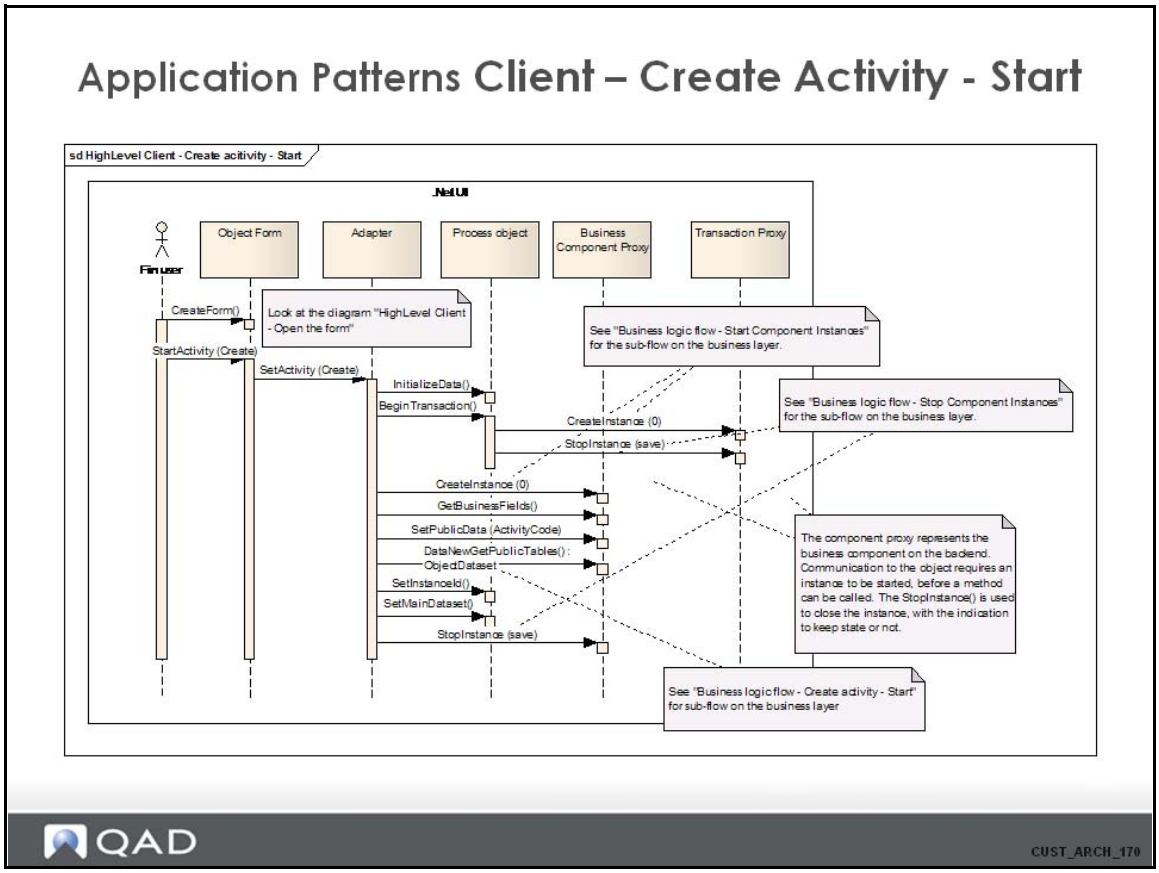
Application Patterns Client



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 'BGl.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

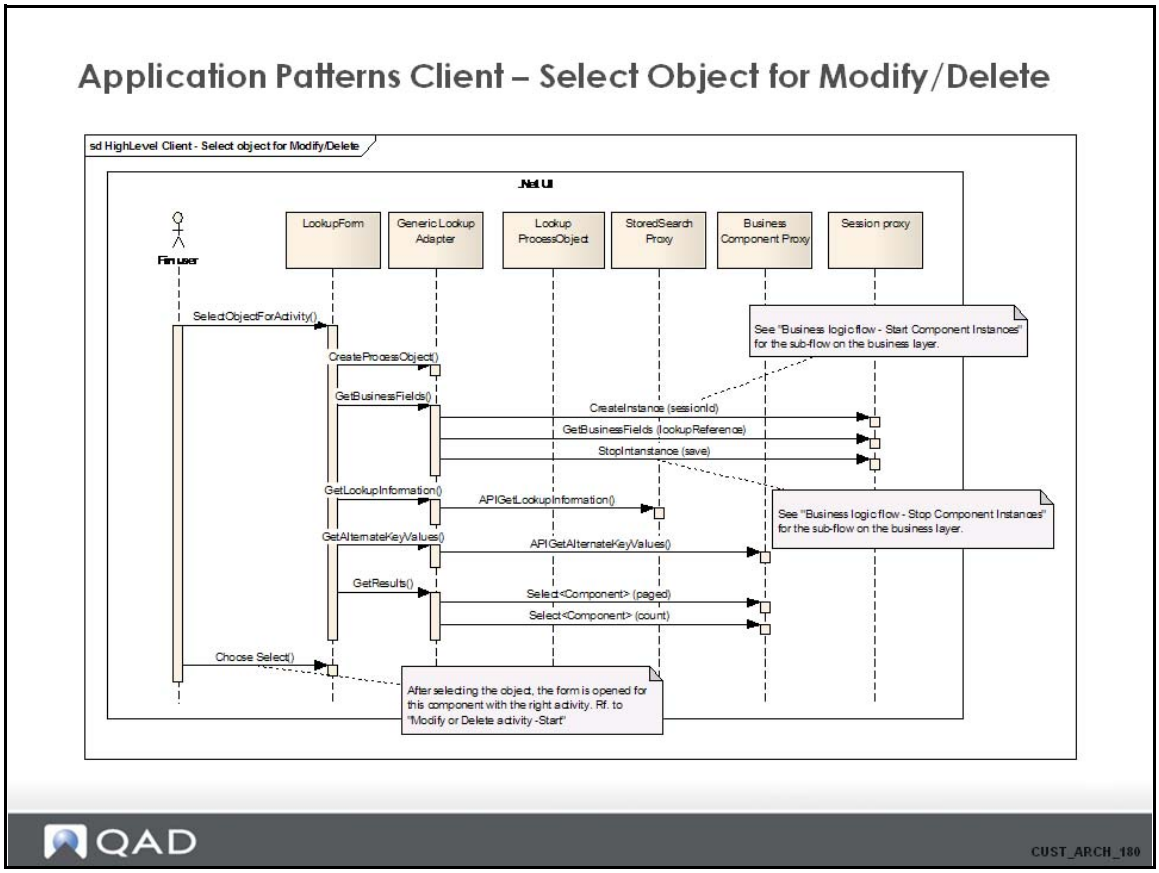


This diagram shows the flow that is executed when a Create activity is started. It assumes that the form is already instantiated. It stops when the form is displayed on the screen and is ready to receive input from the end user.

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.

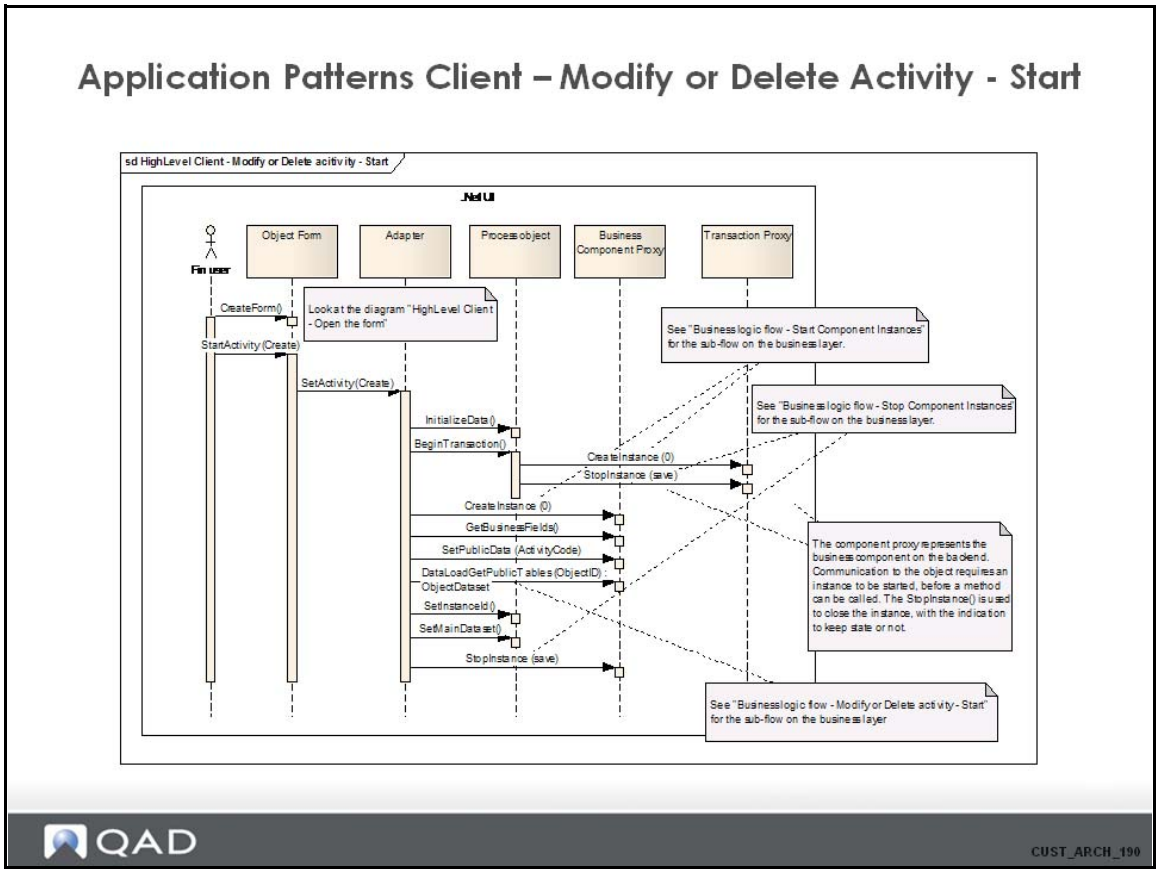
Application Patterns Client



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.

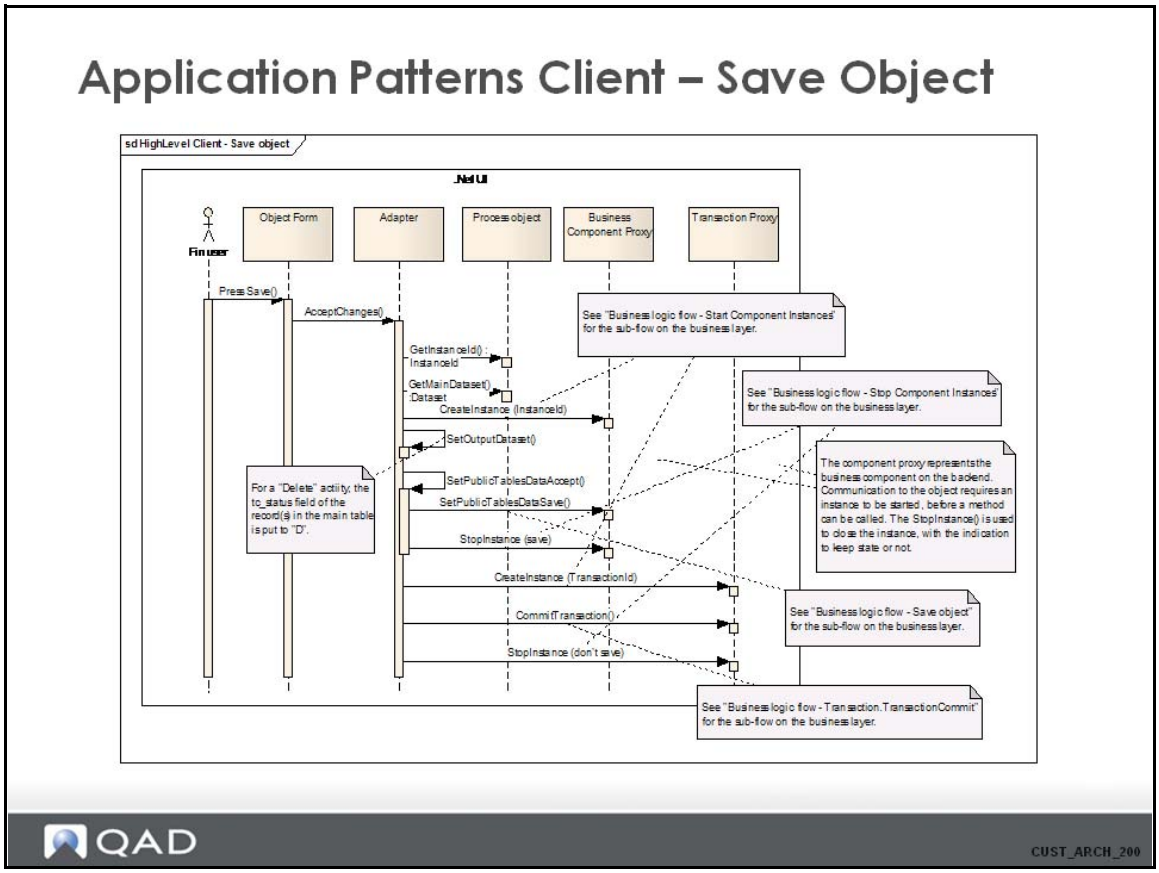
This is a common flow for all activities that require an object be selected before the object form for the activity is started (for example “Modify” or “Delete”).

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

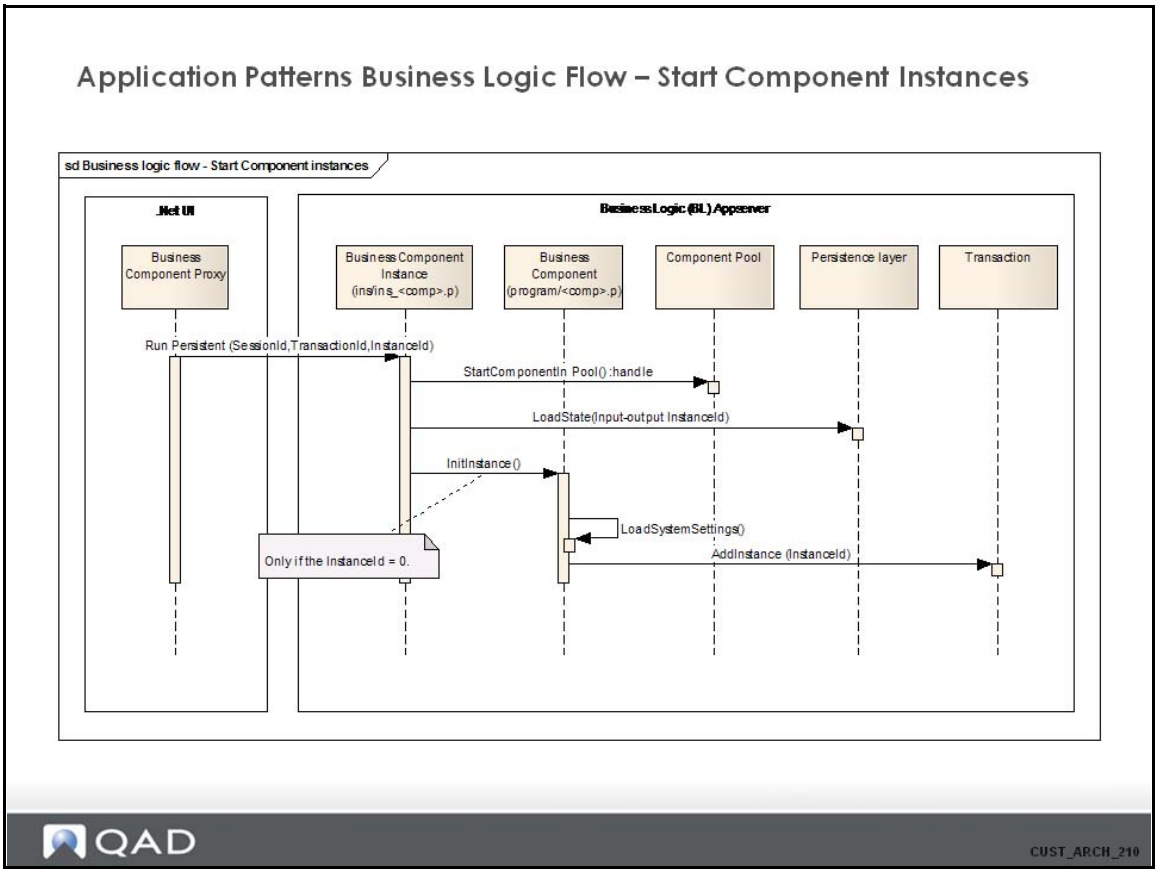


This diagram shows the flow that is executed when the user clicks Save and submits the form.

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.

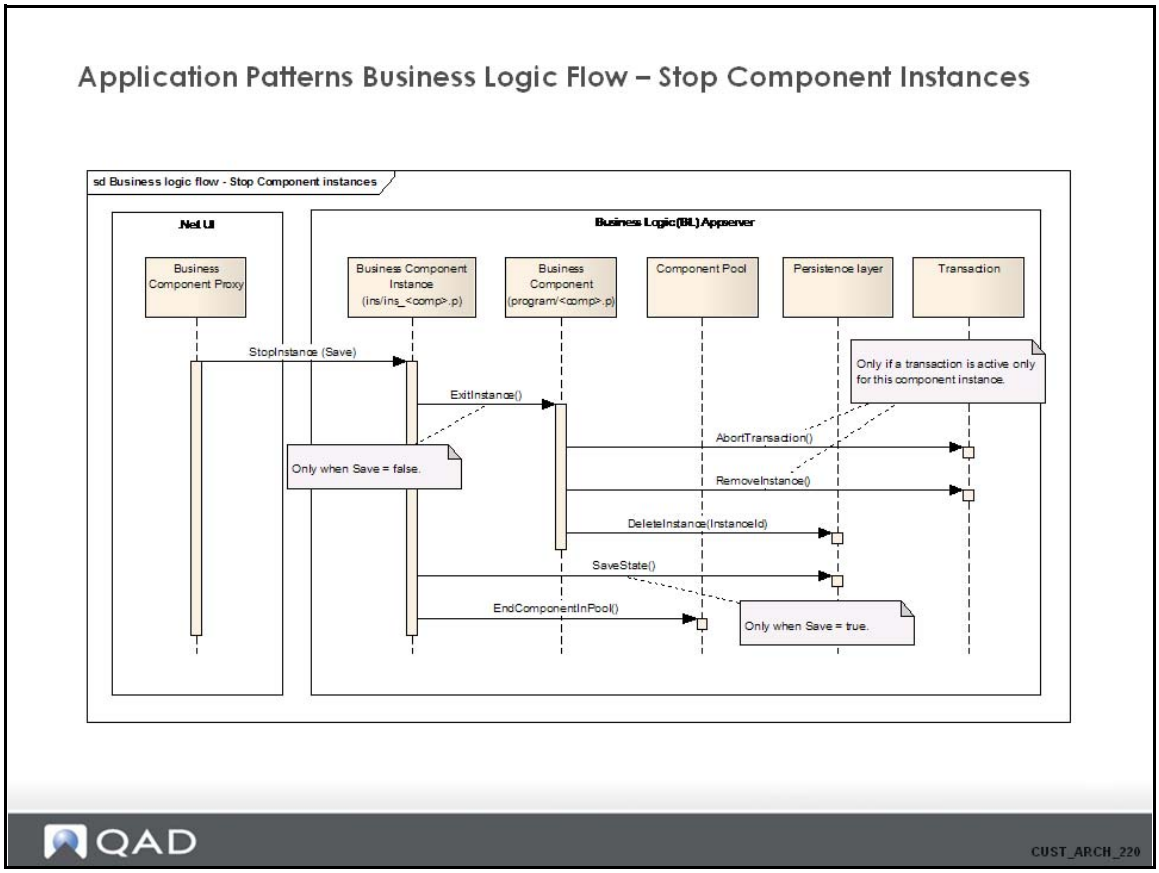
- SessionId is required. It is the component instance ID of the session started previously, which contains the information about the current session.
- InstanceId: If it is 0, it is assumed that a real new instance / state must be created. If it is <>0, the system restores the instance data associated 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



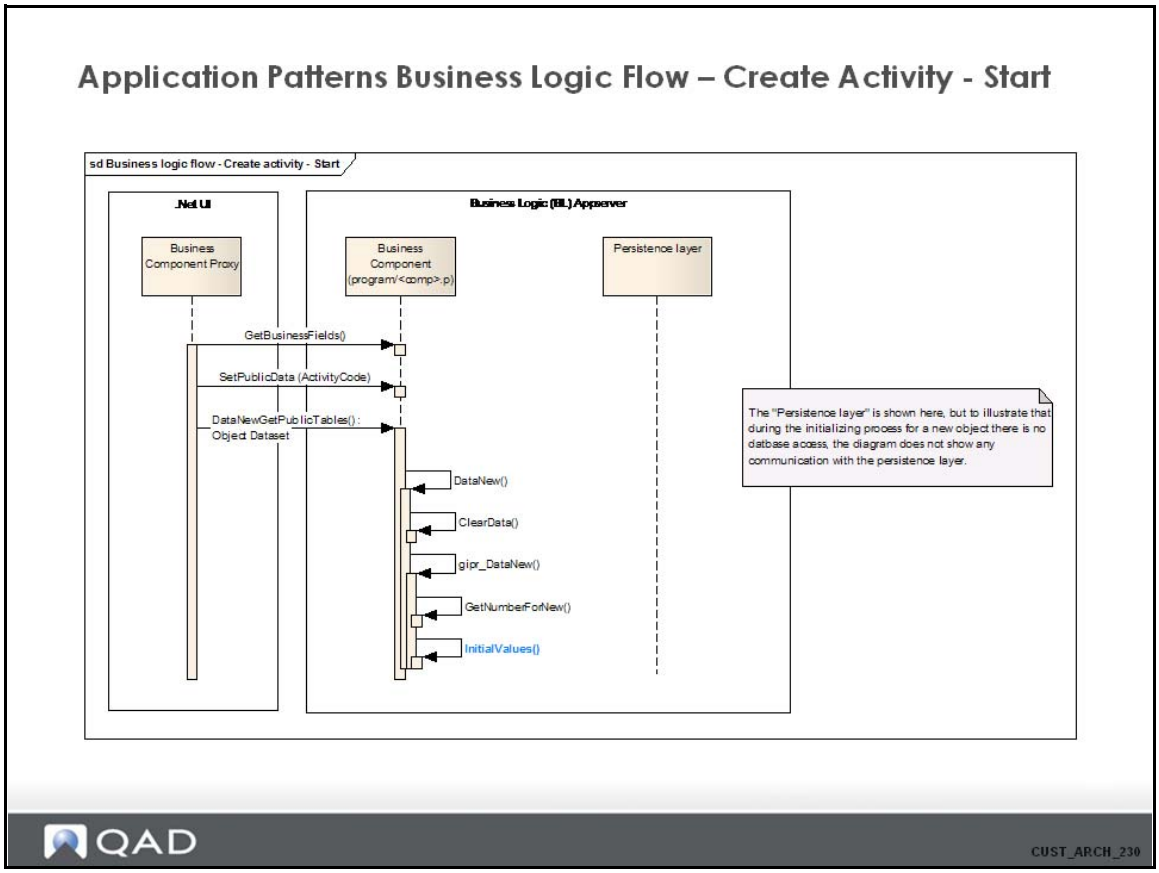
This diagram shows the flow that is executed when a business component instance is stopped. Stopping the instance from the UI means that the instance is no longer needed, and the lock on the stateless connection can be released. This is primarily done by stopping the persistent procedure.

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

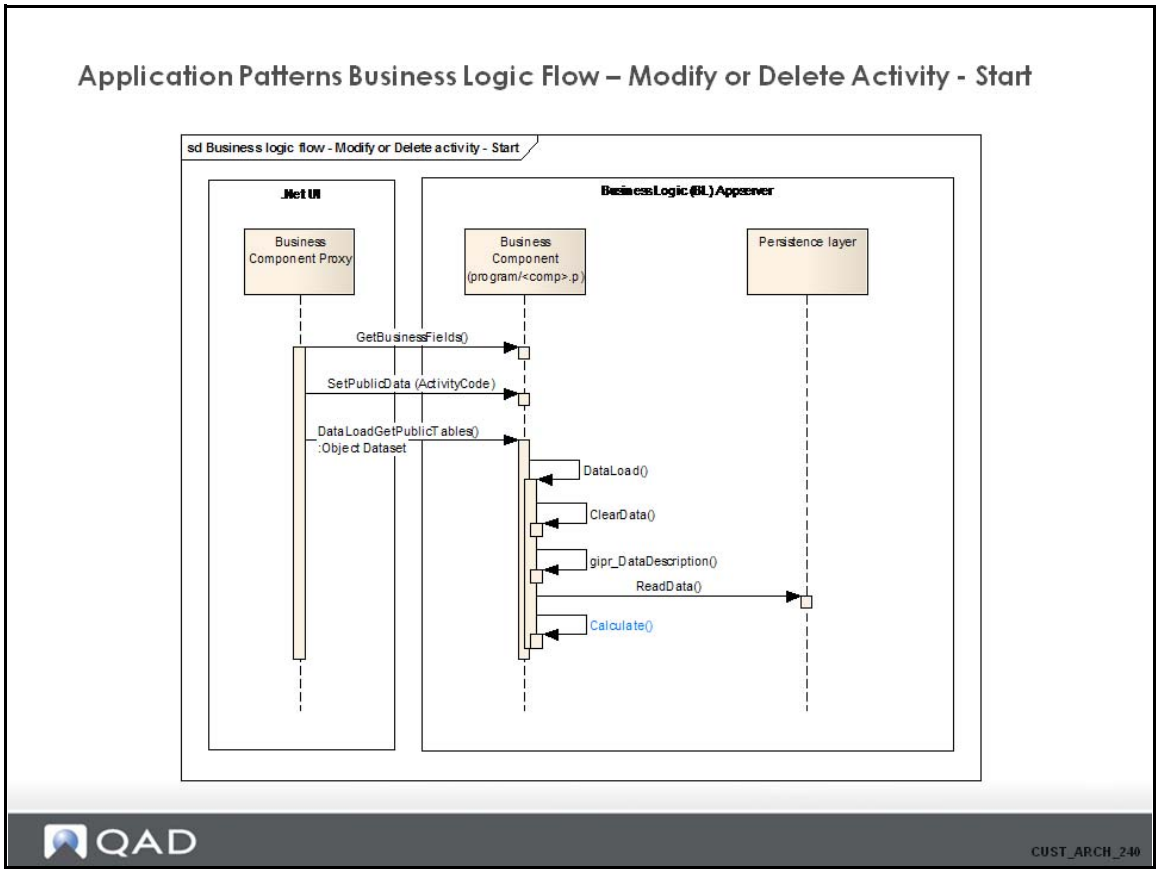


This diagram shows the program flow that is executed on the business logic layer when an object is created.

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.

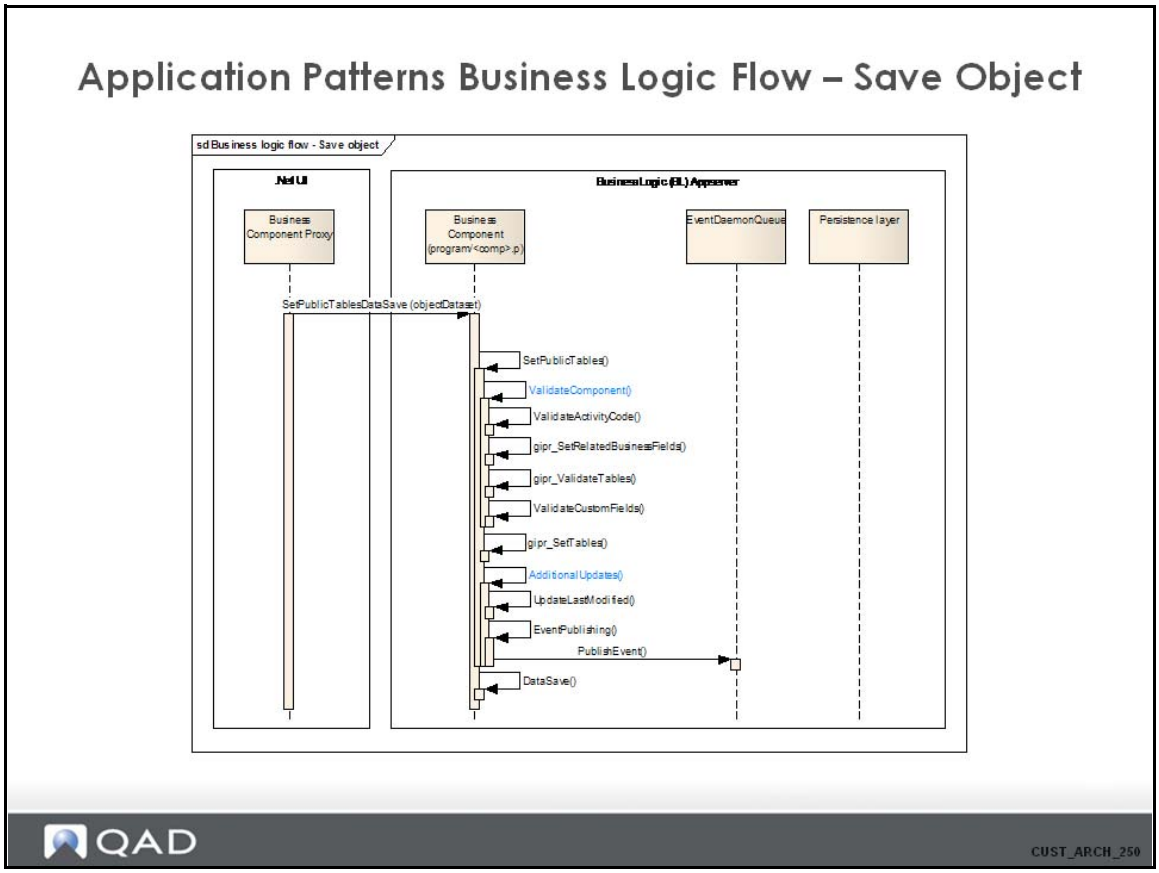
These UI calls are performed when the form is opened in Modify or Delete mode.

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.

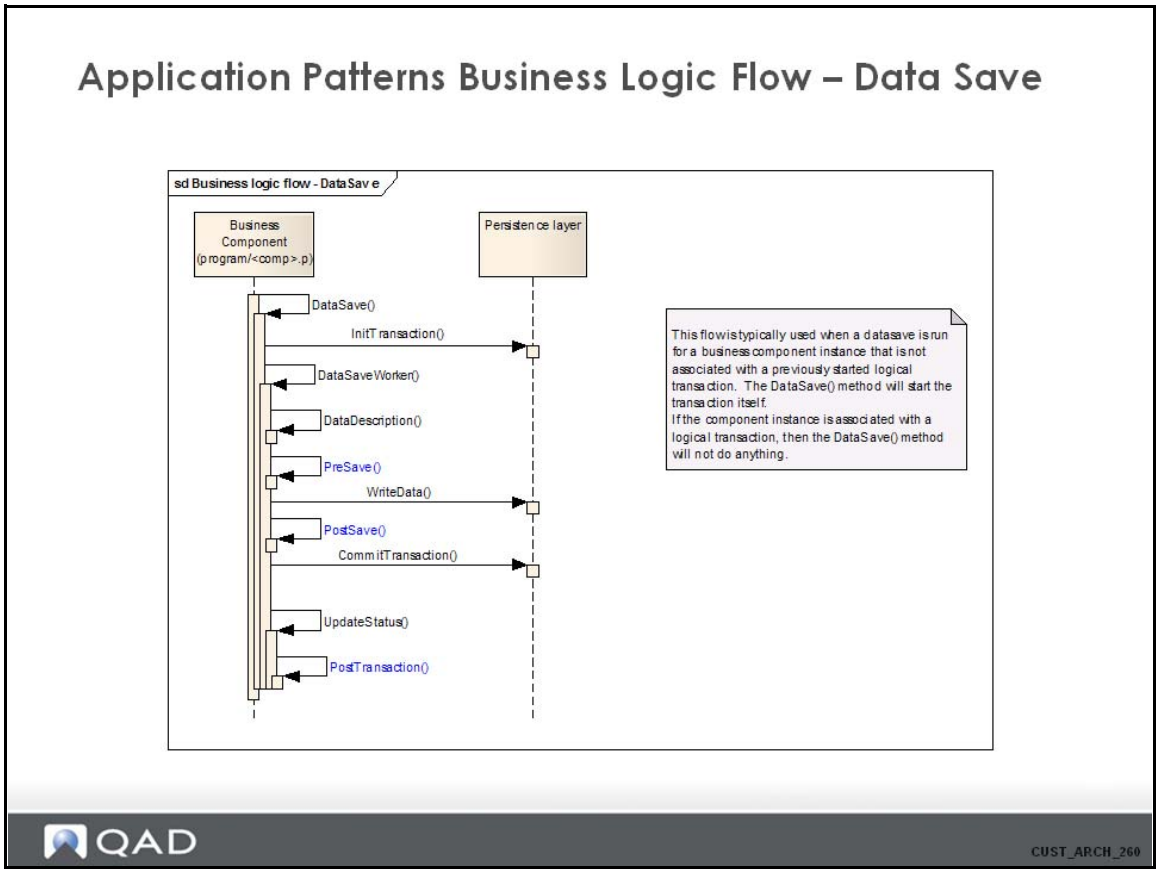
gipr_validateTables is a generated procedure, and contains all validations based on the data model information (for example, mandatory fields and mandatory relations).

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

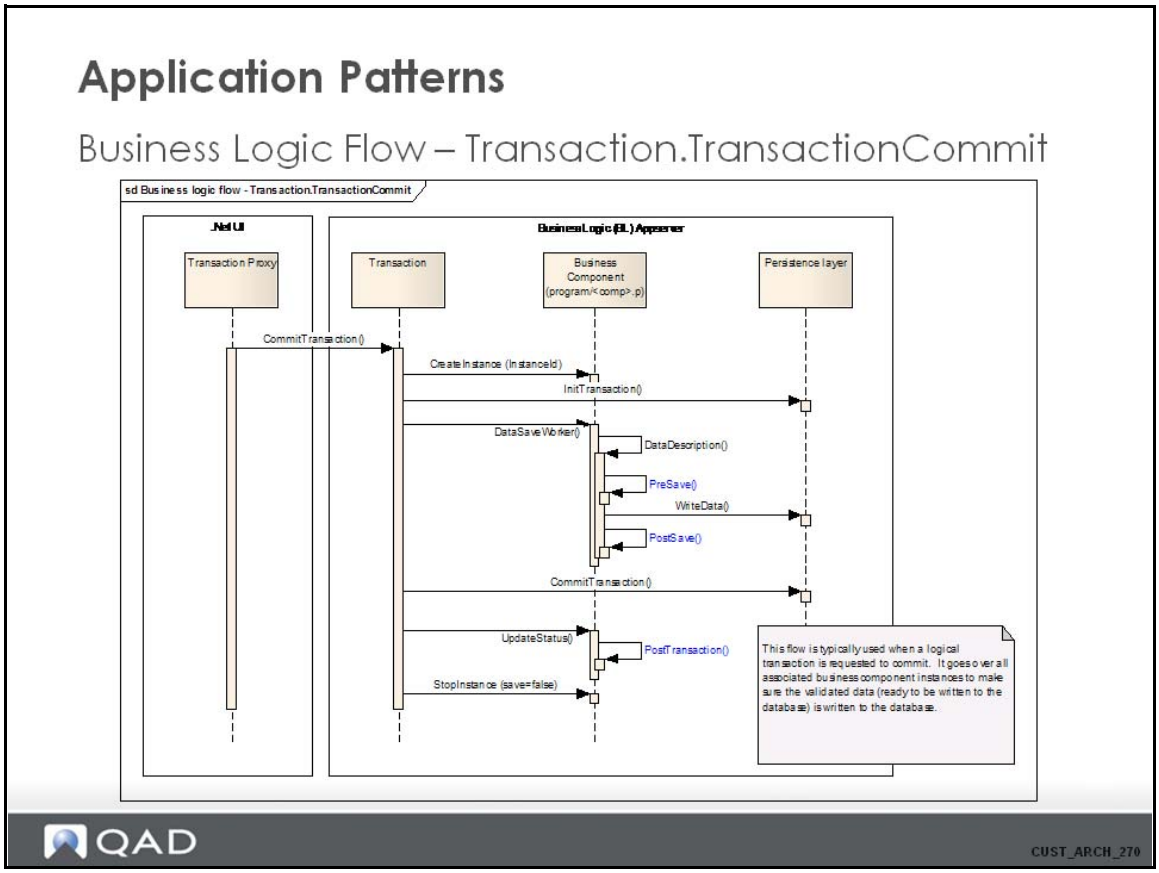


This diagram shows the detail program flow for the execution of the DataSave() method. This flow is only followed when the component instance for which the DataLoad is executed was not previously linked to a logical transaction. In the latter case, the DataSave() does nothing, and the logical transaction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



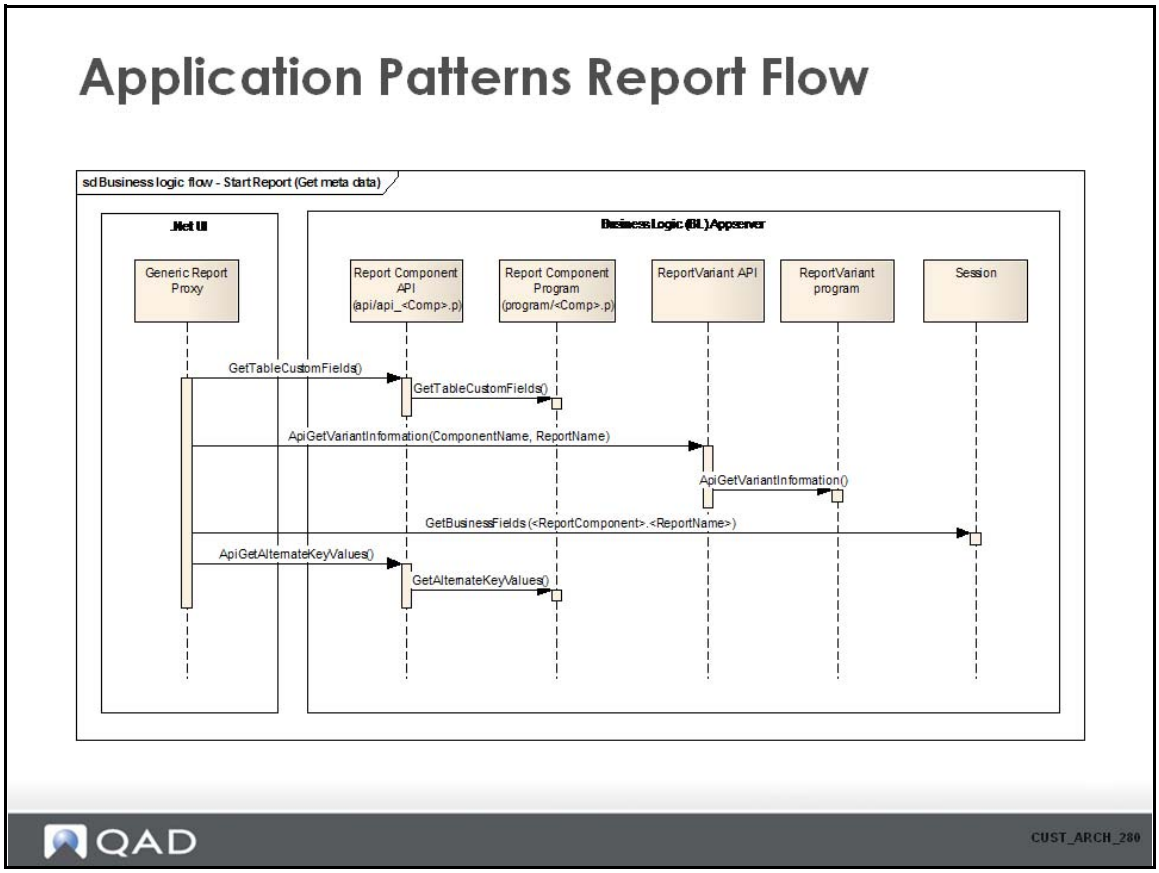
This diagram shows the detail program flow for the execution of the Transaction.CommitTransaction() method.

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

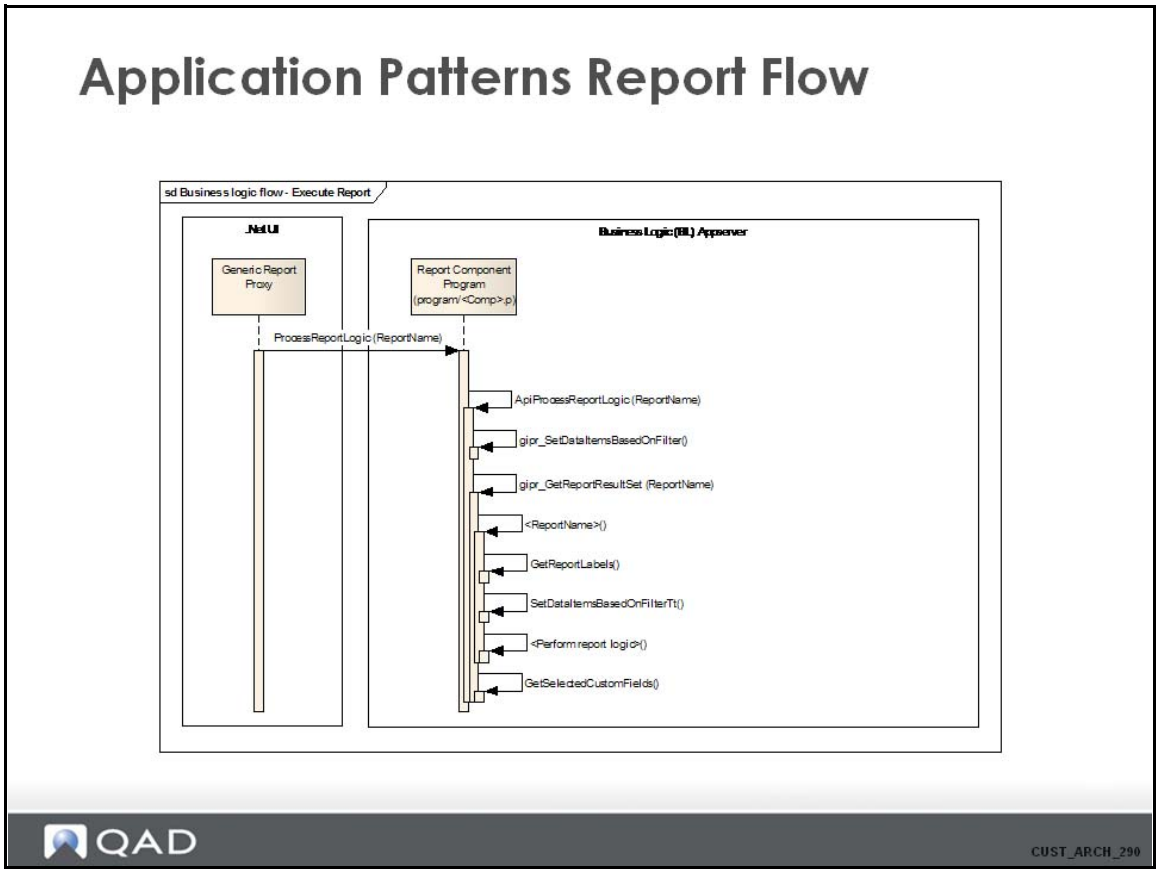
The logic loops through all associated component instances and writes the changed data to the database in sub-transactions. The Transaction component instance itself controls the life span of the physical transaction.

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

Application Patterns Report Flow



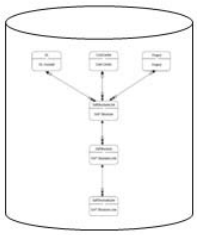
Application Patterns Report Flow



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

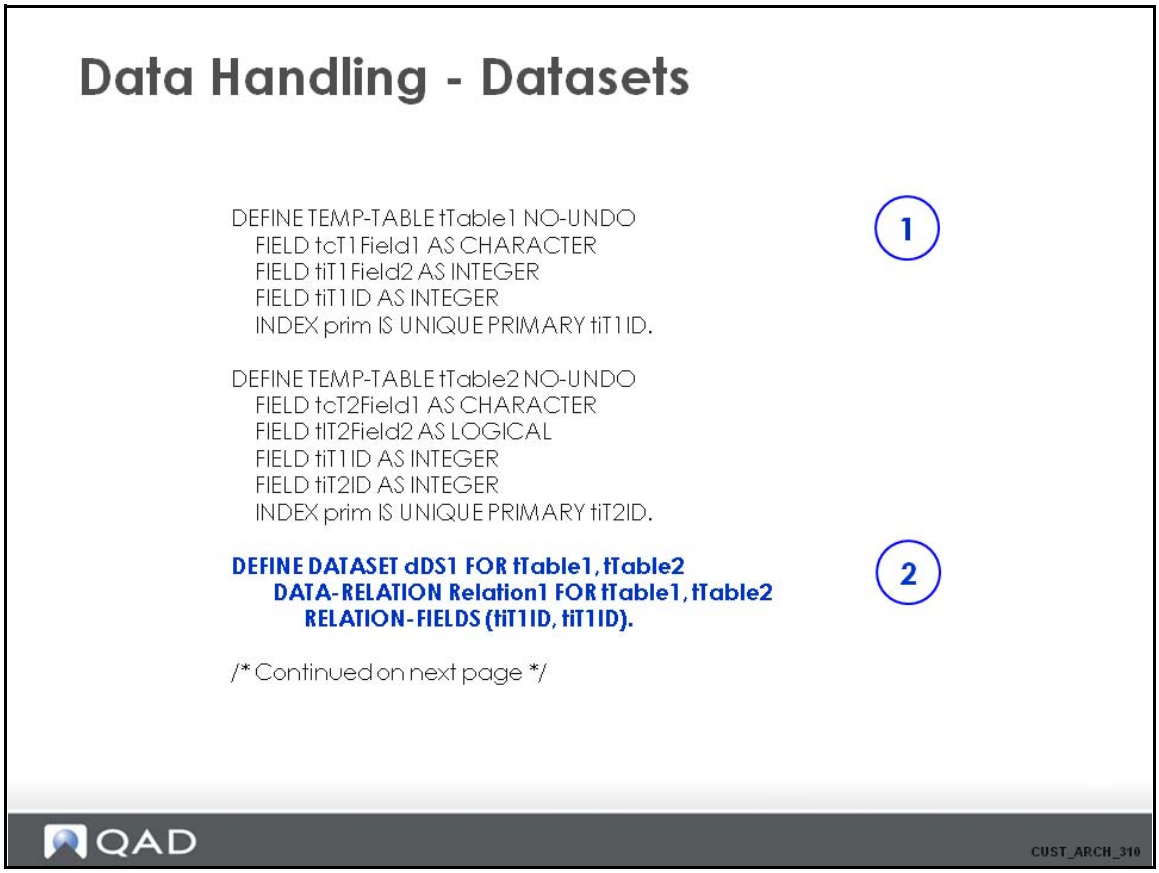


- A dataset can be passed as a parameter to a procedure
- A dataset can easily be serialized to XML format
- A Progress dataset matches a .NET dataset

QAD

CUST_ARCH_300

Data Handling - Datasets



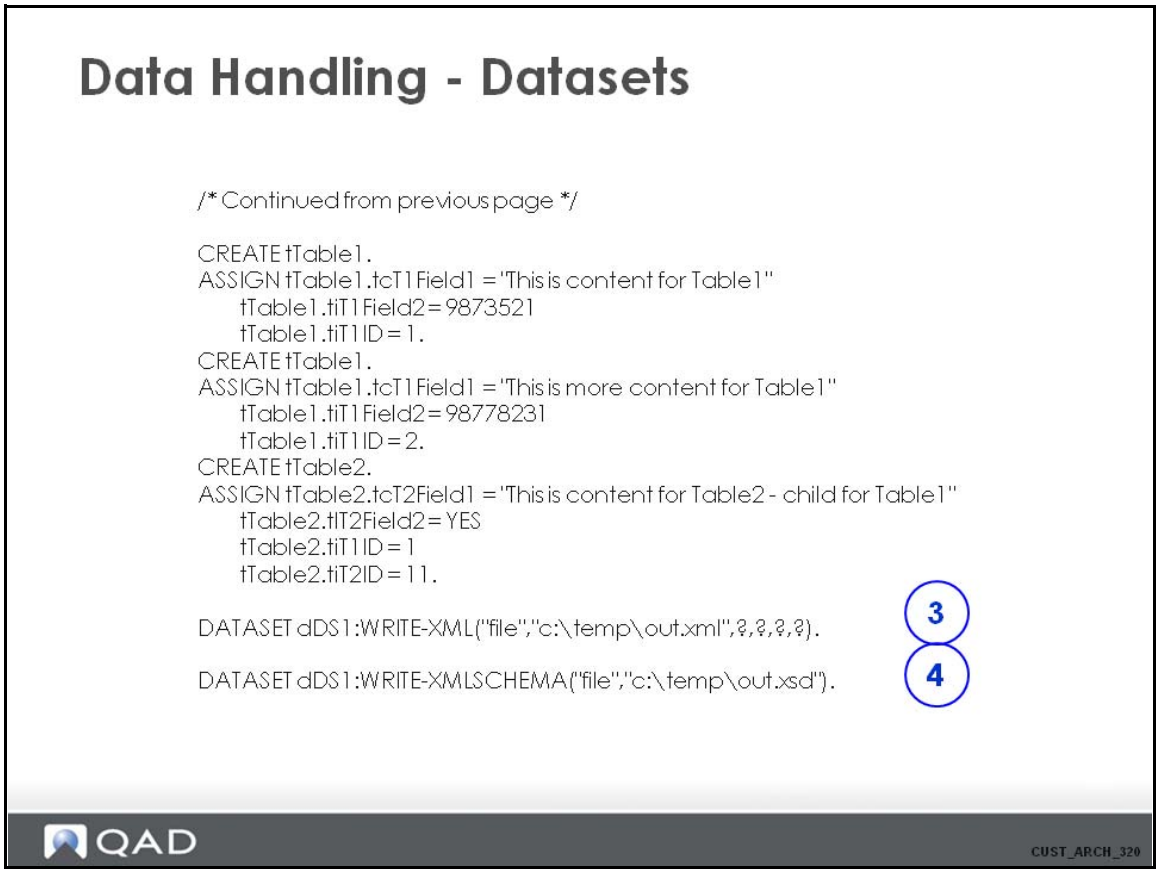
A dataset exists of temp-tables.

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.

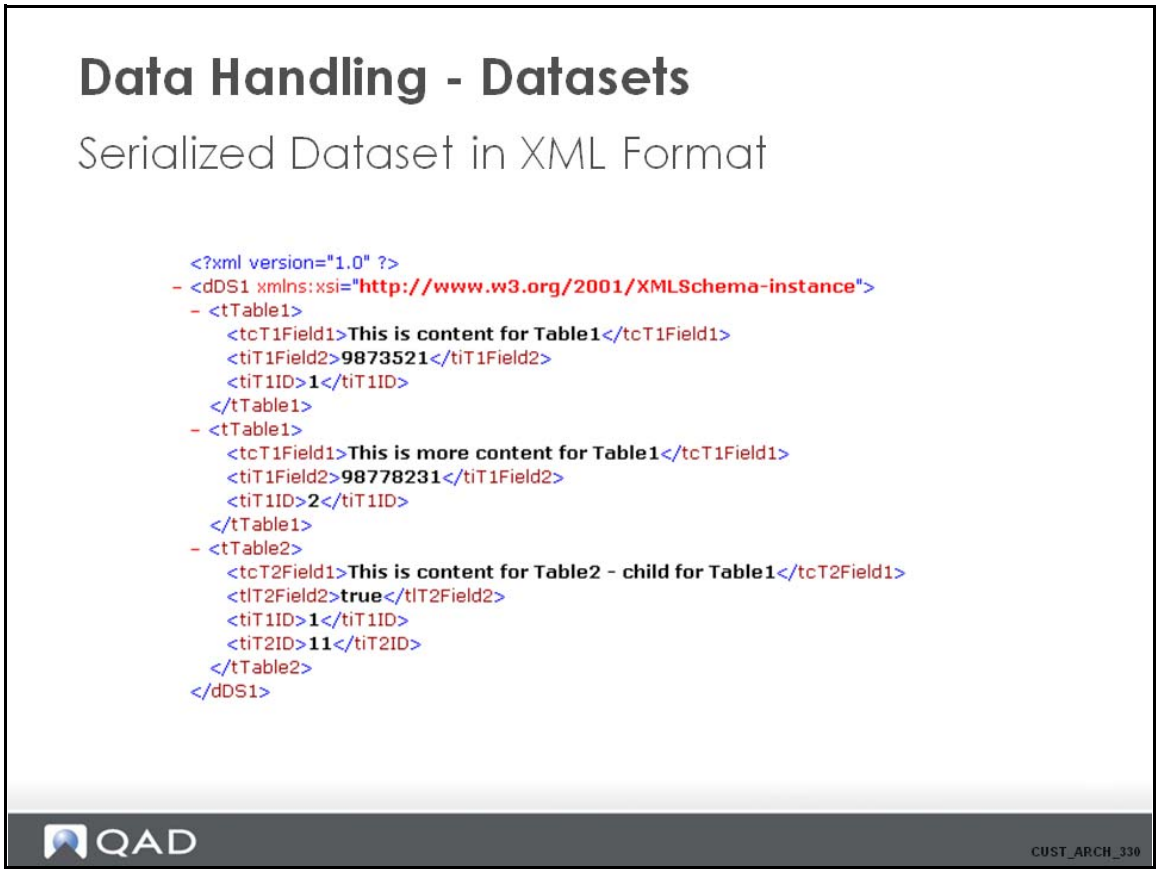
Data Handling - Datasets



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.

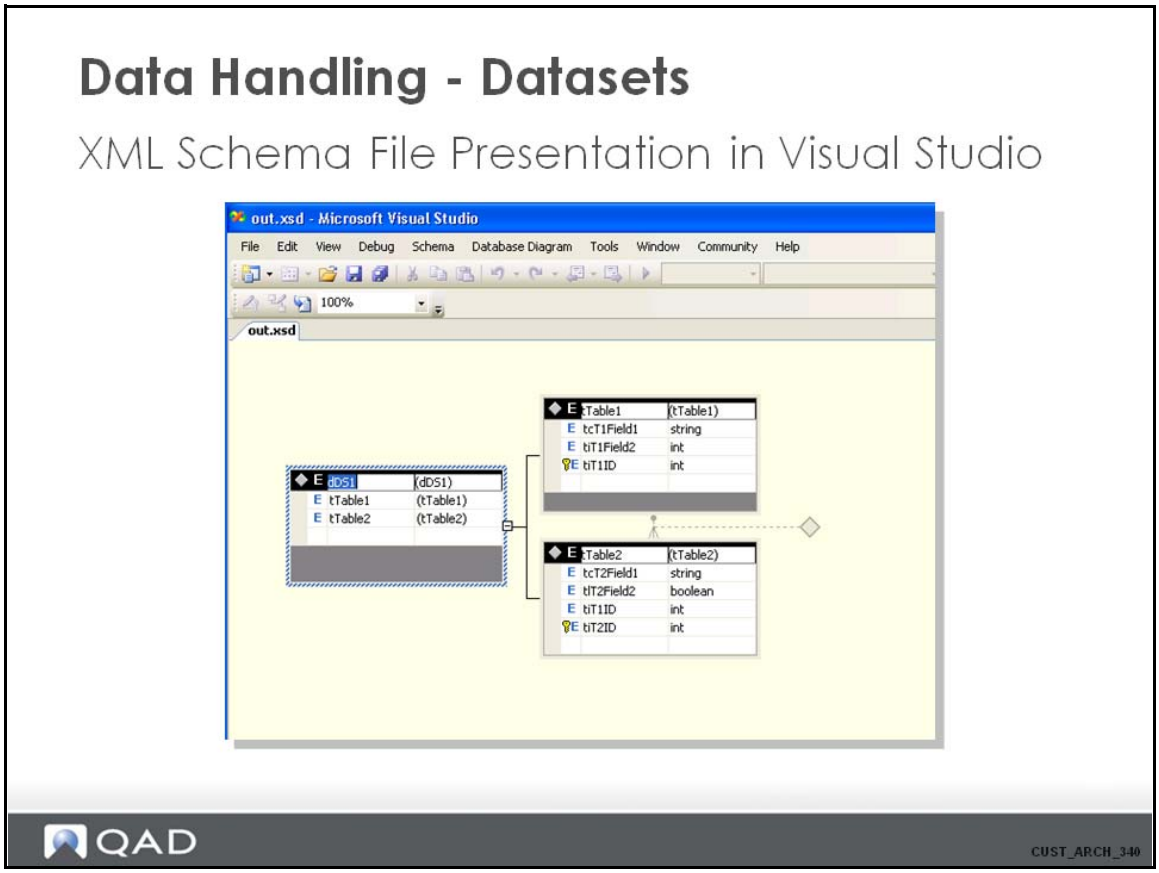
Data Handling - Datasets



The root element in the XML file is the dataset.

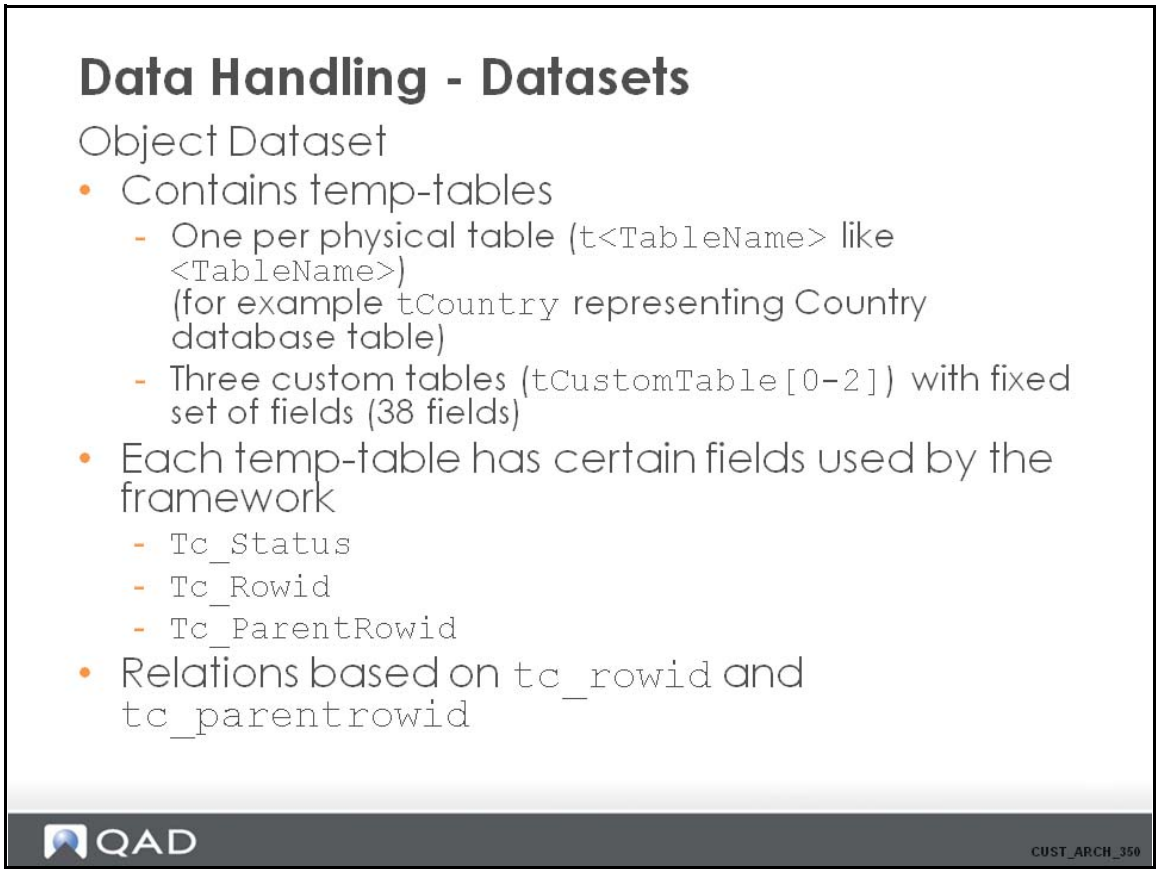
Every record is a sub-element of the dataset element, with the name of the table as the name of the element. An alternative hierarchical representation exists based on the data relations; in this case it would mean that the <tTable2> element would be a sub-element of <tTable1> (where <tiID1> = 1).

Data Handling - Datasets



Note the relationship between tTable1 and tTable2.

Data Handling - Datasets



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.

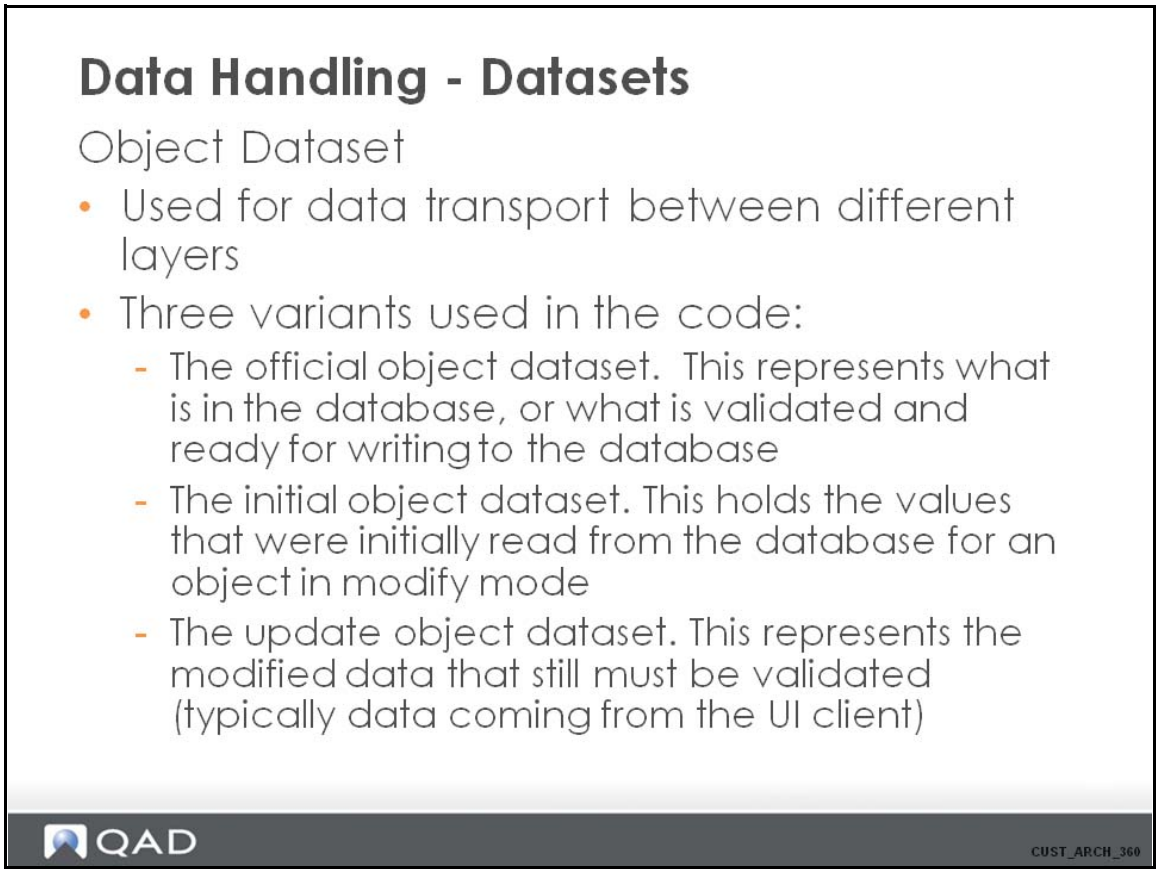
For customization purposes, each object dataset also contains three custom tables that customization developers can use to extend the data in the object datasets. They contain a fixed set of fields that the framework recognizes and can be filled in by the customization developer.

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



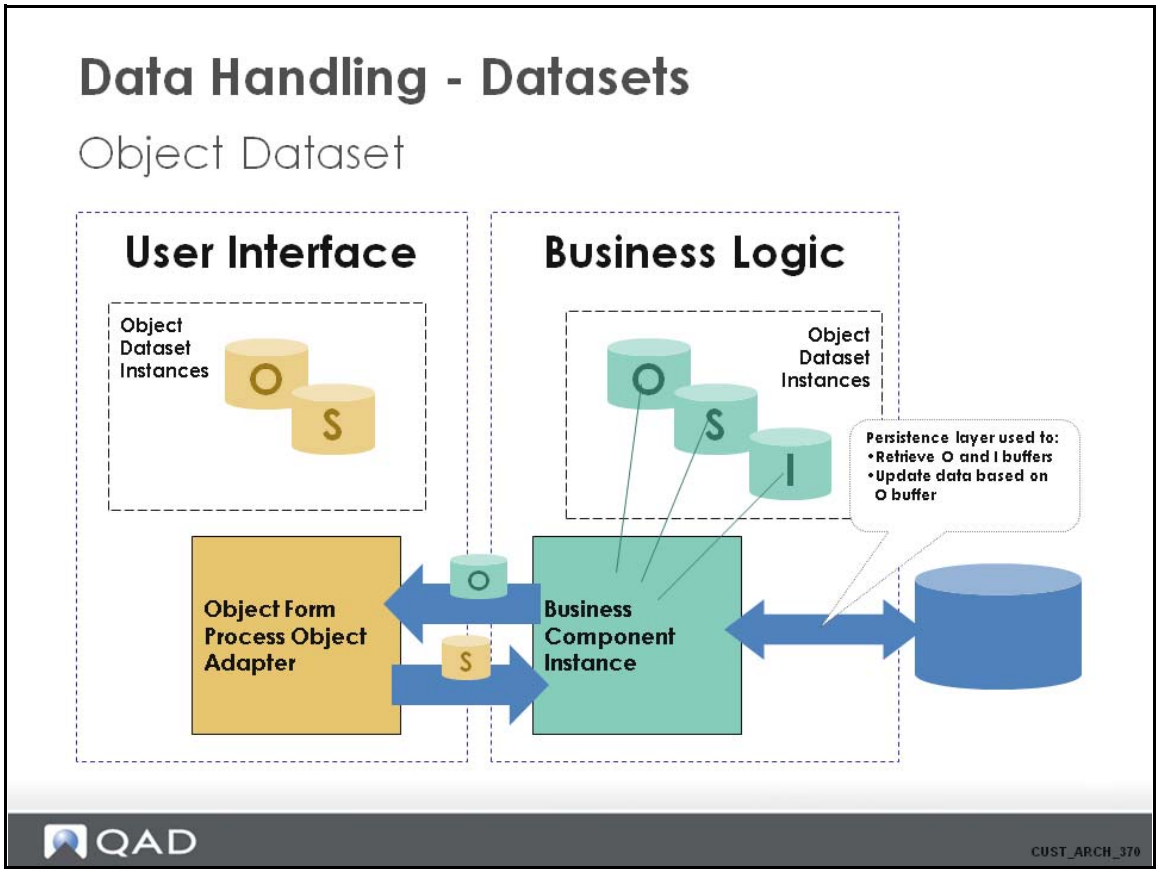
The object dataset is used to pass data between the layers. The business logic layer never directly uses database tables. The data is always retrieved via the persistence layer, and stored in the object dataset. The data the client UI layer is working with is received from the BL backend in the form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



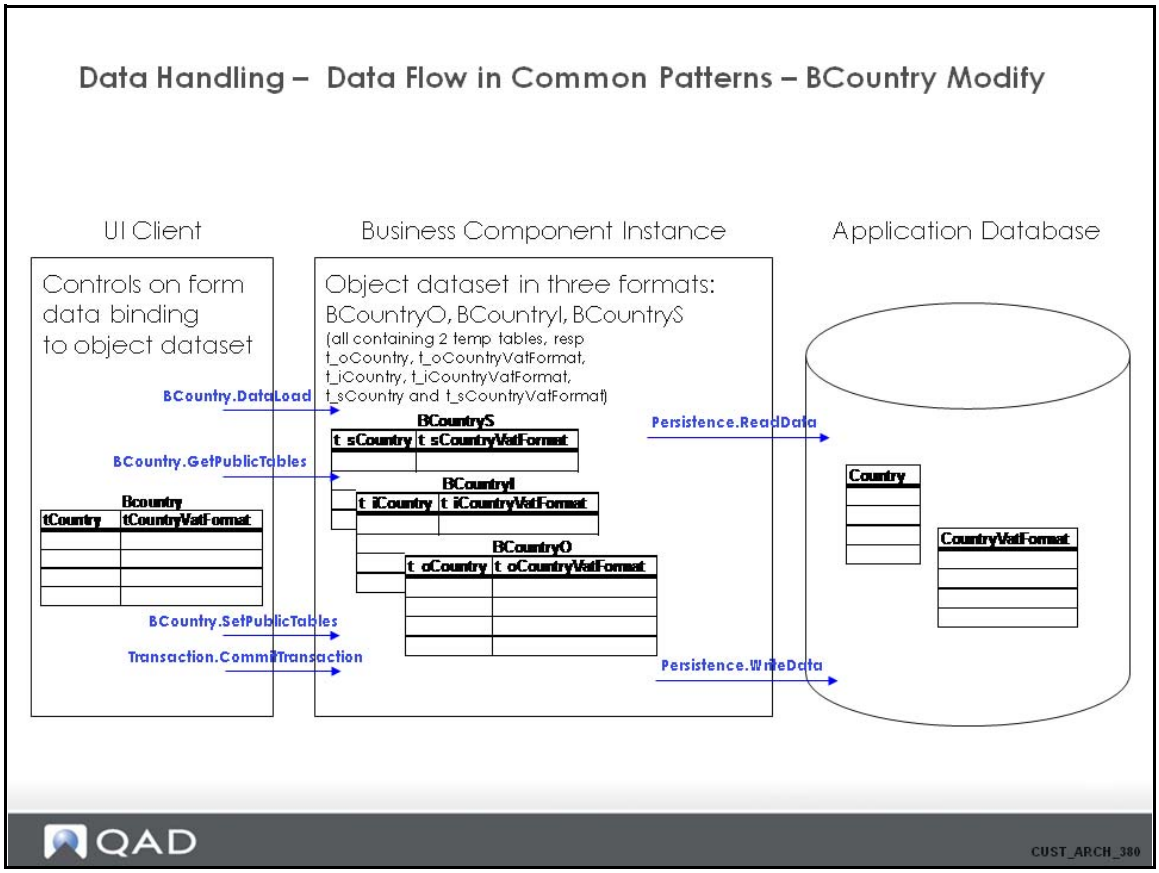
The business layer uses the object dataset instances to:

- 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)

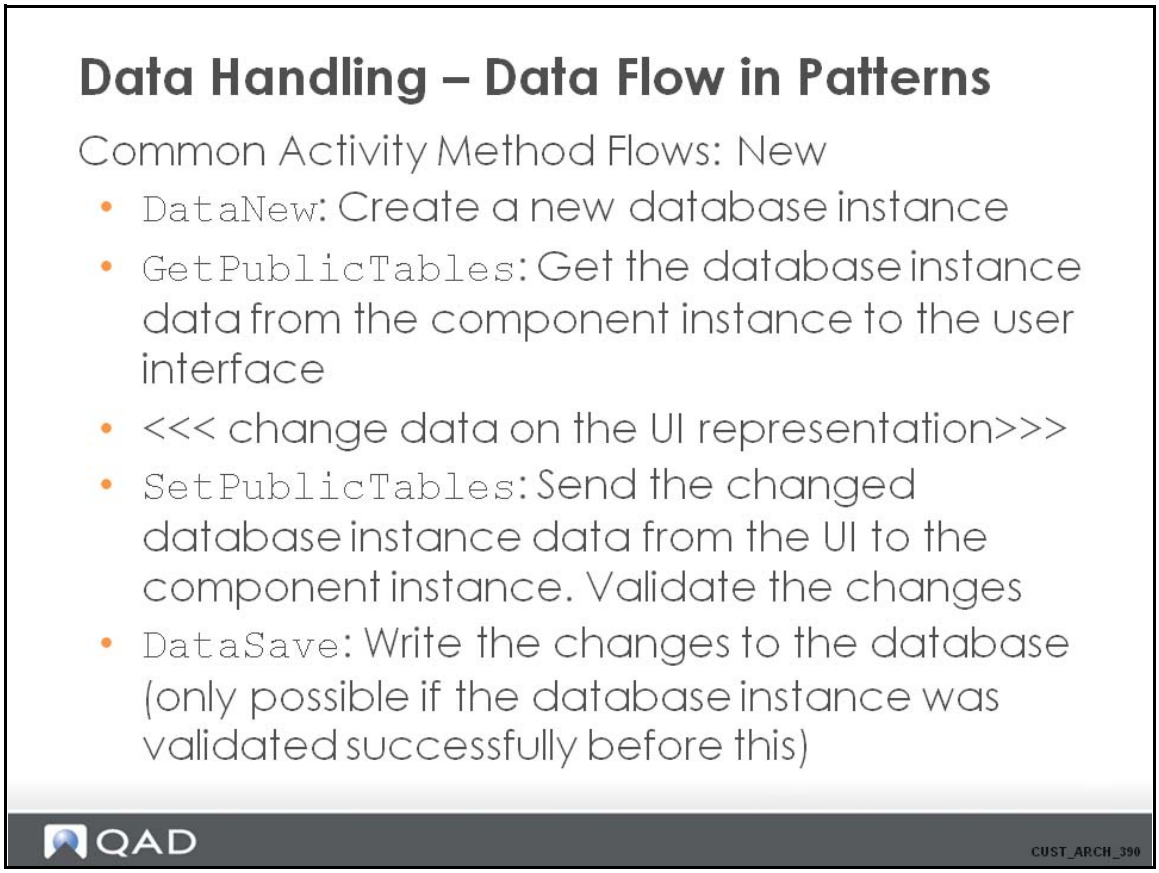
Data Handling - Data Flow In Common Patterns



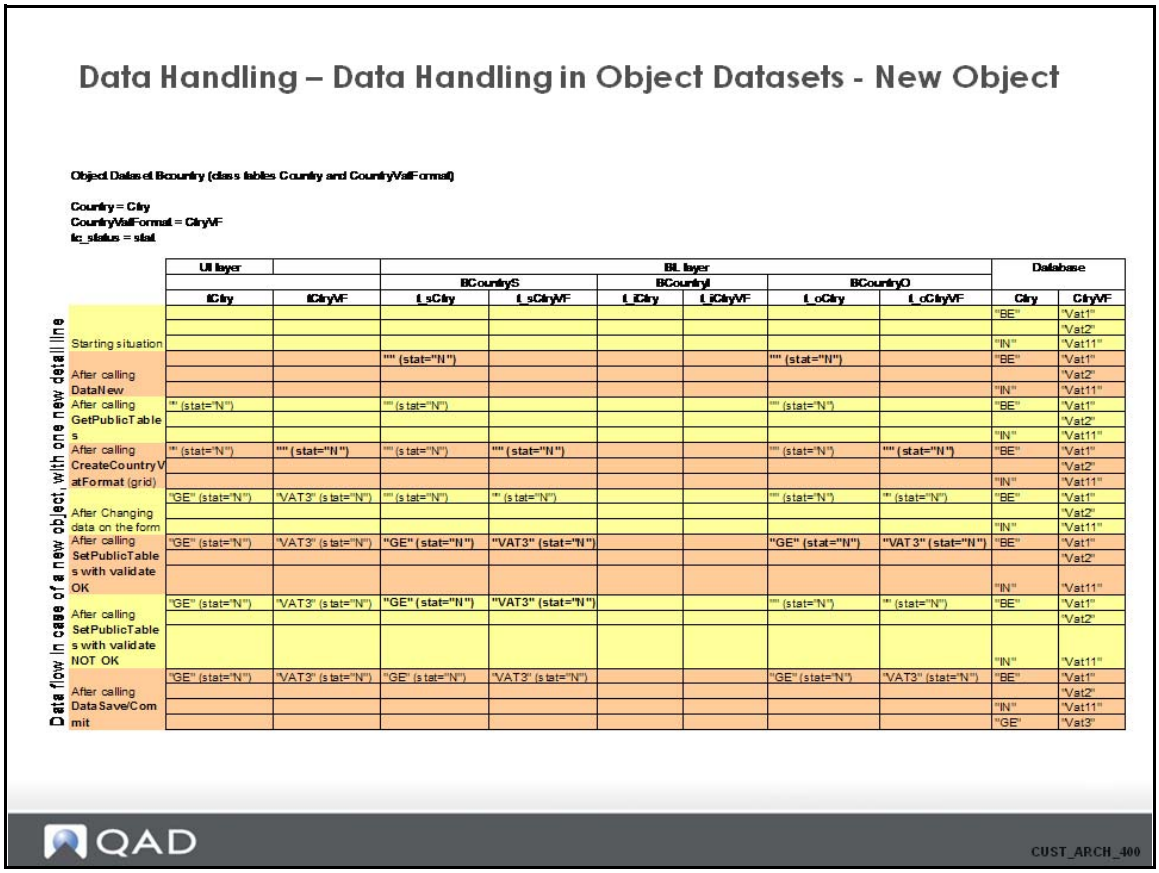
- BCountry is a business component that inherits from the “Database” component. This ensures that the component encapsulates all data access to the tables it is associated with. BCountry is responsible for tables Country and CountryVatFormat, for which a 1-N primary relation is defined in the datamodel.
- 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



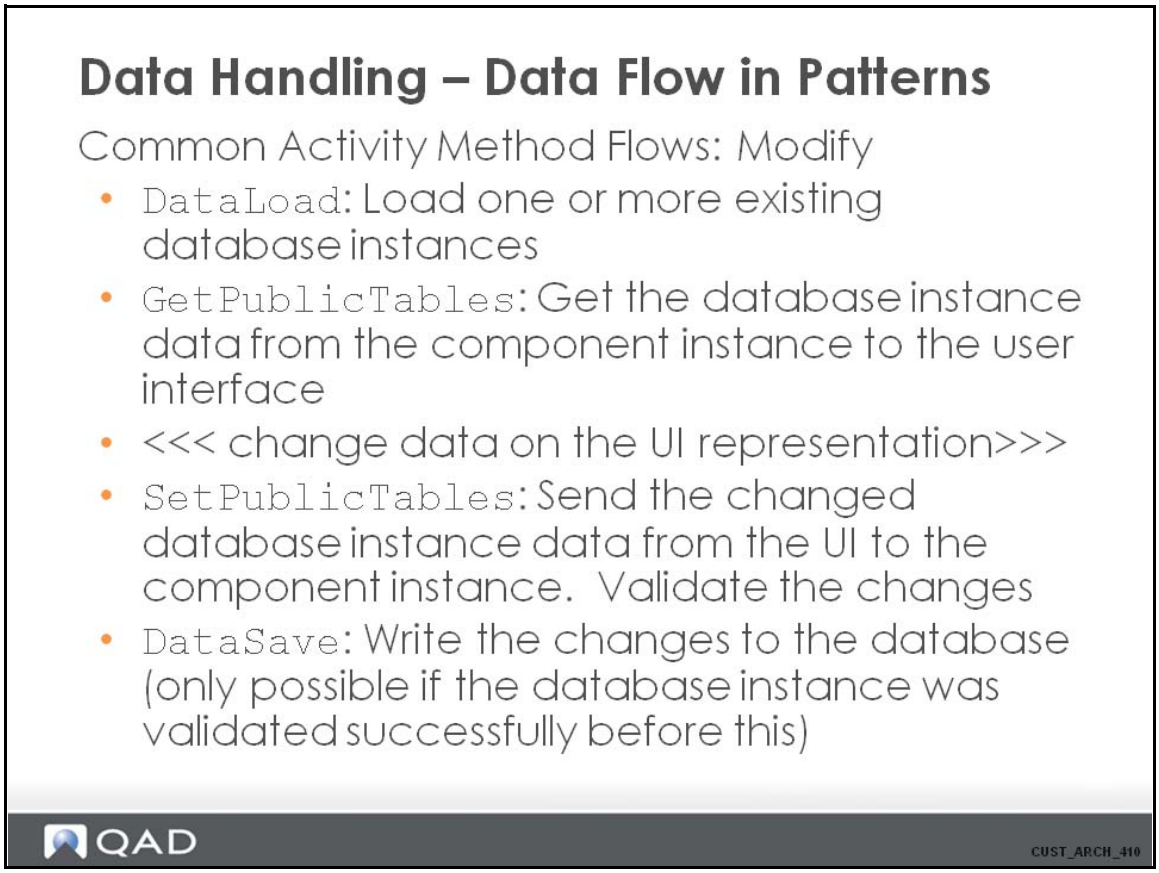
Data Handling - Data Handling in Object Datasets - New Object



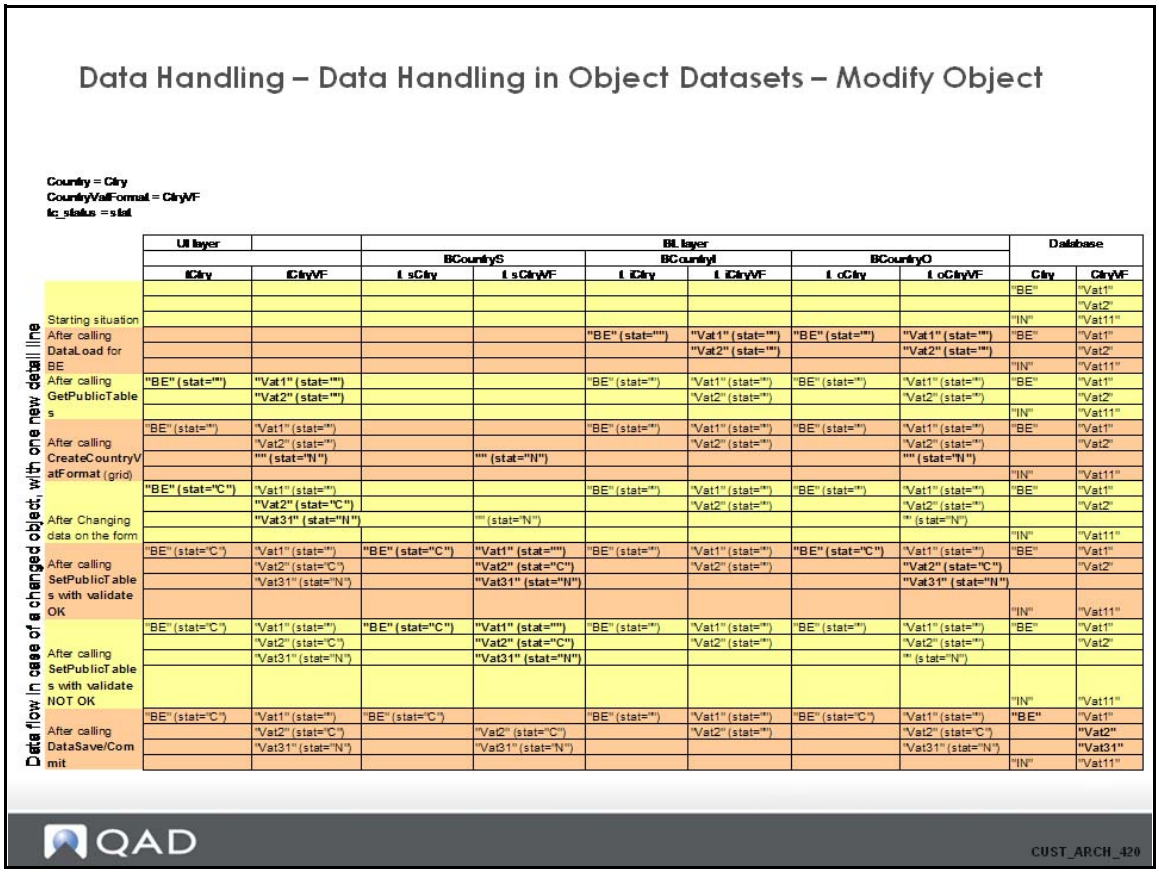
Notes:

- The BCountryI dataset is not used in this case, because optimistic locking is not needed for a newly created record.
- DataNew() makes sure a new record in the t_sCountry and t_oCountry is created. Tc_status is always “N”, and tc_rowid is a negative number, starting with “-1” for the first created record. DataNew() only creates records for the main class table, not for the child tables. So in this example, DataNew() does not create a record in t_sCountryVatFormat or in t_oCountryVatFormat.
- For each of the child tables (in this case on tCountryVatFormat), a separate method “Create<tableName>” exists. The method is used to create and initialize a new record in that table and transfer it immediately to the client. Underlying this, a call to the method AddDetailLine() is made. The name of the child table is supplied as a parameter.
- Transaction.CommitTransaction or BCountry.DataSave (depending on whether the component instance was attached to a logical transaction). If only the flag indicating a successful validation is switched on, the data is written to the database.
- Within the dataset, relations between the tables are based on tc_parentrowid = tc_rowid condition. In this example: tCountryVatFormat.tc_parentrowid = tCountry.tc_rowid.

Data Handling - Data Flow in Patterns

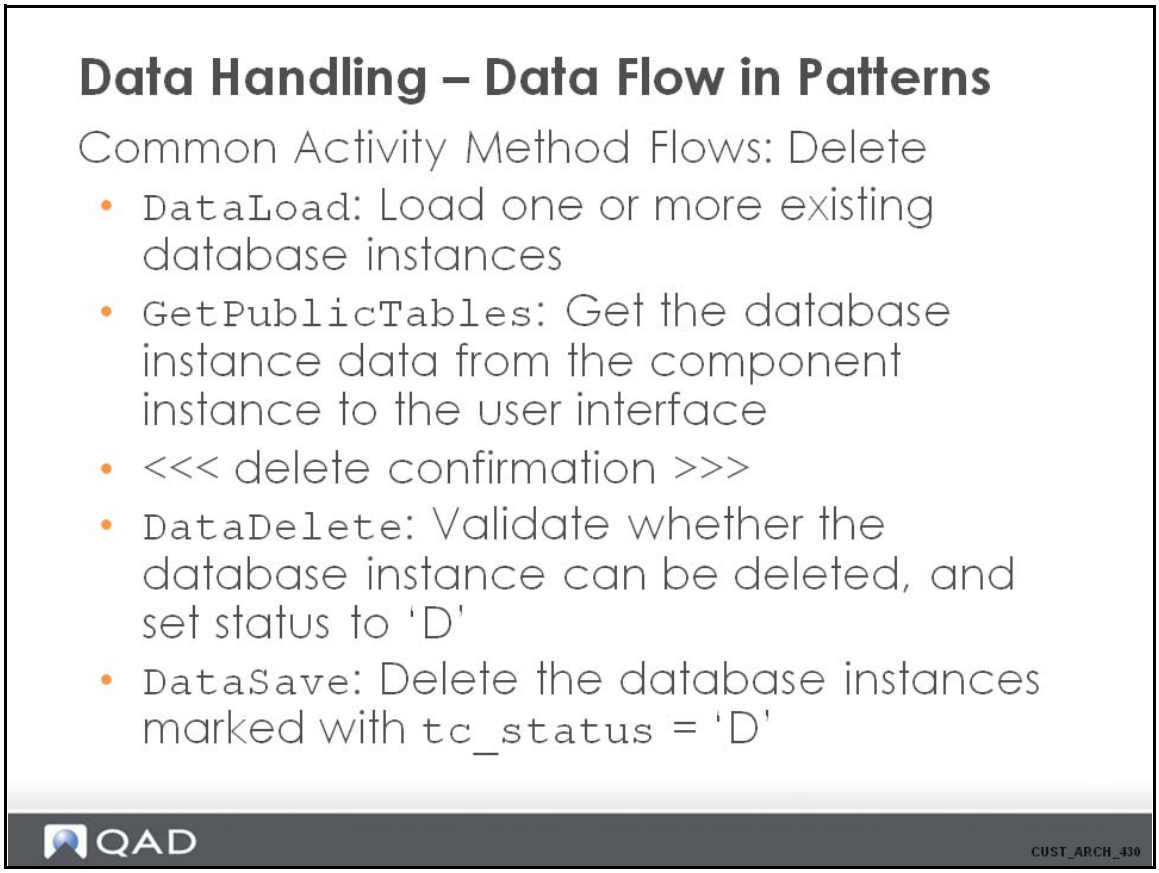


Data Handling - Data Handling in Object Datasets - Modify Object



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 Handling in Object Datasets - Delete Objects

Data Handling – Data Handling in Object Datasets – Delete Object

Object Dataset Bcountry (class tables Country and CountryVatFormat)

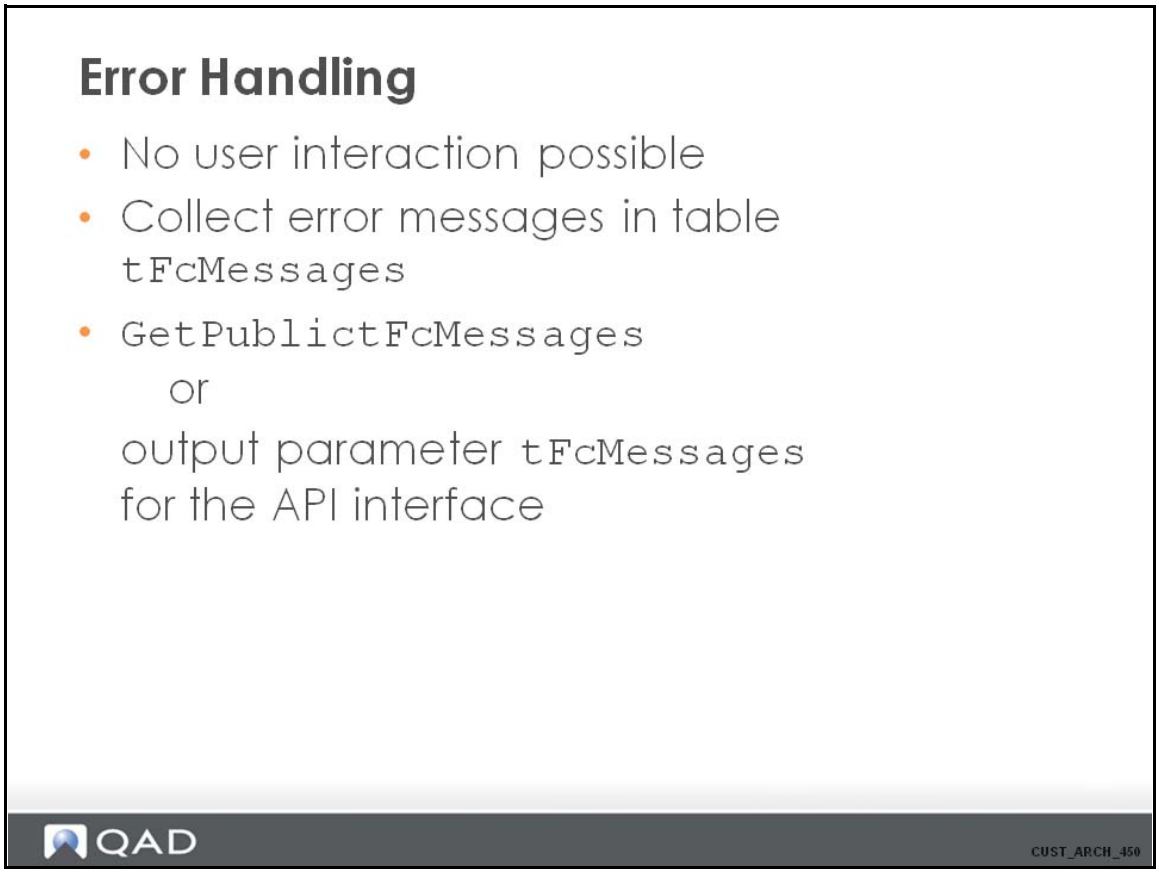
Country = Ctry
CountryVatFormat = CtryVF
tc_status = stat

Data flow in case of a deleted object (Country "BE")

	UI layer		BL layer						Database	
			BCountryS		BCountryI		BCountryO			
	tCtry	tCtryVF	t_sCtry	t_sCtryVF	t_iCtry	t_iCtryVF	t_oCtry	t_oCtryVF	Ctry	CtryVF
									"BE"	"Vat1"
										"Vat2"
Starting situation									"IN"	"Vat11"
After calling **DataLoad**or BE					**"BE" (stat="")**	**"Vat1" (stat="")**	**"BE" (stat="")**	**"Vat1" (stat="")**	"BE"	"Vat1"
						"Vat2" (stat="")		**"Vat2" (stat="")**		"Vat2"
									"IN"	"Vat11"
After calling **GetPublicTable**	**"BE" (stat="")**	**"Vat1" (stat="")**			"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE"	"Vat1"
s		**"Vat2" (stat="")**				"Vat2" (stat="")		"Vat2" (stat="")		"Vat2"
									"IN"	"Vat11"
After clicking "Delete" button	**"BE" (stat="D")**	"Vat1" (stat="")			"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE"	"Vat1"
		"Vat2" (stat="")				"Vat2" (stat="")		"Vat2" (stat="")		"Vat2"
									"IN"	"Vat11"
After calling **SetPublicTable**	"BE" (stat="D")	"Vat1" (stat="")	**"BE" (stat="D")**	**"Vat1" (stat="")**	"BE" (stat="")	"Vat1" (stat="")	**"BE" (stat="D")**	"Vat1" (stat="")	"BE"	"Vat1"
s with validate		"Vat2" (stat="")		**"Vat2" (stat="")**		"Vat2" (stat="")		**"Vat2" (stat="")**		"Vat2"
OK									"IN"	"Vat11"
After calling **SetPublicTable**	"BE" (stat="D")	"Vat1" (stat="")	**"BE" (stat="D")**	**"Vat1" (stat="")**	"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="")	"Vat1" (stat="")	"BE"	"Vat1"
s with validate		"Vat2" (stat="")		**"Vat2" (stat="")**		"Vat2" (stat="")		"Vat2" (stat="")		"Vat2"
NOT OK									"IN"	"Vat11"
After calling **DataSave/Com**	"BE" (stat="D")	"Vat1" (stat="")	"BE" (stat="D")		"BE" (stat="")	"Vat1" (stat="")	"BE" (stat="D")	"Vat1" (stat="")	"IN"	"Vat11"
mit		"Vat2" (stat="")		"Vat2" (stat="")		"Vat2" (stat="")		"Vat2" (stat="")		

QAD CUST_ARCH_440

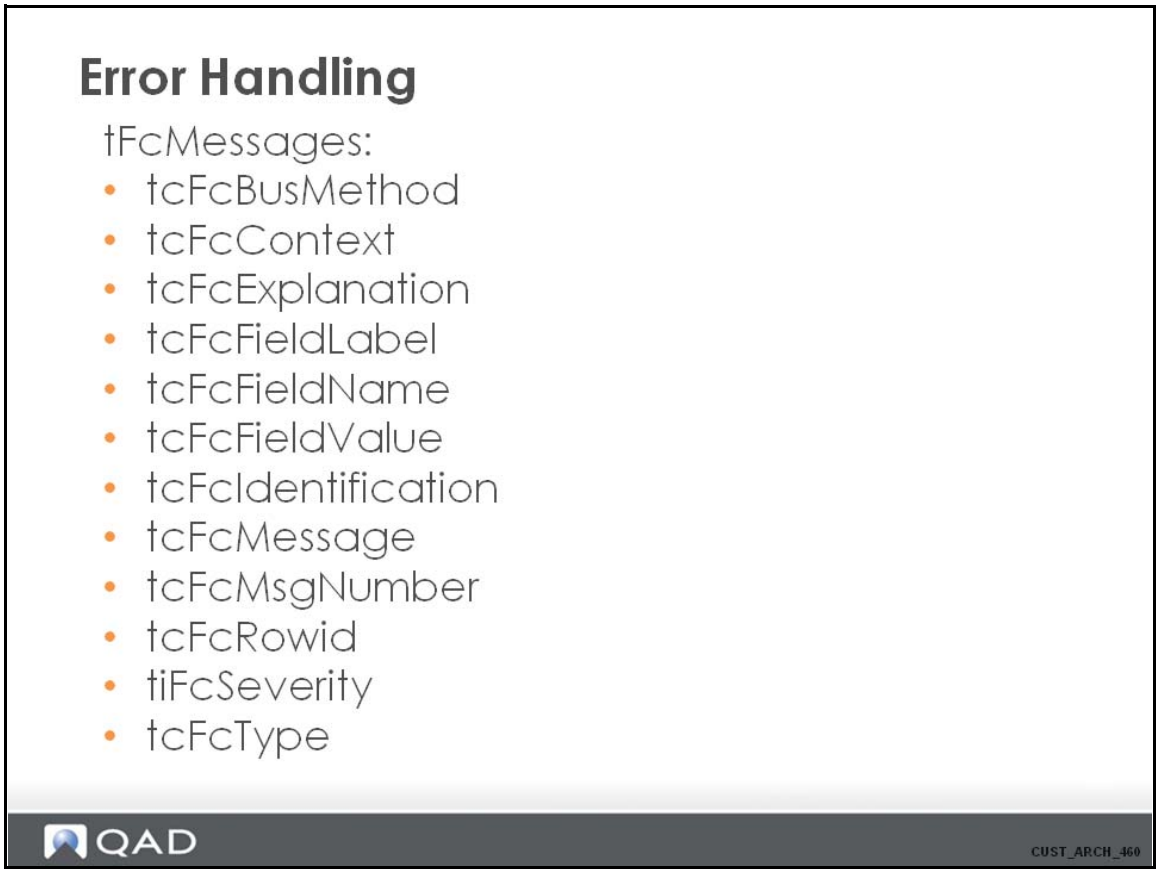
Error Handling



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



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

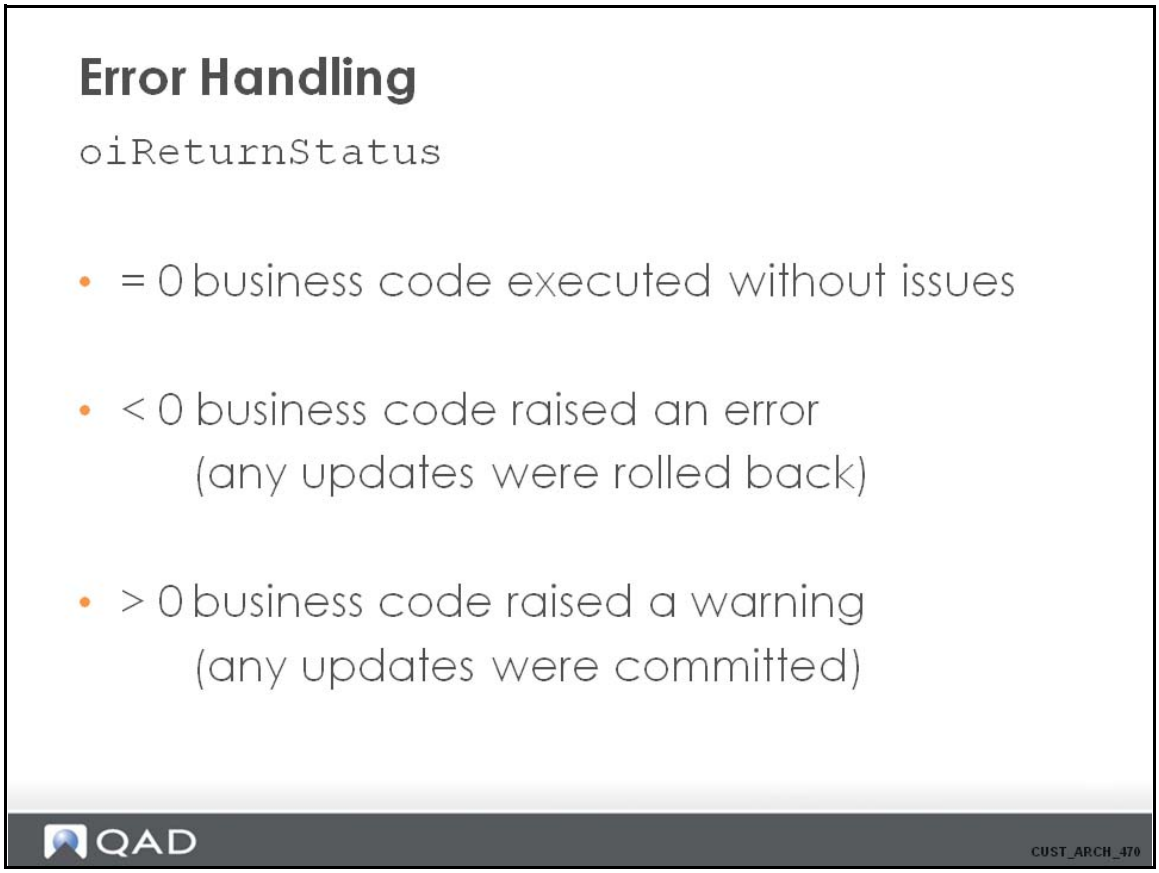
If the error is a validation error on data in the object dataset, these columns indicate what record and what column on that record contains an incorrect value.

tcFcContext

tcFcExplanation

tcFcIdentification

Error Handling



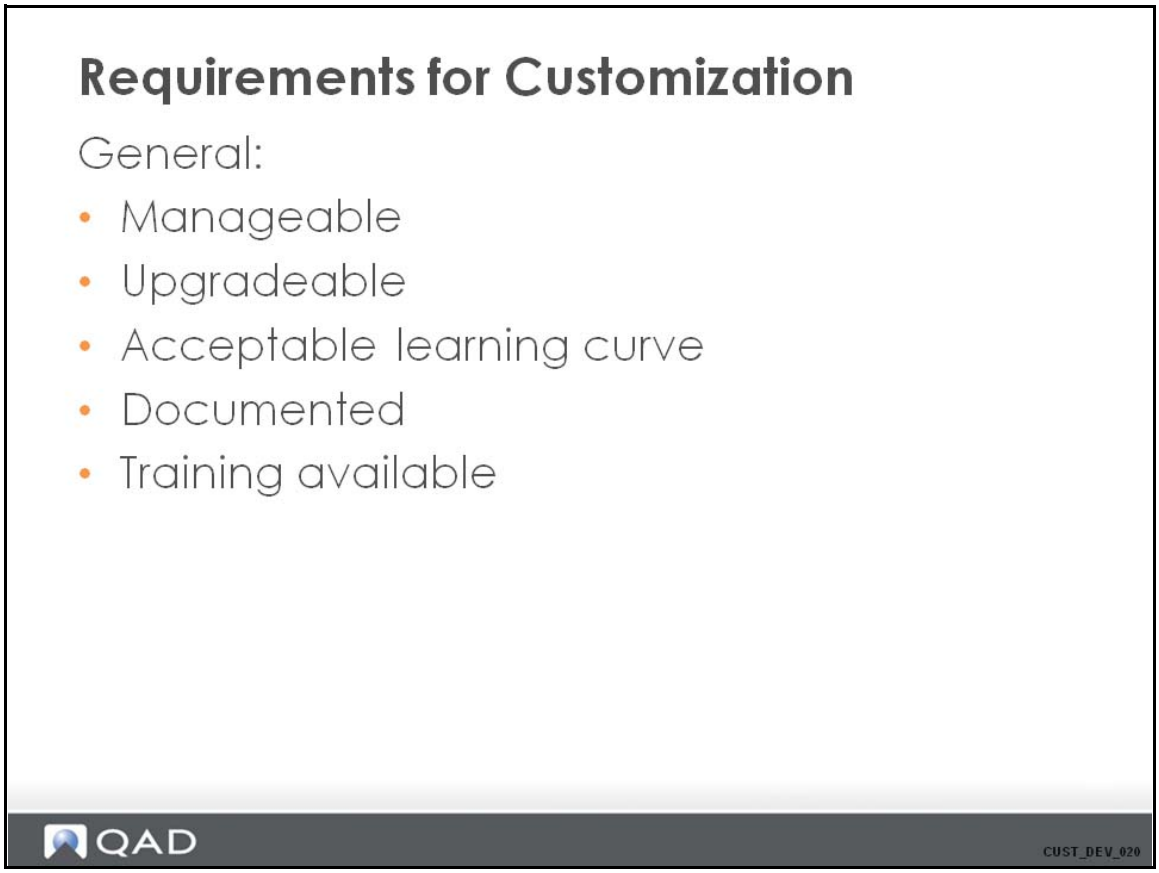
(Almost) every business method has an output parameter oiReturnStatus to indicate if the method executed correctly.

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

Requirements for Customization



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.

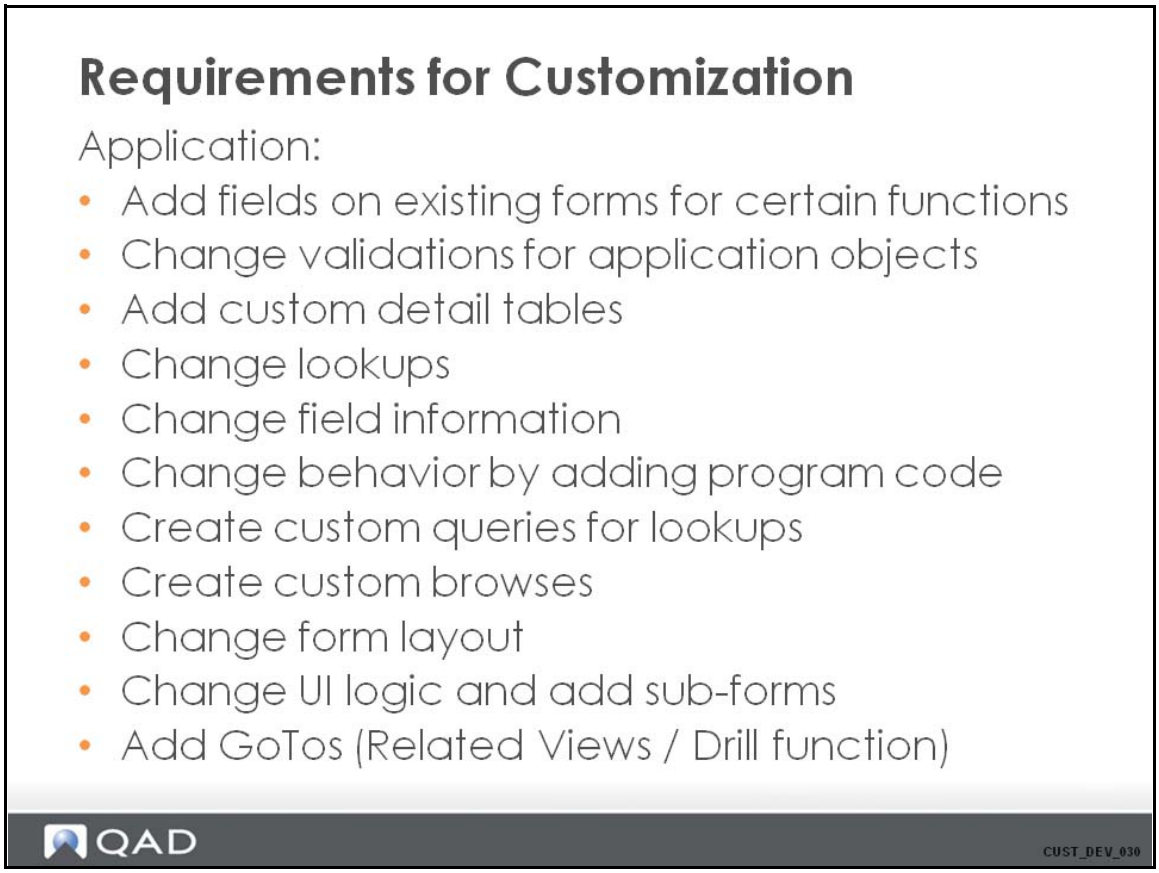
[G5] Documented

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



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

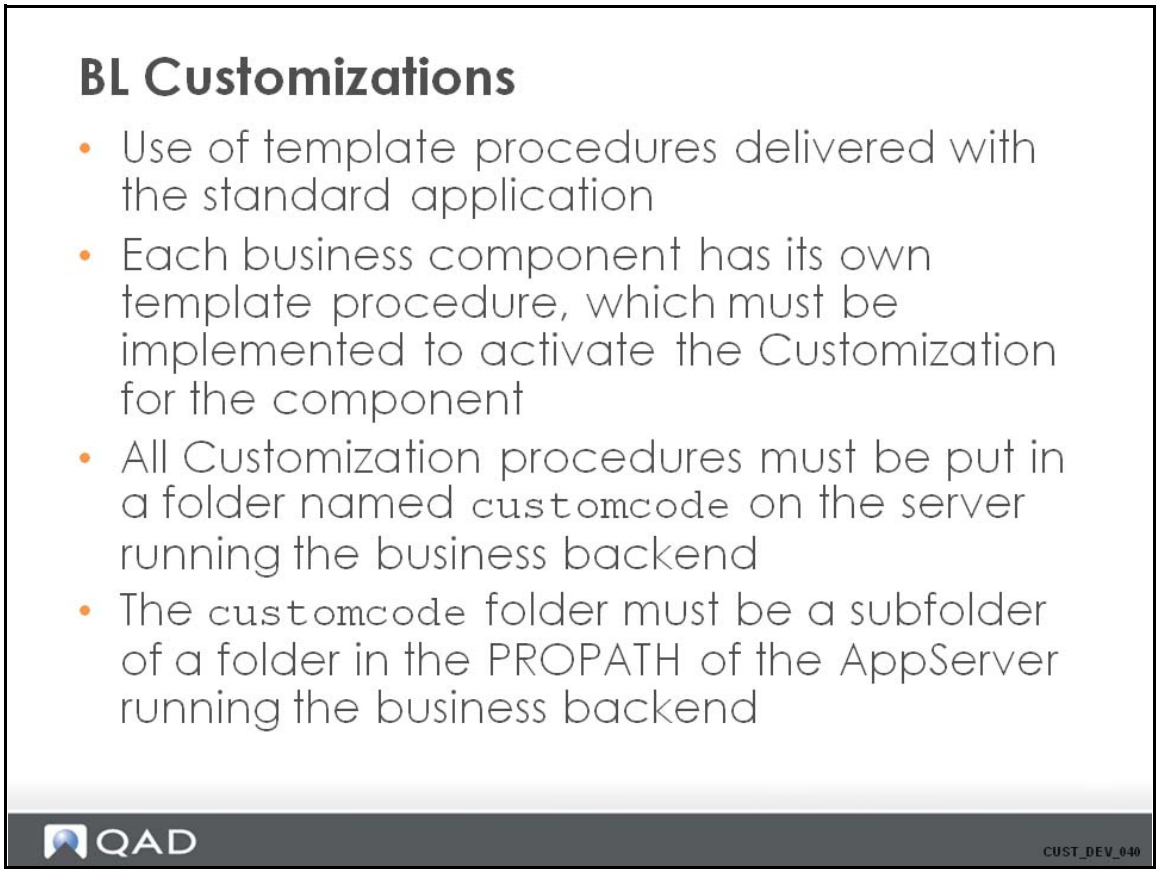
It should be possible to write extensions to the UI logic. The customization developer should be able to add code on certain events and add code before or after the normal code on the execution of predefined infrastructure methods.

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.

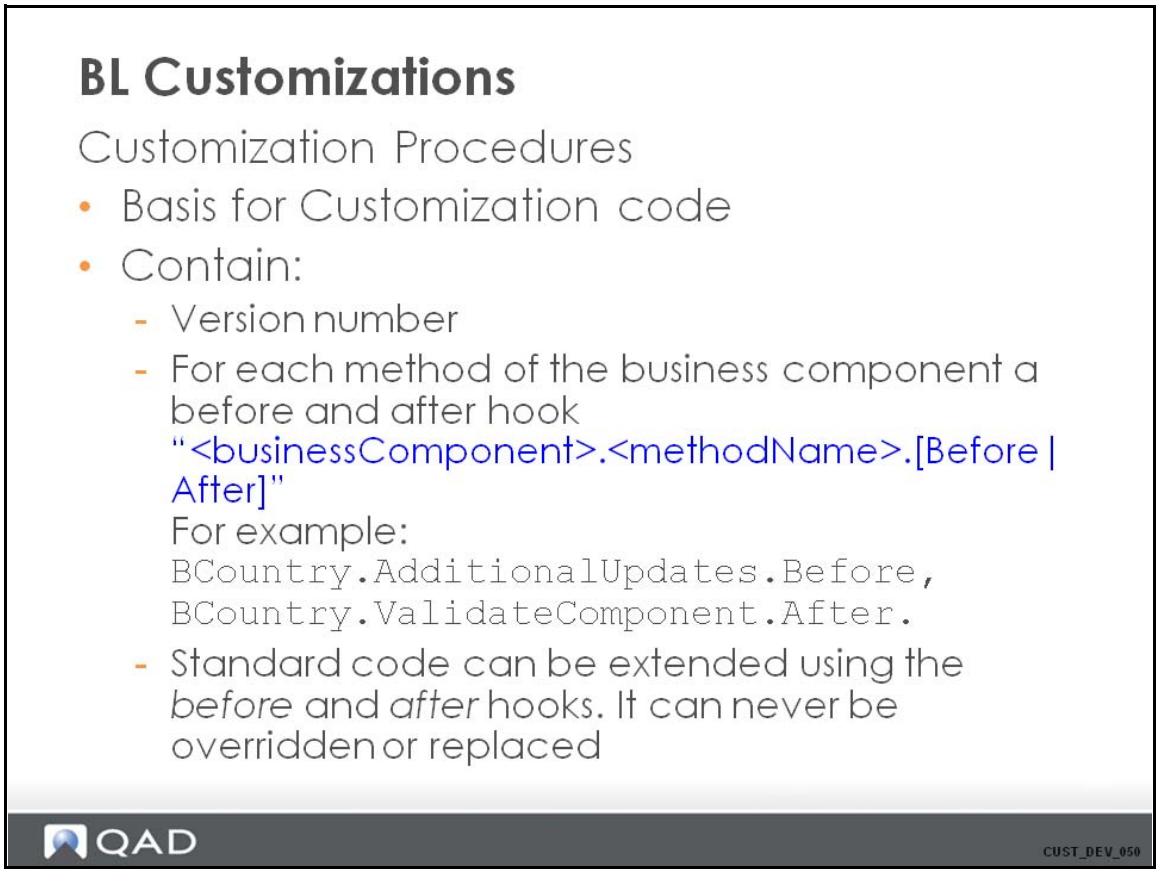
BL Customizations



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



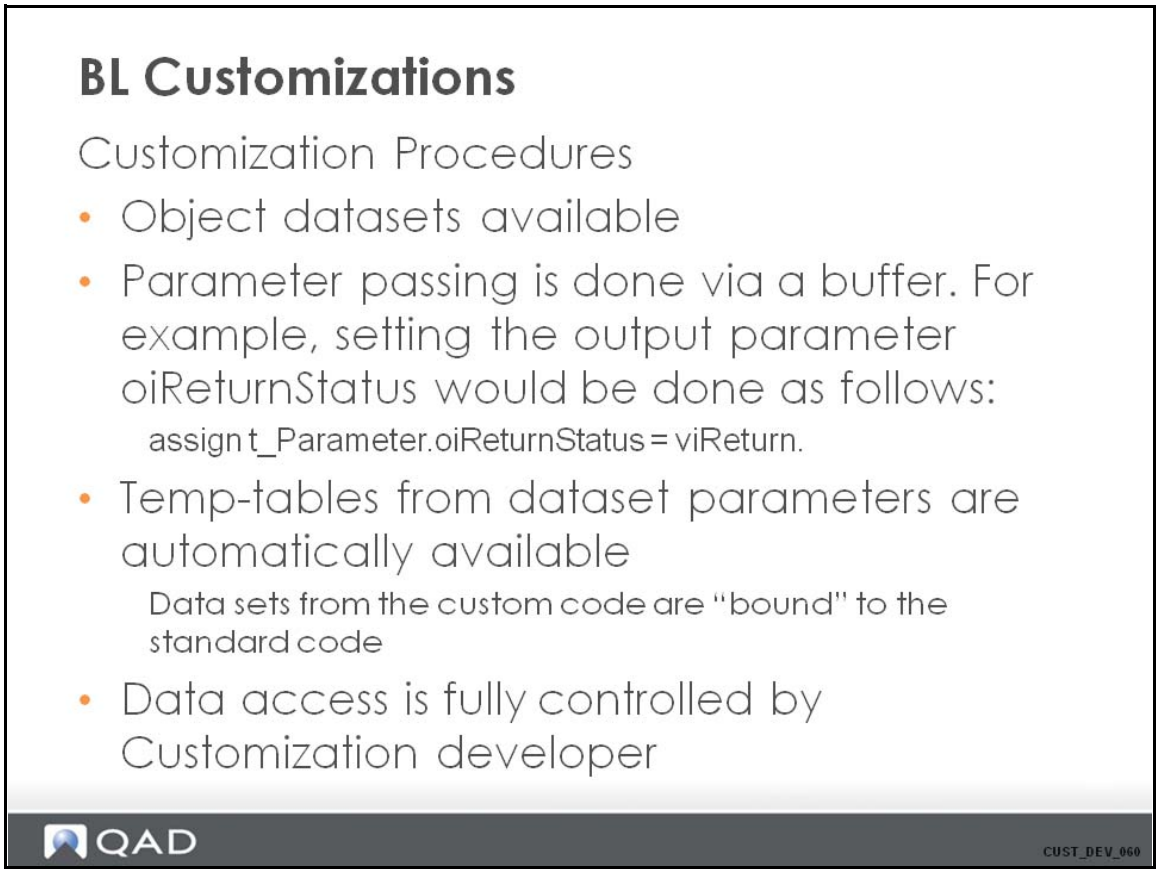
The name of the method and the business component can be used to navigate the HTML documentation to find the right documentation and source code for the standard implementation of the method.

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

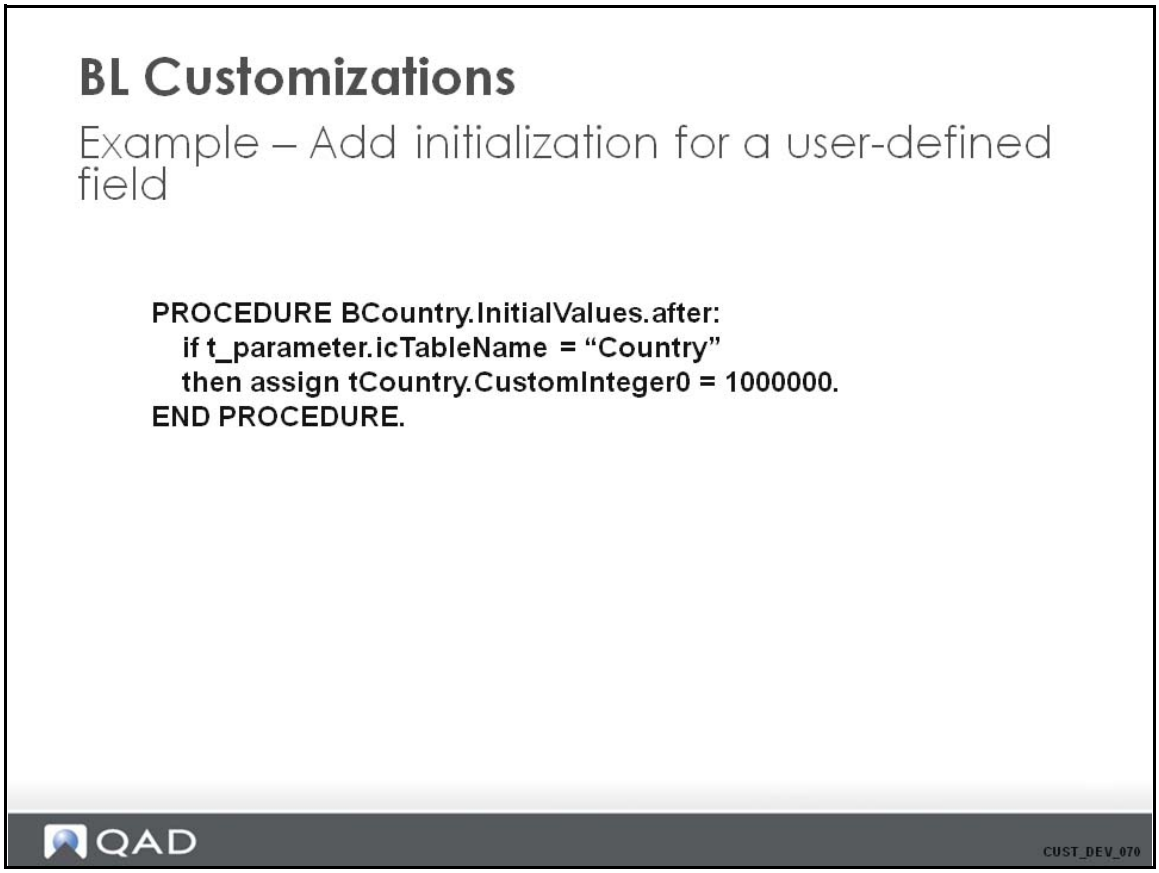


Data access in the standard component logic is always implemented using the persistence layer as Data Access layer. For the customization, we do not want to apply this rule because the interface to the persistence layer is not straightforward. For the standard code development within the QAD development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

Attention!

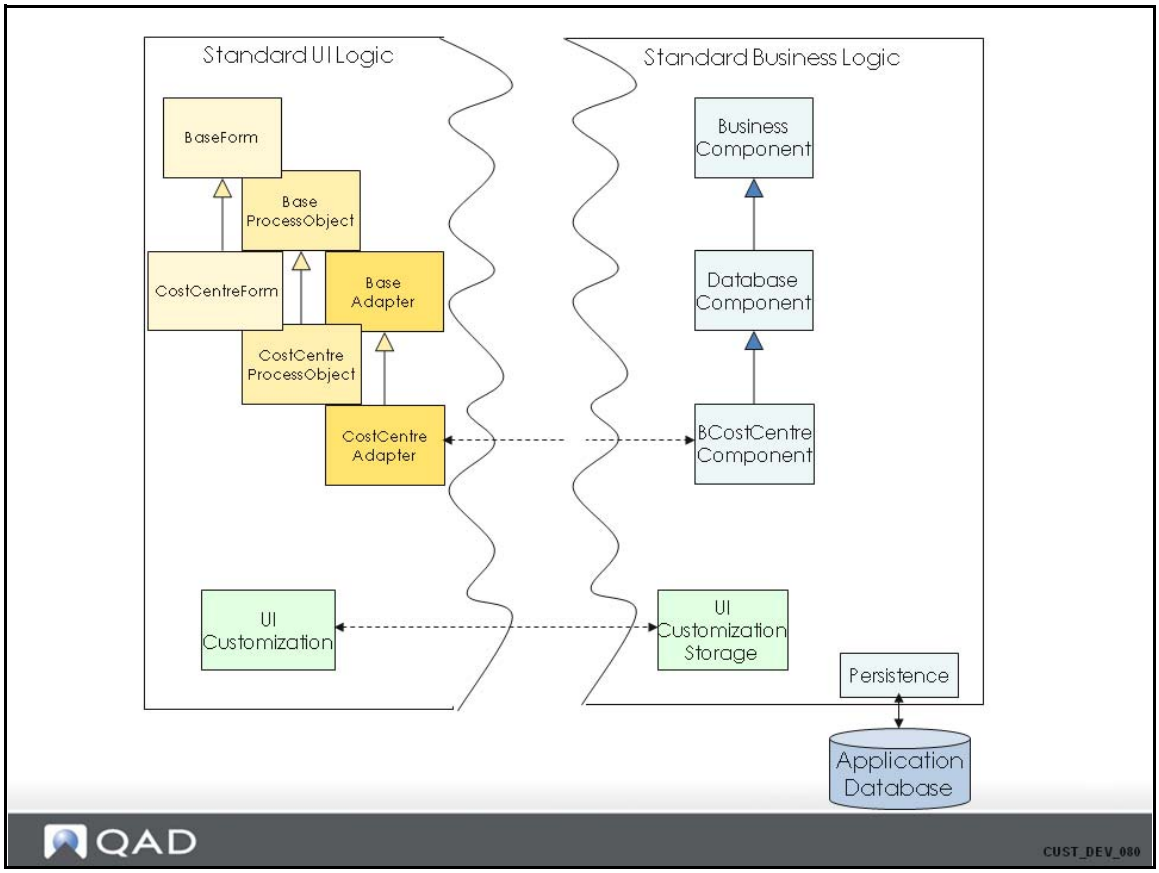
Since everything stays persistent in memory, care should be taken for record scopes. For example, if you write code to update a table in the PostSave.After event, since the transaction is not finished, an exclusive lock on a record is downgraded to a share lock. To avoid problems, it is best to use a 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

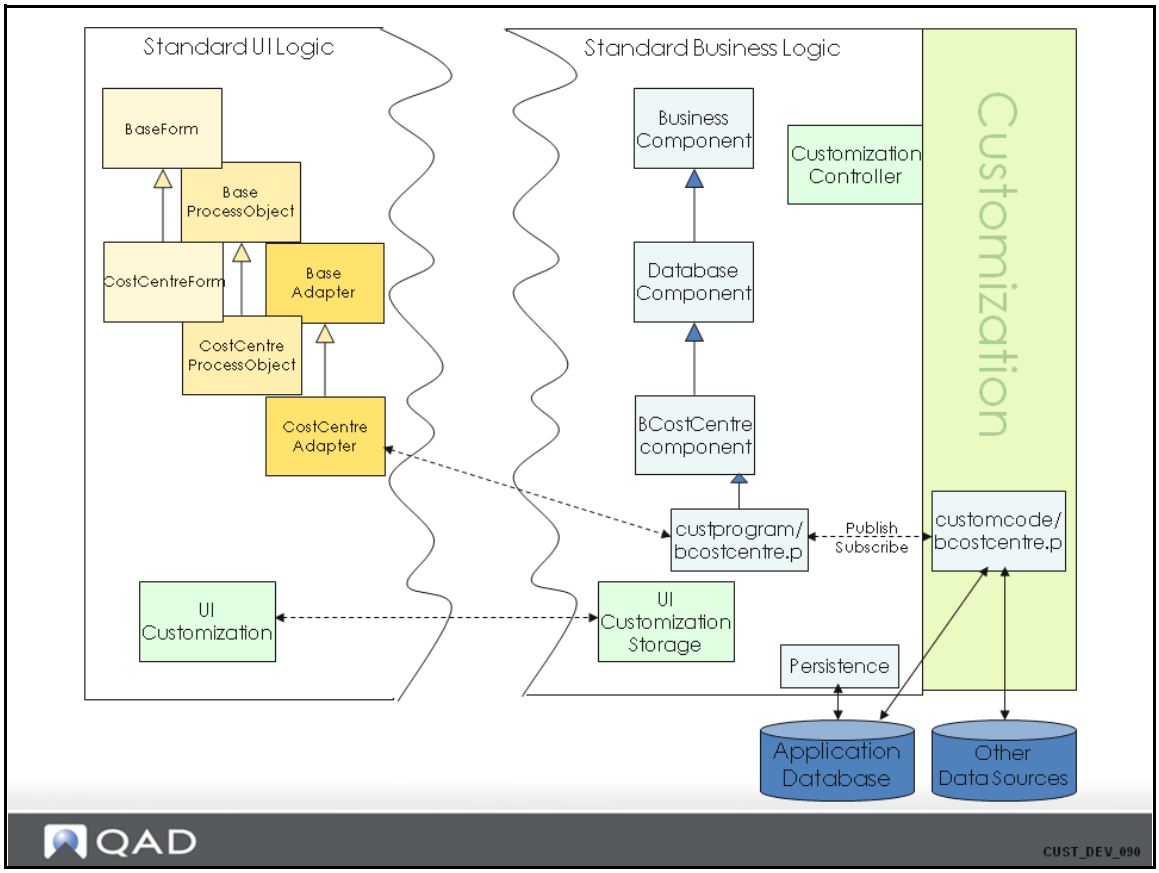


The object dataset is accessed through the temp-tables that make up the object dataset. In this example, we use the table tCountry. The newly created buffer is available, so no find/for each is required.

Customization Diagram



Customization Diagram

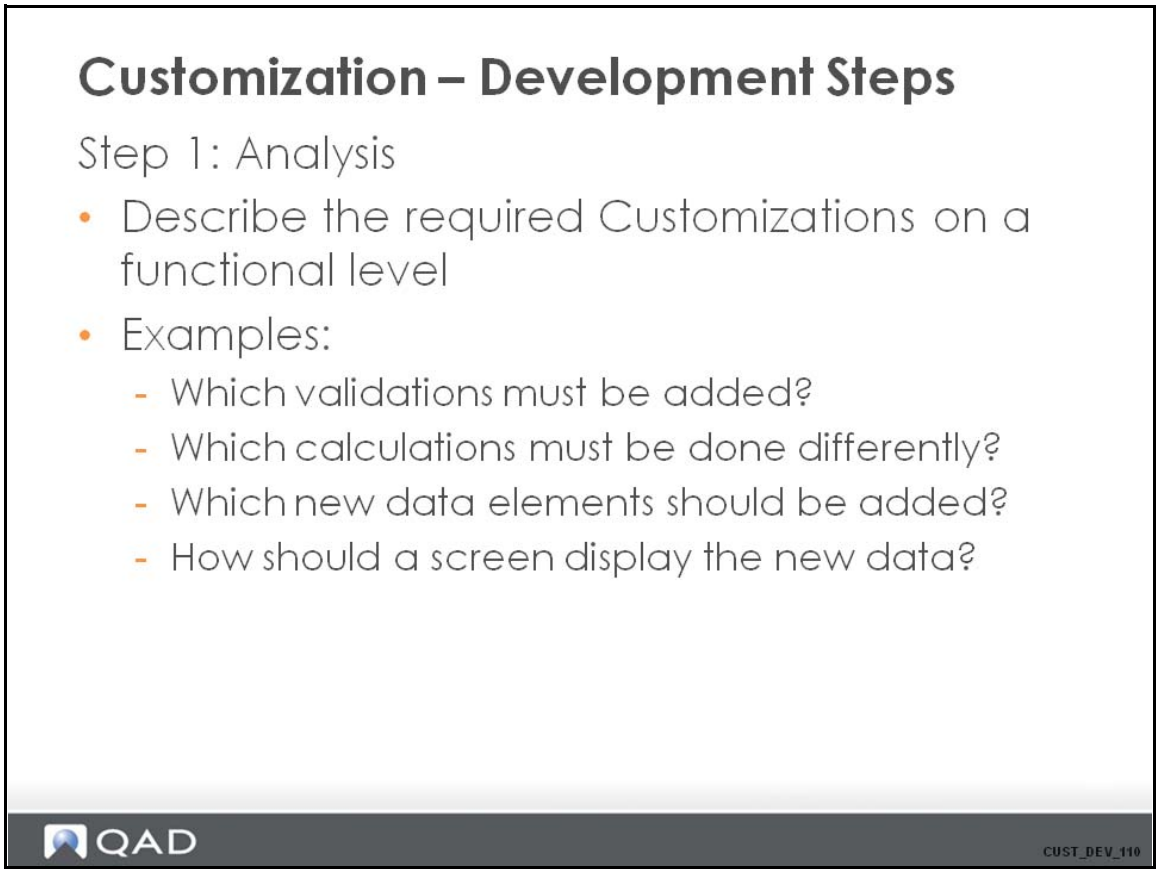


The ComponentPool session super-procedure starts the customization controller component. At startup (typically startup of the AppServer agent), it identifies all available customizations by searching in the PROPATH for a folder named “customcode”. It keeps in memory which components are customized (for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



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

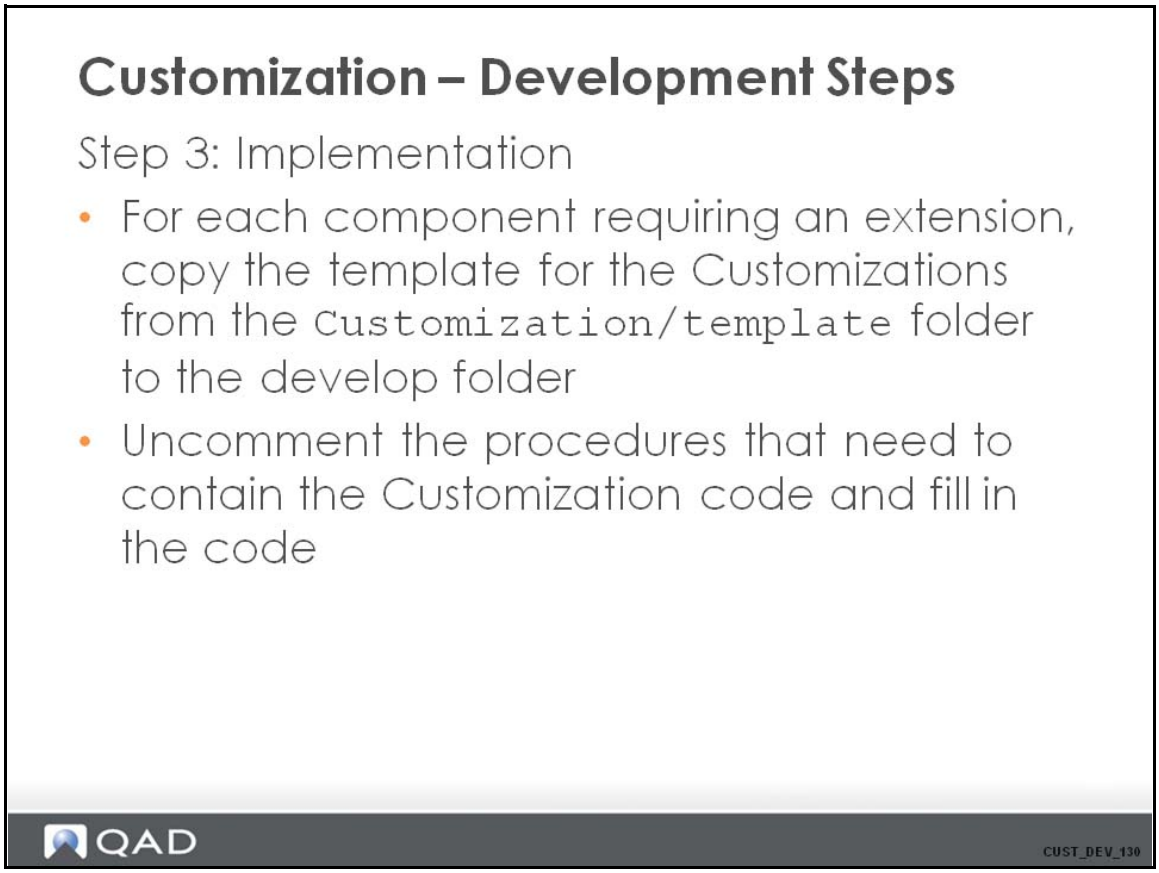
This can be a very high-level description and does not need to go into technical details. At this stage, the person making the analysis is purely focused on functionality and not on whether the customization is feasible with non-intrusive customization or requires intrusive customization.

Customization - Development Steps

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.

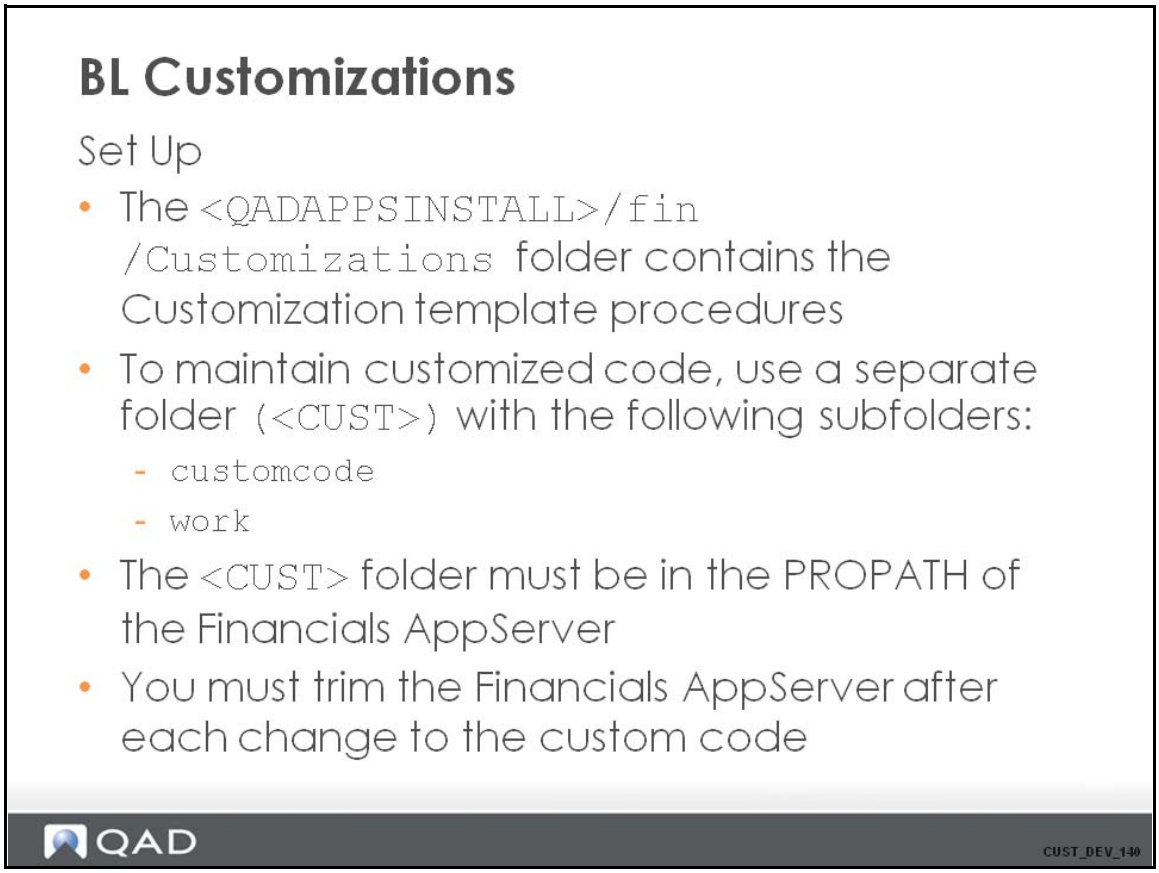
The design step is the most difficult. This requires comprehensive knowledge of the application, the data model, the application structure, and the patterns. In complex cases, QAD services might need to be involved to explain how the function works and how a certain customization can be achieved.

Customization - Development Steps



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

BL Customizations



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.

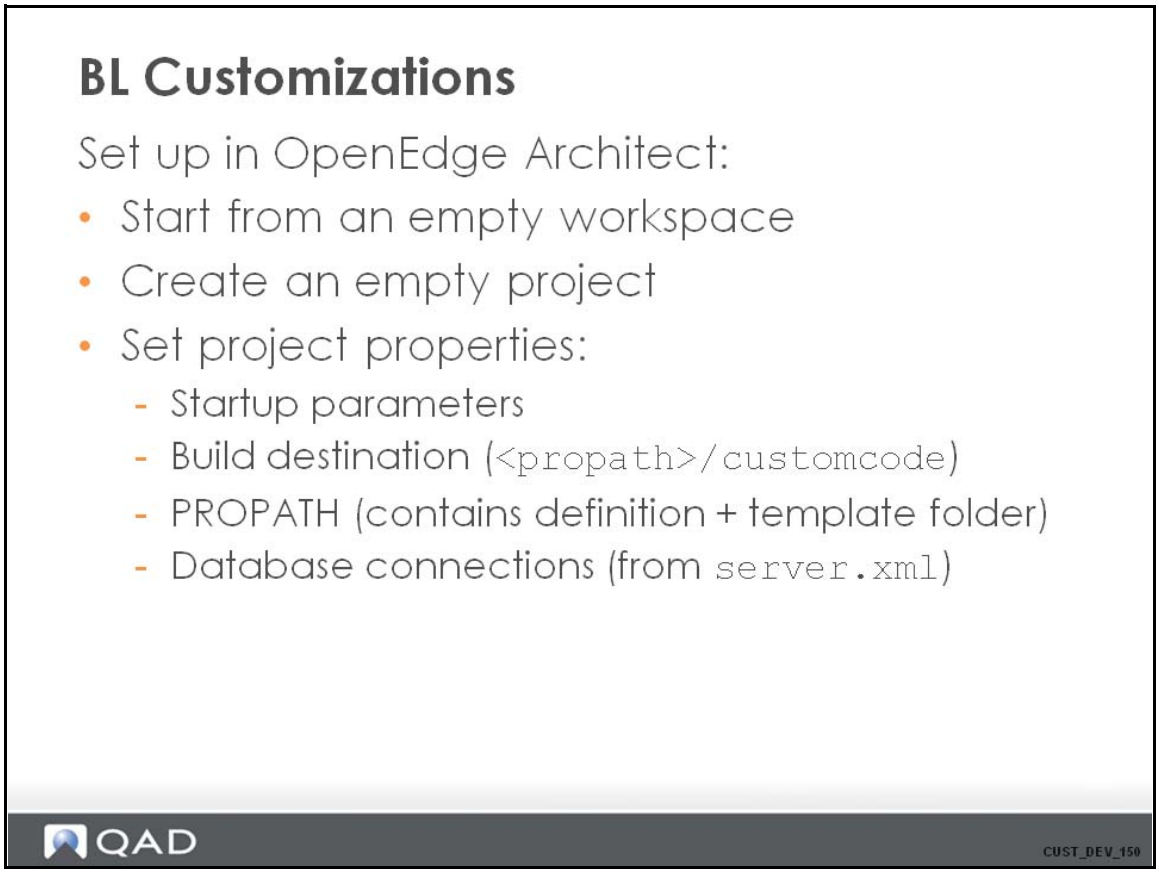
The changes to customization procedures only take effect after you trim the Financials AppServer.

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



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
```

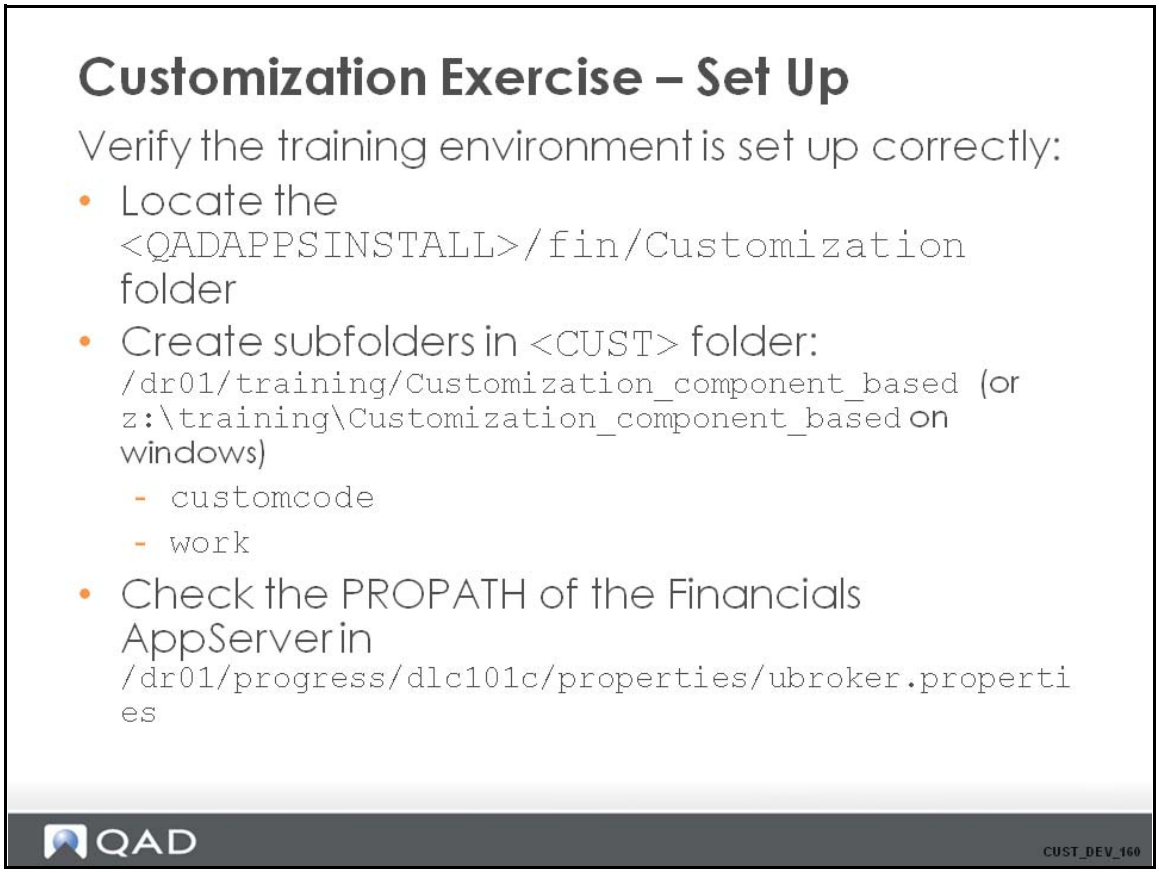
`PROPATH` (`PROPATH` for compilation does not match the PROPATH for runtime)

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

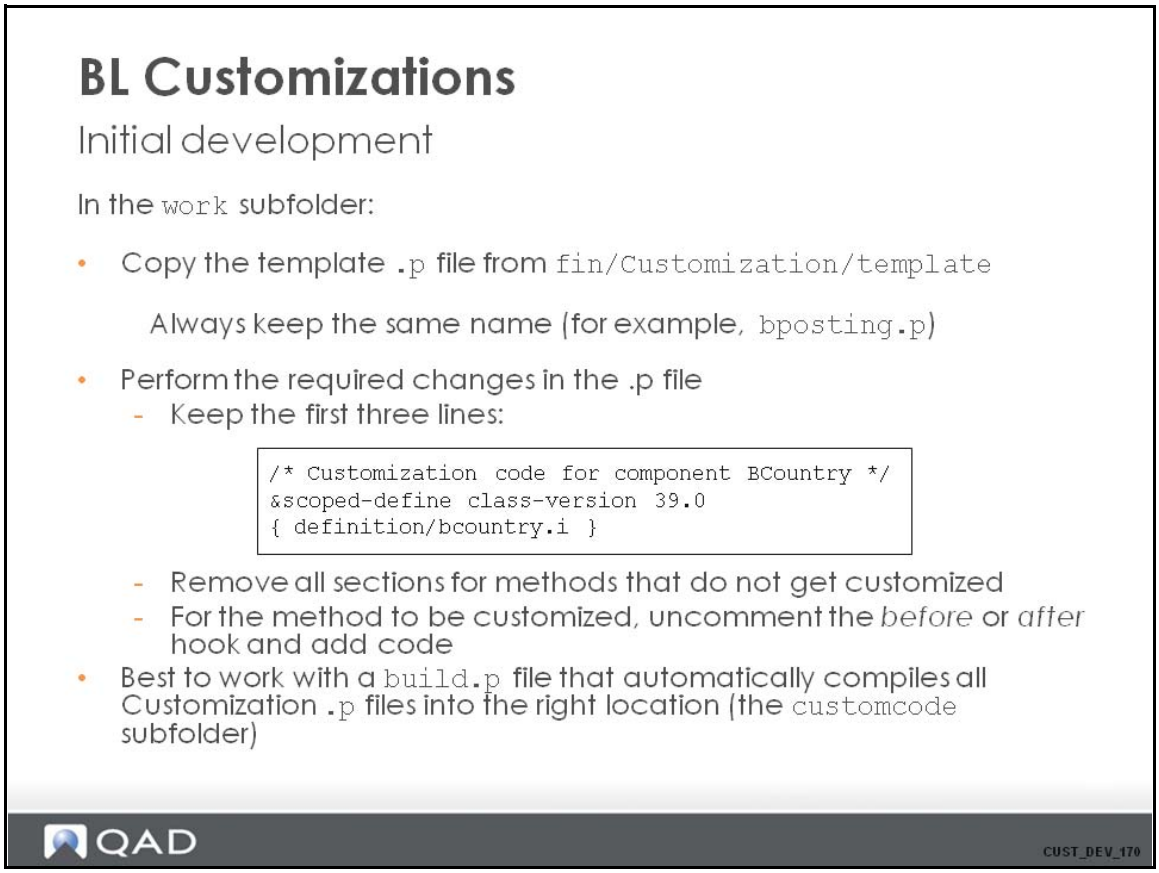
Database connections

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

Customization Exercise - Set Up



BL Customizations



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.

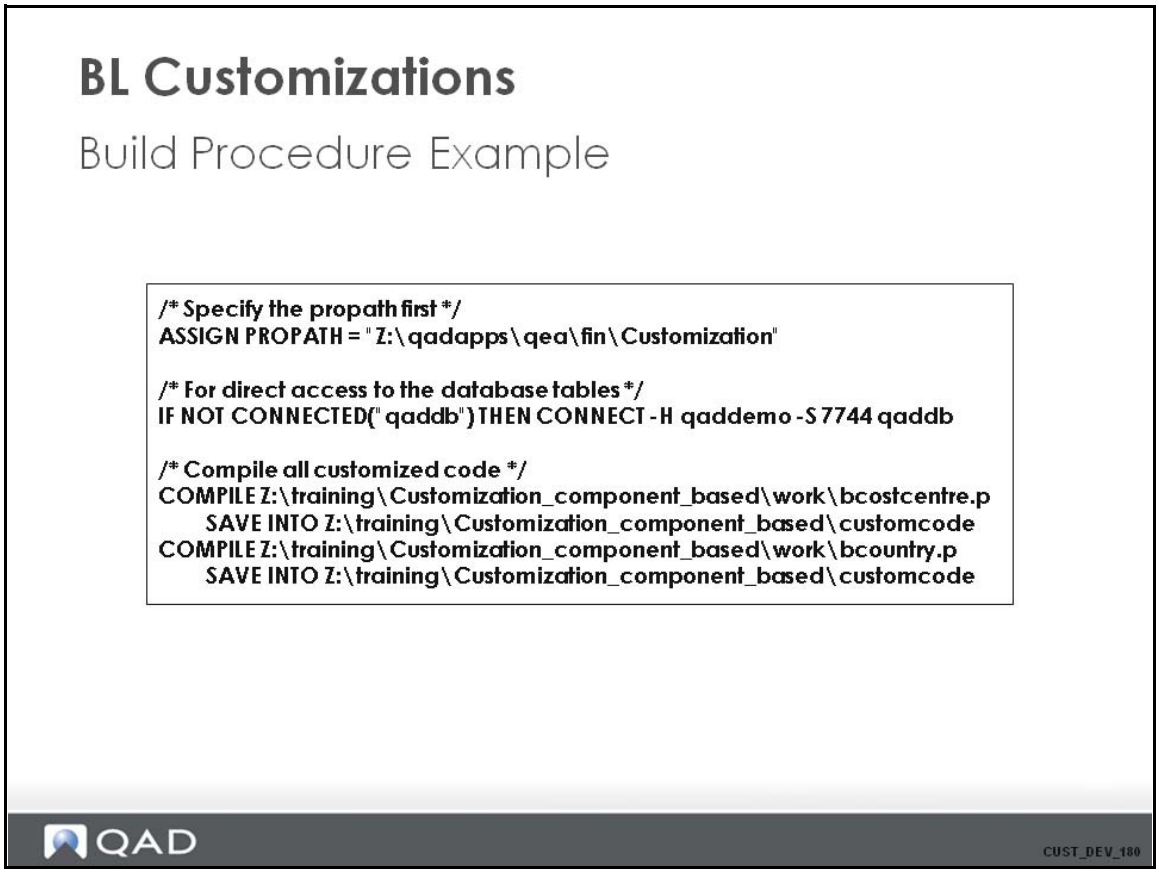
For example:

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`.

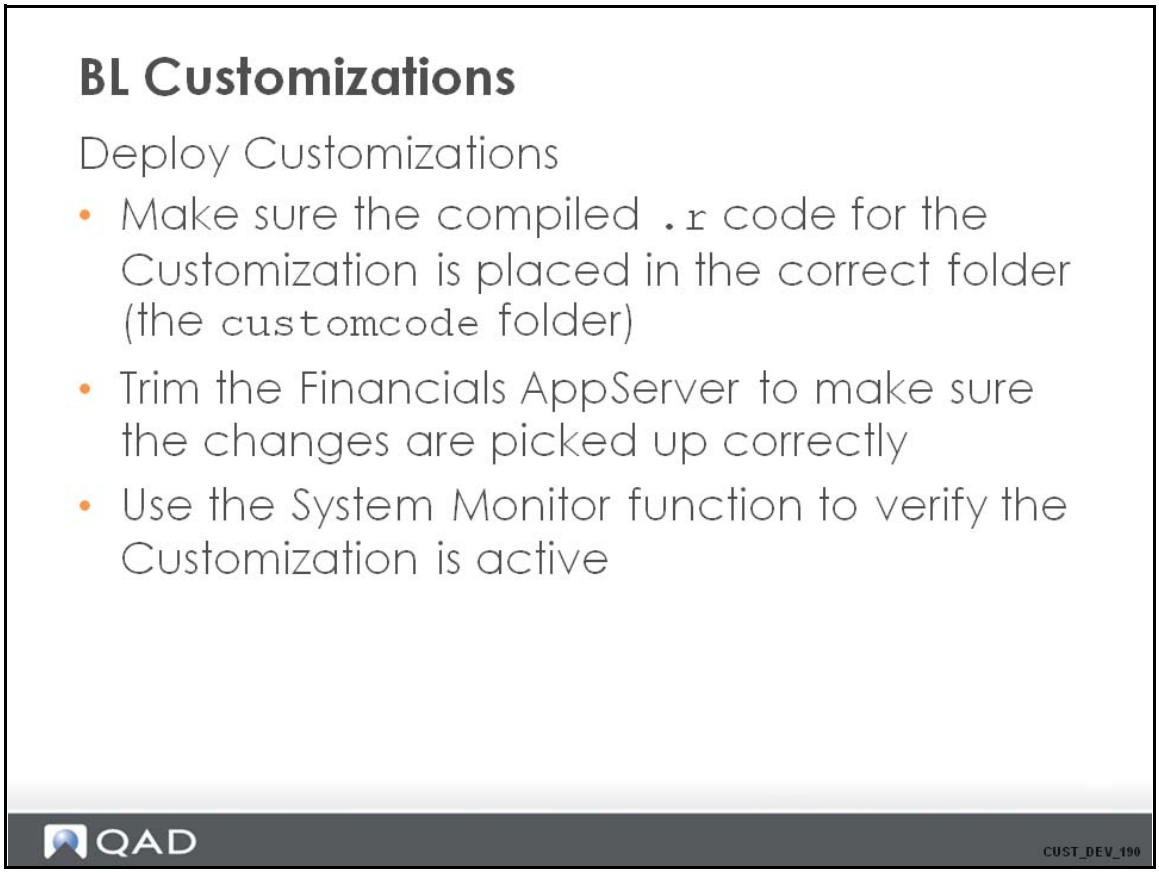
BL Customizations



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.

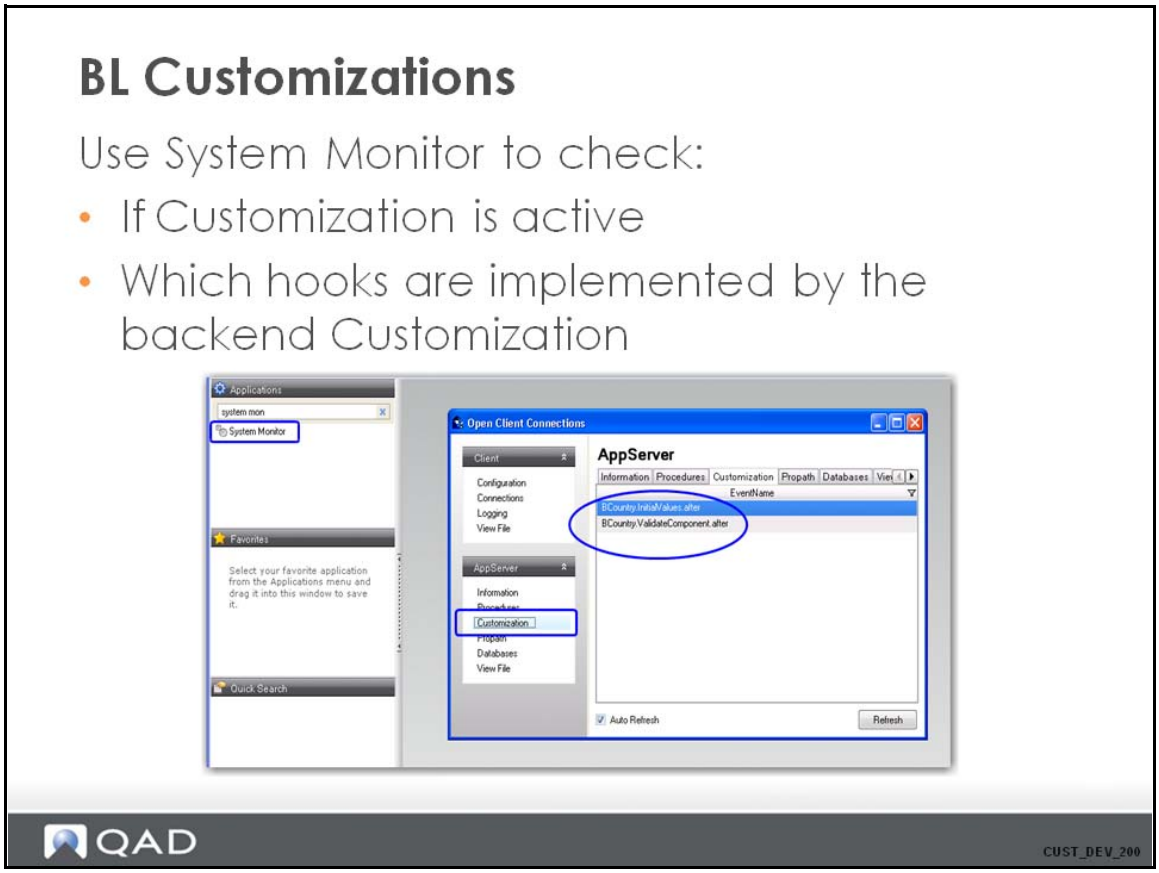
In the above example, it connects to the central qad applications database. If another database were used, the connection also must be made at runtime. To specify this connection, you update the central `server.xml` file.

BL Customizations



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.

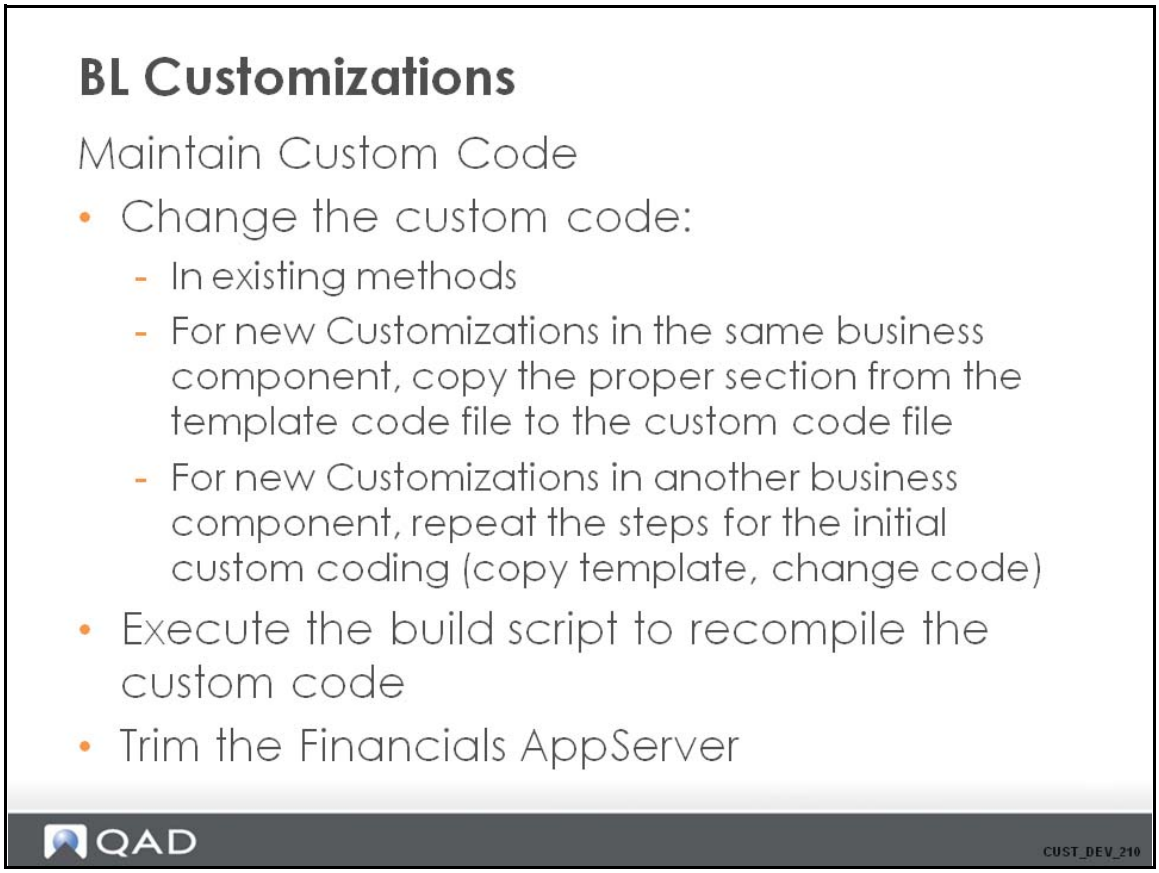
BL Customizations



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

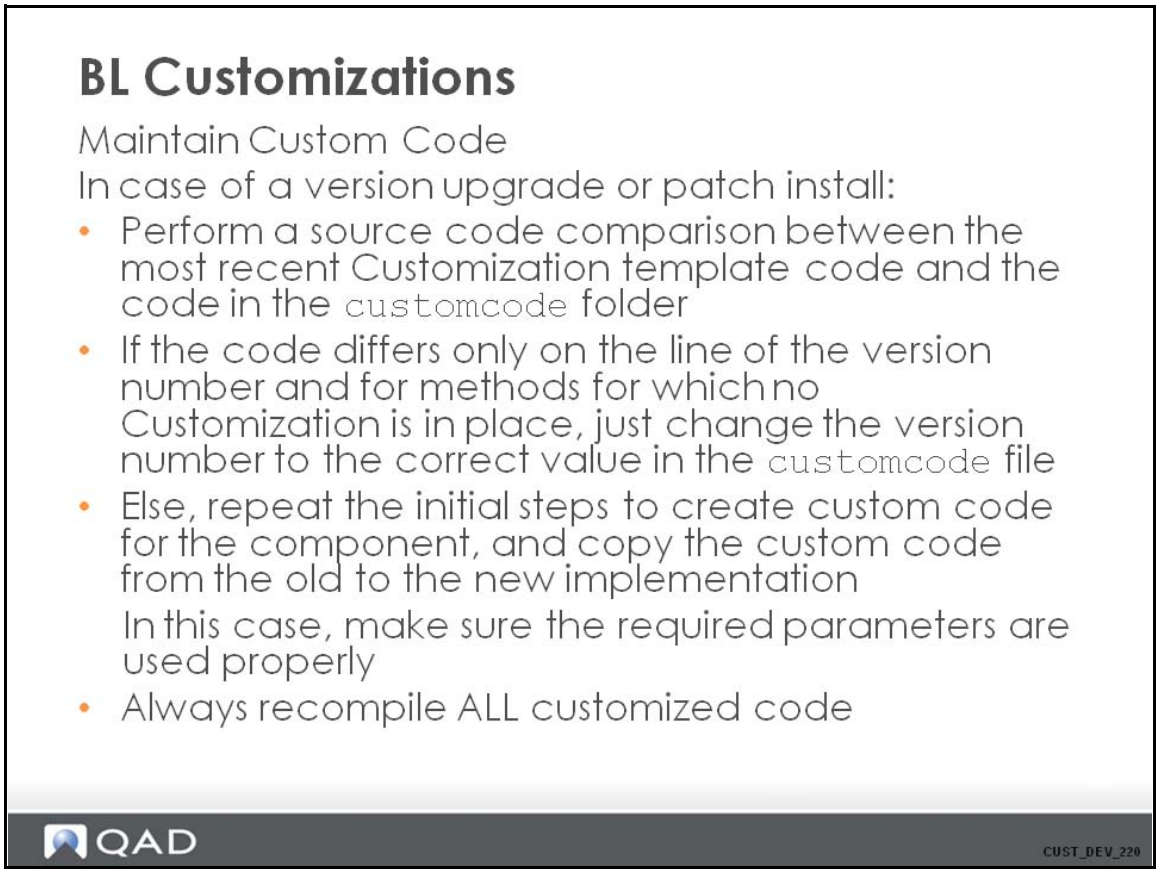
One of the interesting pieces of information is the list of customized hooks on the business layer. You can retrieve this by selecting “Customization” in the AppServer section.

BL Customizations

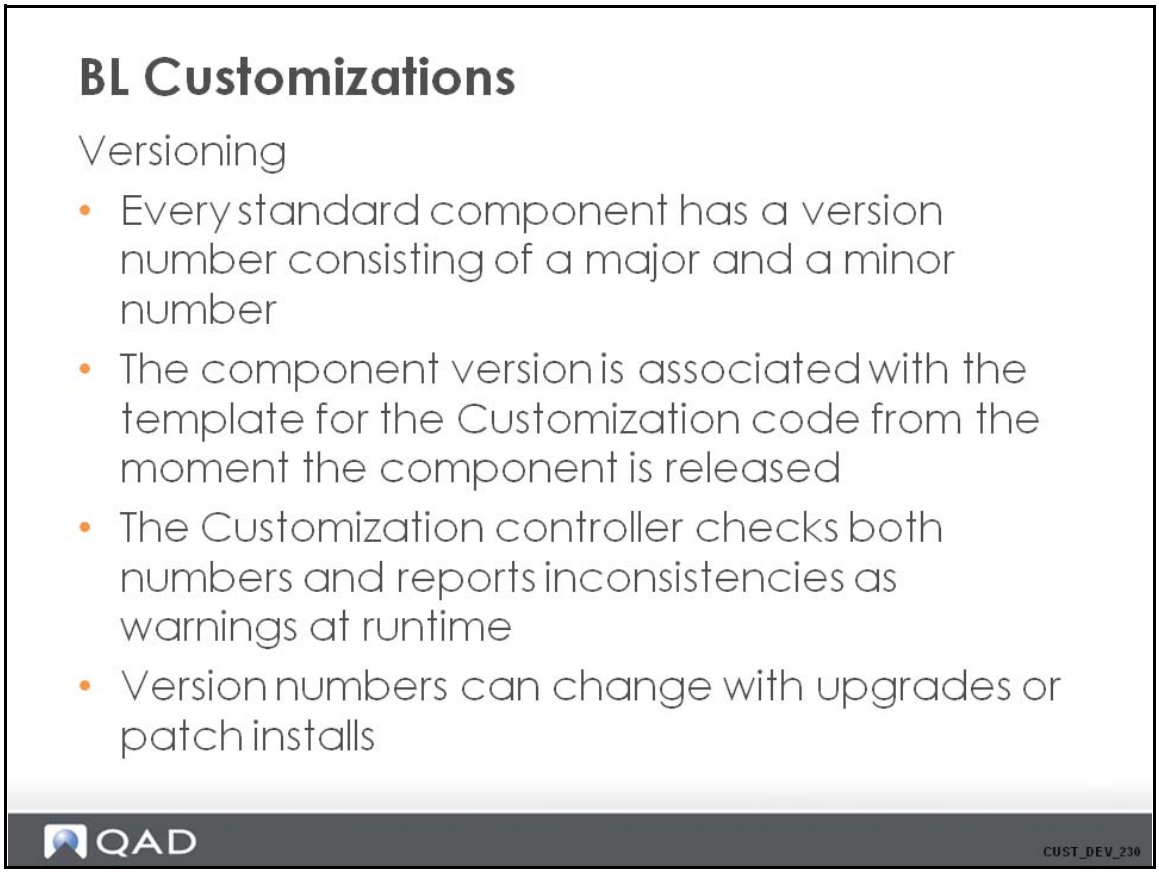


The customization developer has the ability to add code using custom code, but it is not possible to override standard logic.

BL Customizations



BL Customizations



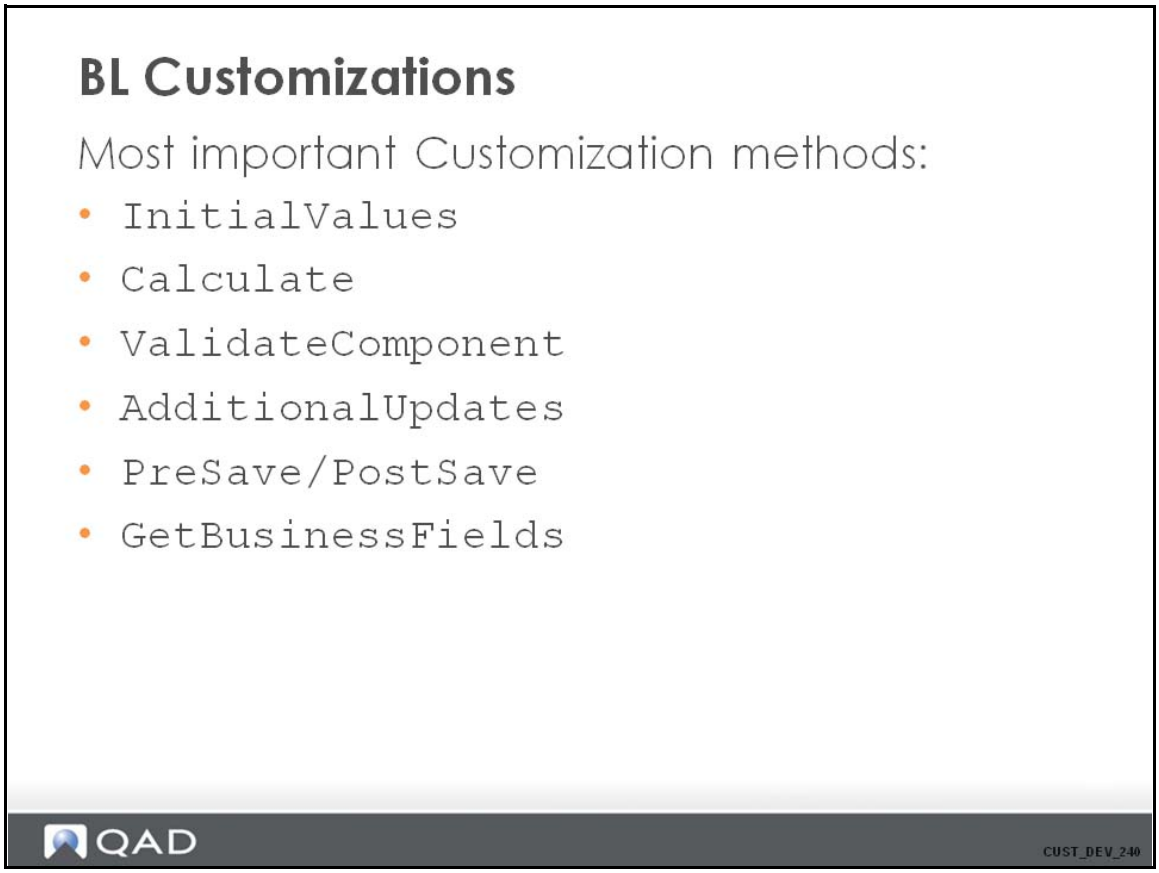
The mechanism built into the application ensures that every time a customization is started, the version is checked against the version of the standard component code that is currently used in memory. In a normal situation, this is not an issue. The system reports (in the form of a warning) as soon as 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.

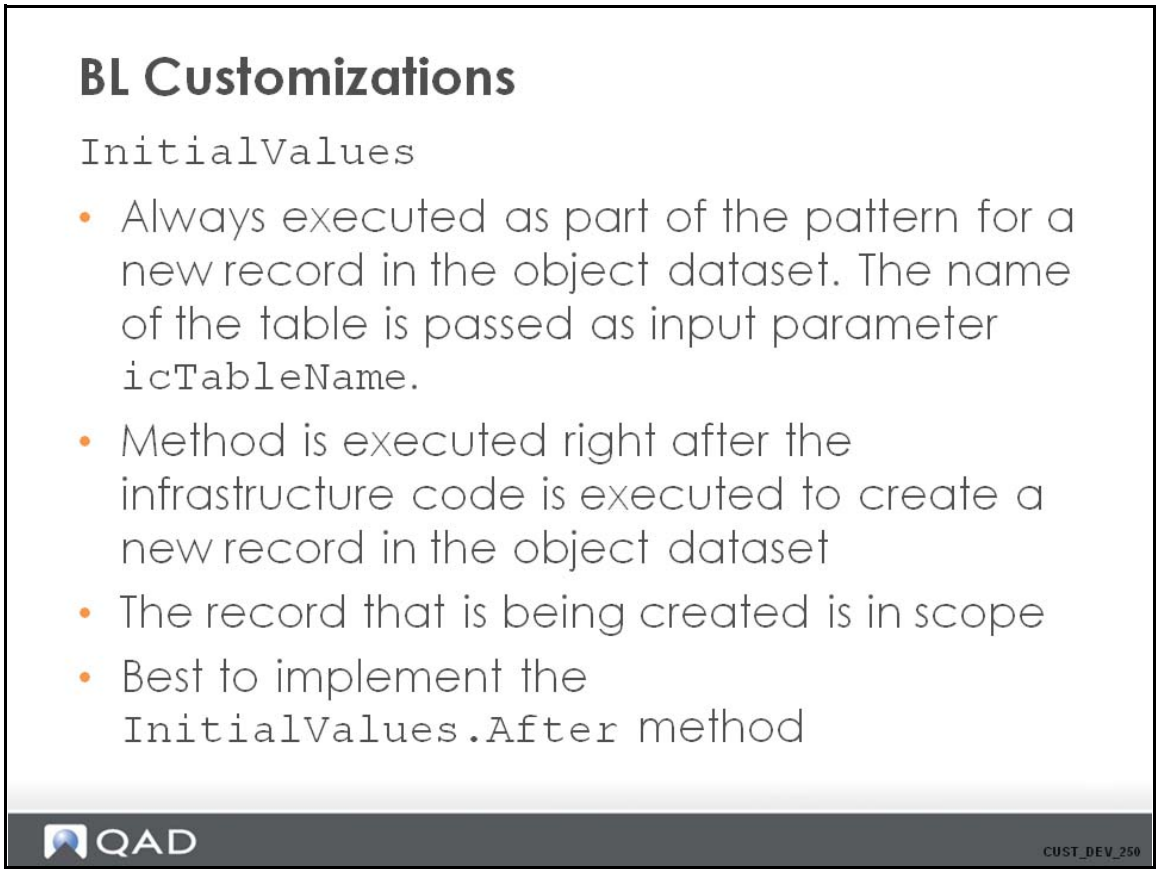
If the compiler displays the version conflict warnings, it is possible that there is still no issue, and that changes in the standard code do not impact the correct working of the custom code. In this case, the proposed procedure is to compare the custom code with the latest version of the template for 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



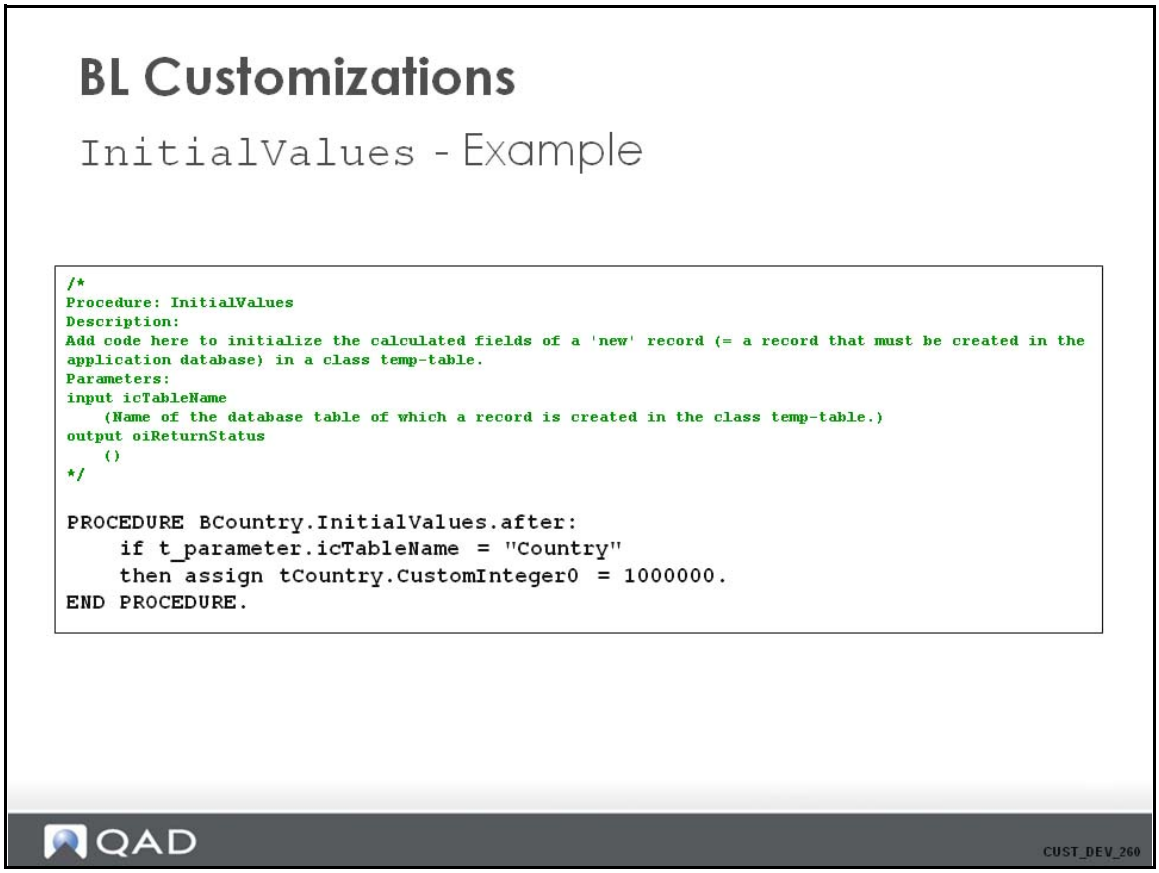
BL Customizations - InitialValues



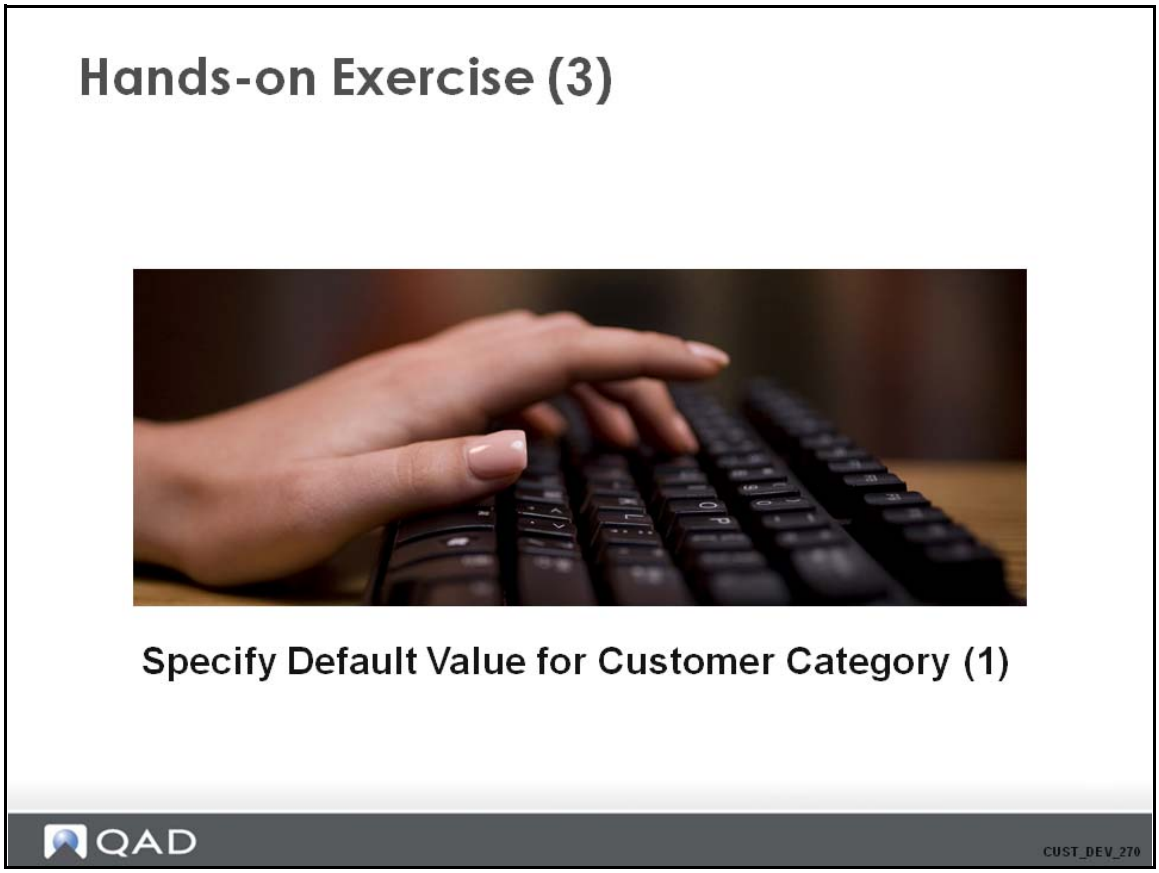
The InitialValues() method is executed after the infrastructure code filled in the fixed fields tc_rowid, tc_parentrowid and tc_status. In the standard code, the identifier field is also filled in. For example tCountry.Country_ID. An internal sequence is used to determine the value.

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.

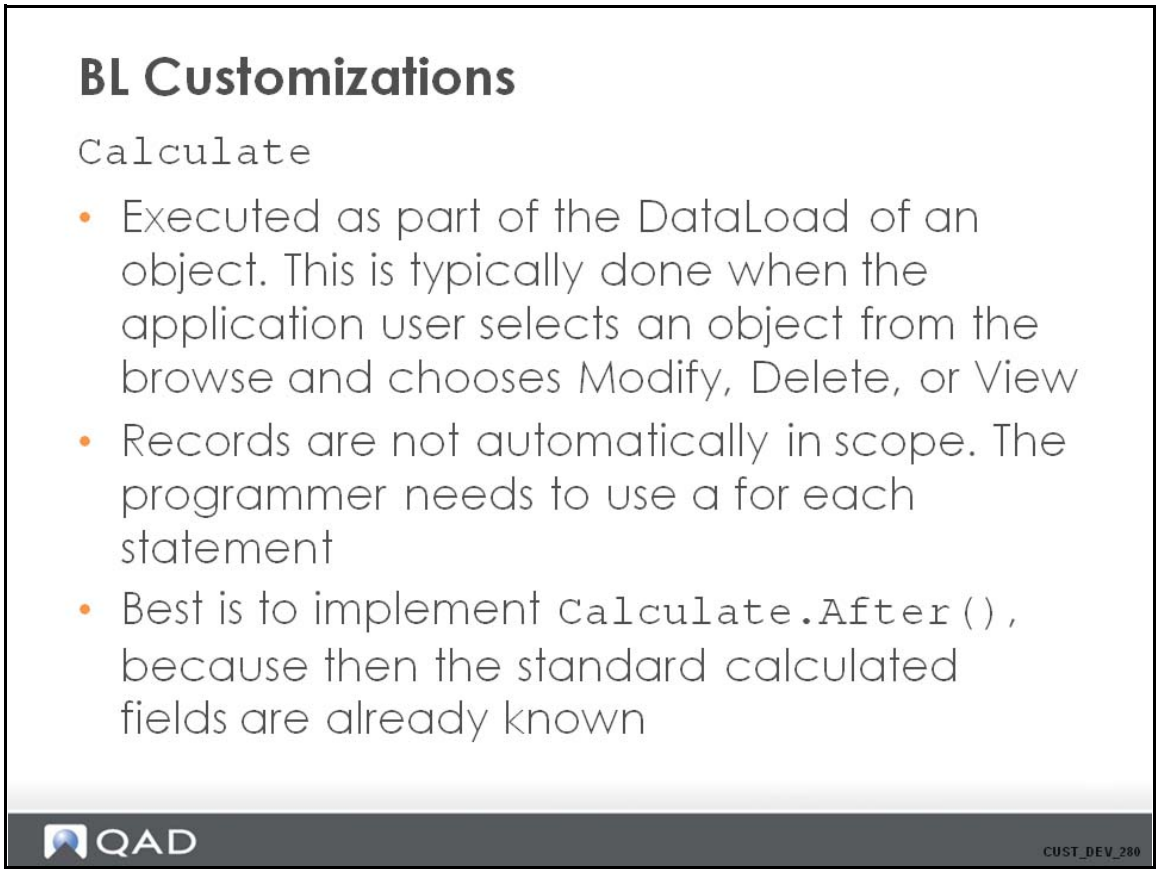
BL Customizations - InitialValues



Hands-on Exercise (3)



BL Customizations - Calculate

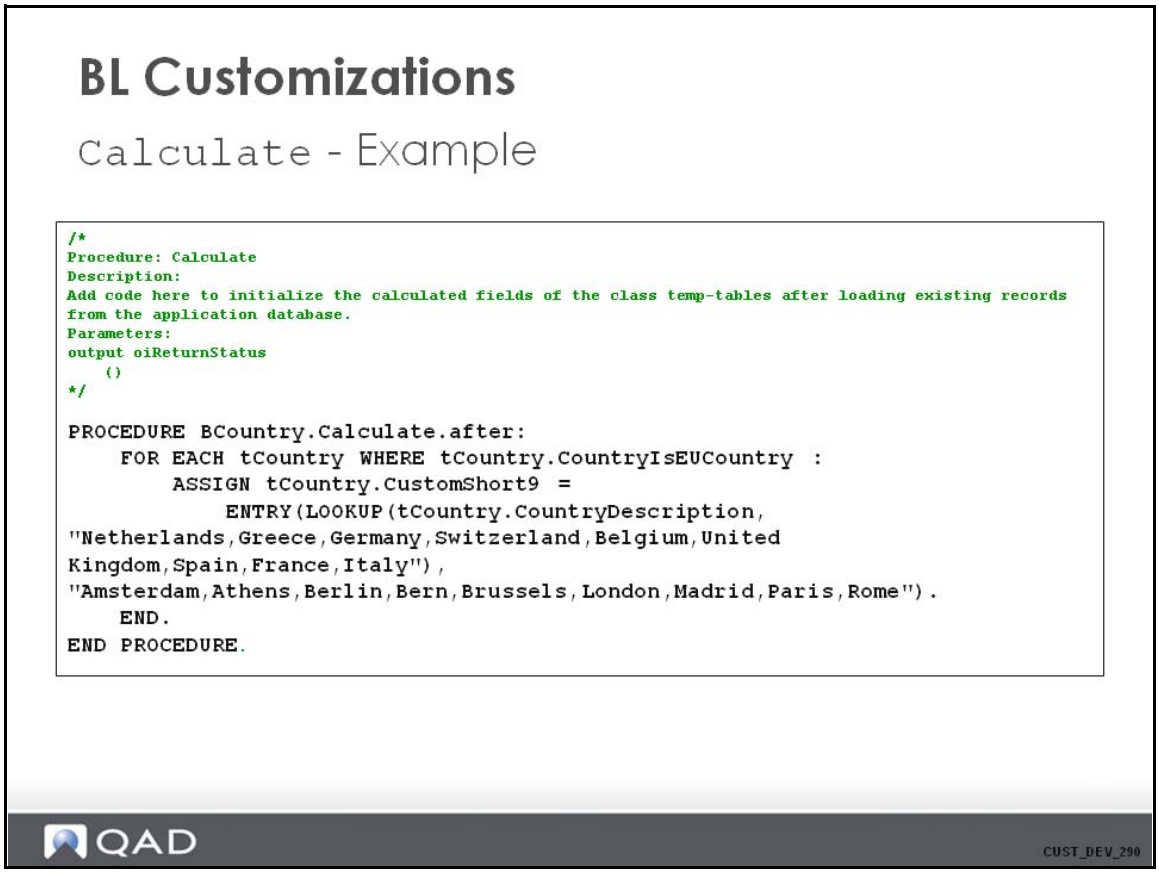


Each time a Financials object is loaded for modification, the application goes through the DataLoad pattern. The first thing that happens is the data is read from the database. Right after that, the Calculate method is executed. The intent of this method is to “calculate” the data for the calculated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

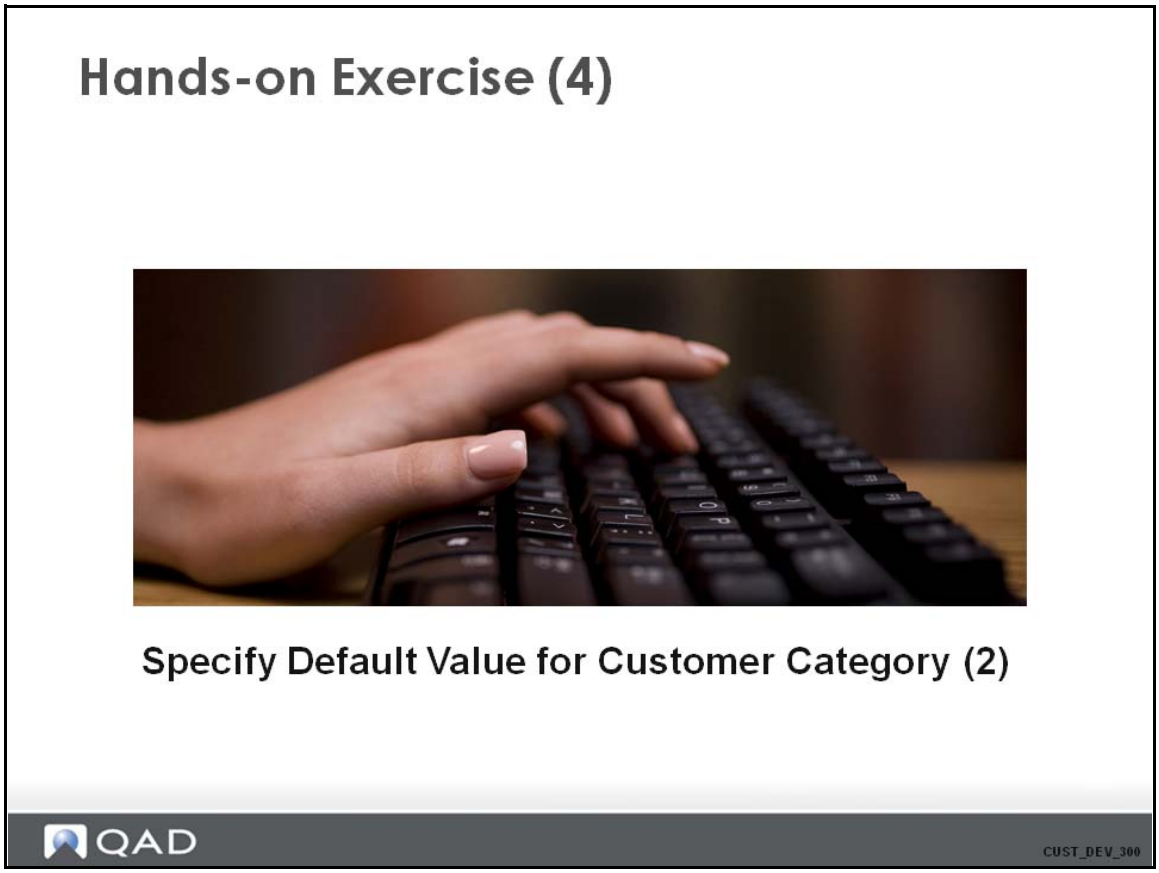
Multiple objects, main table records, and child table records can be loaded during the data load. Therefore, the programmer needs to use for-each statements to loop over the records in the object dataset.

BL Customizations - Calculate

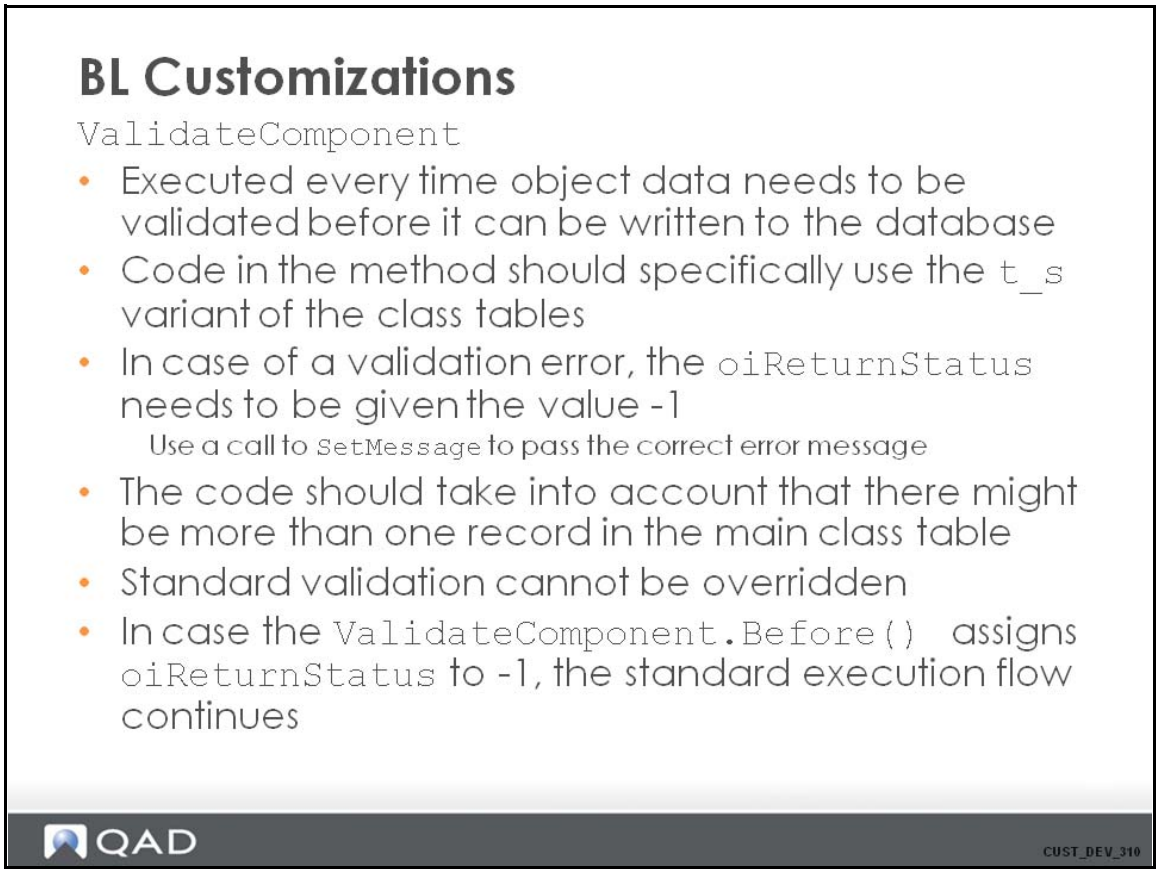


Save the capital of some European countries into CustomShort9.

Hands-on Exercise (4)



BL Customizations - ValidateComponent



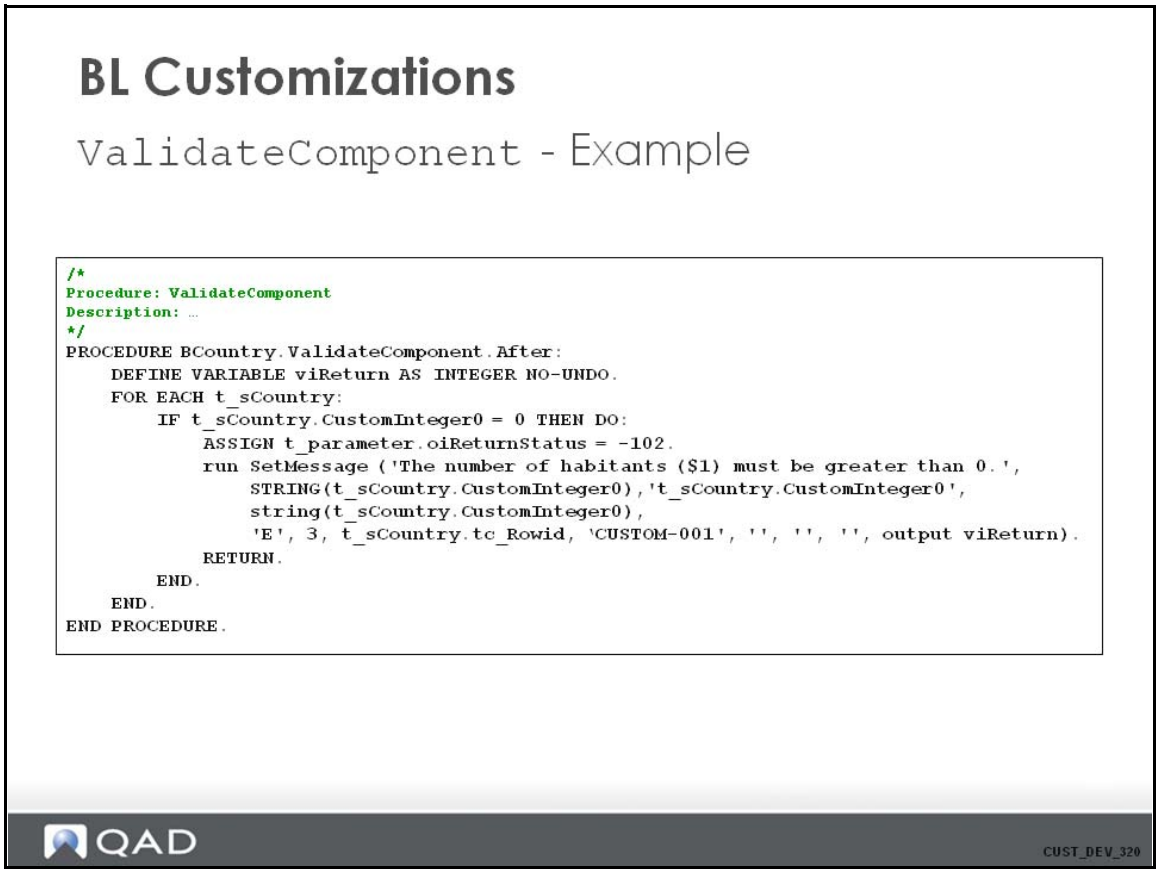
ValidateComponent is the central validation method for each component. It is typically intended to contain all object-wide validation. Besides the ValidateComponent, very specific field-level validations might also be in place.

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.

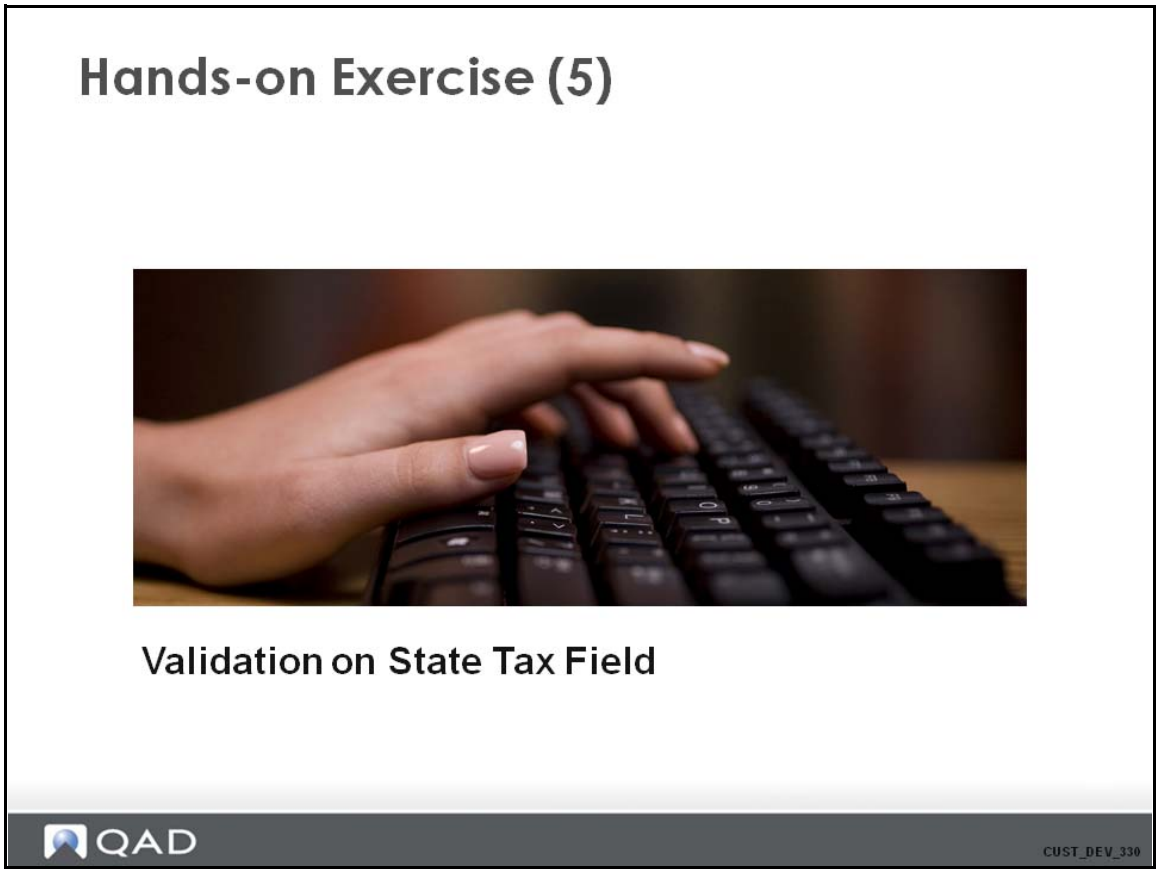
After the data has been validated, the system updates the data in the official object dataset (O buffers) with the validated changed data, executes AdditionalUpdates, and performs a DataSave.

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.

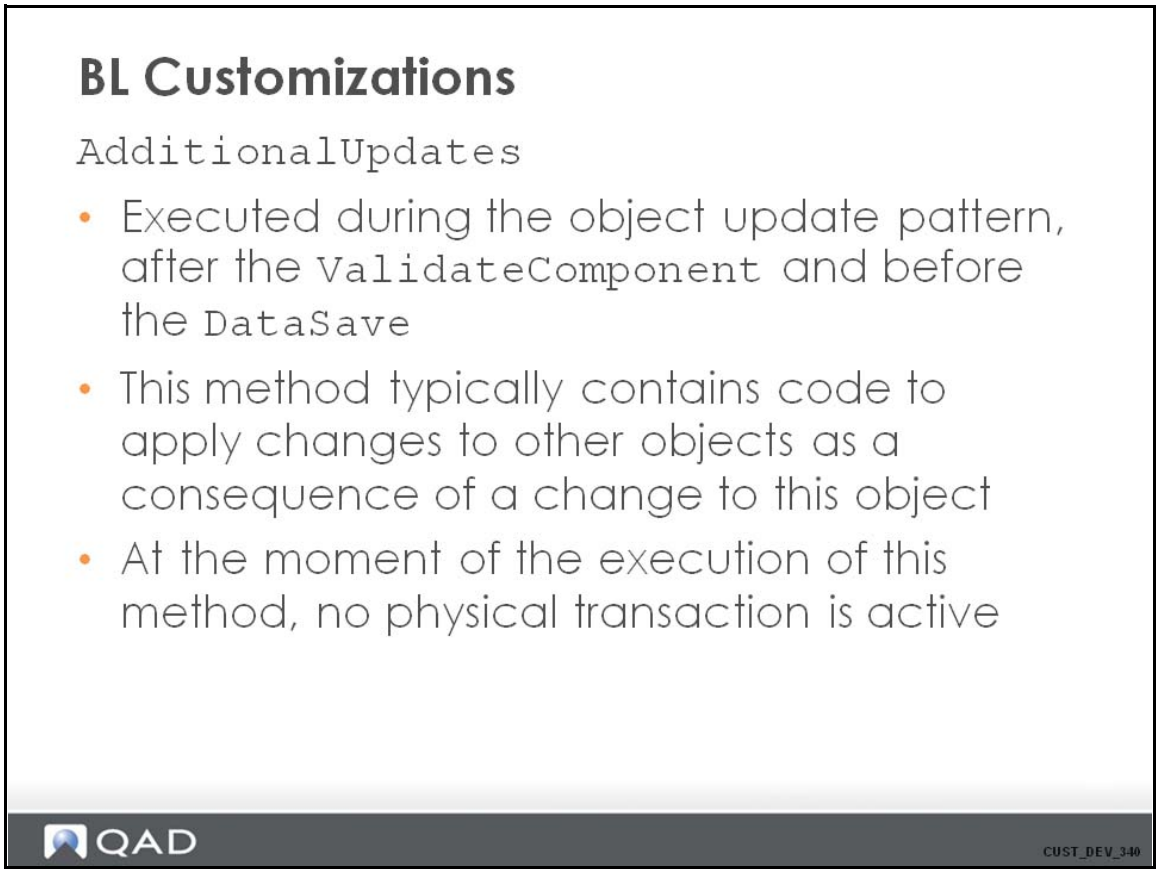
BL Customizations - ValidateComponent



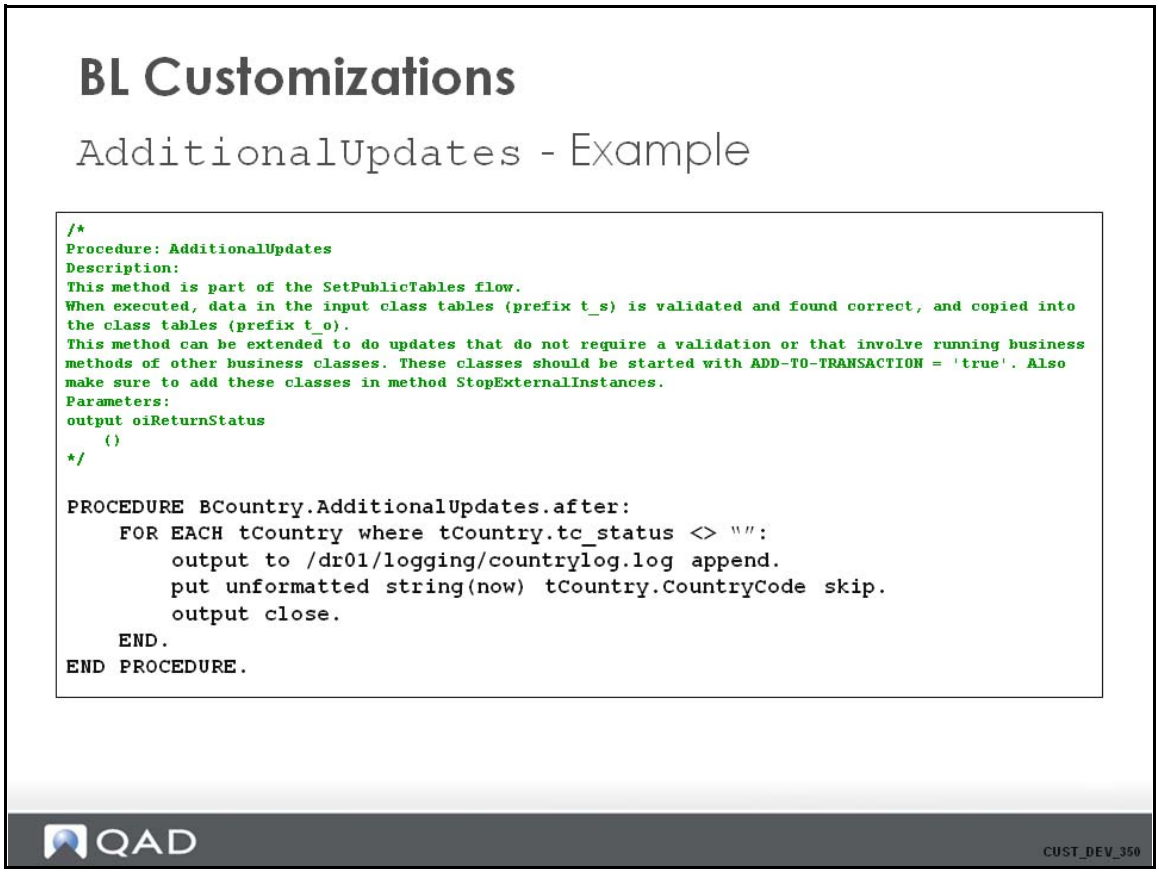
Hands-on Exercise (5)



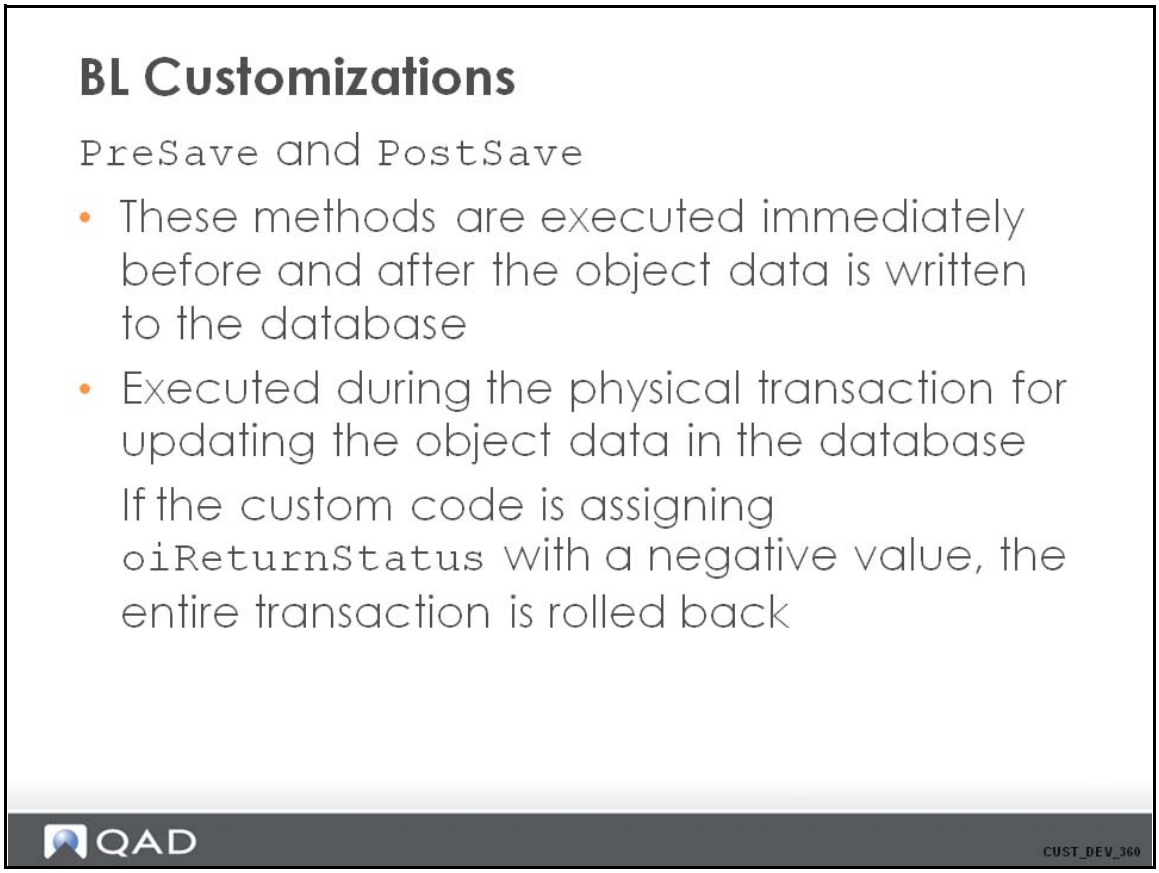
BL Customizations - Additional Updates



BL Customizations - Additional Updates

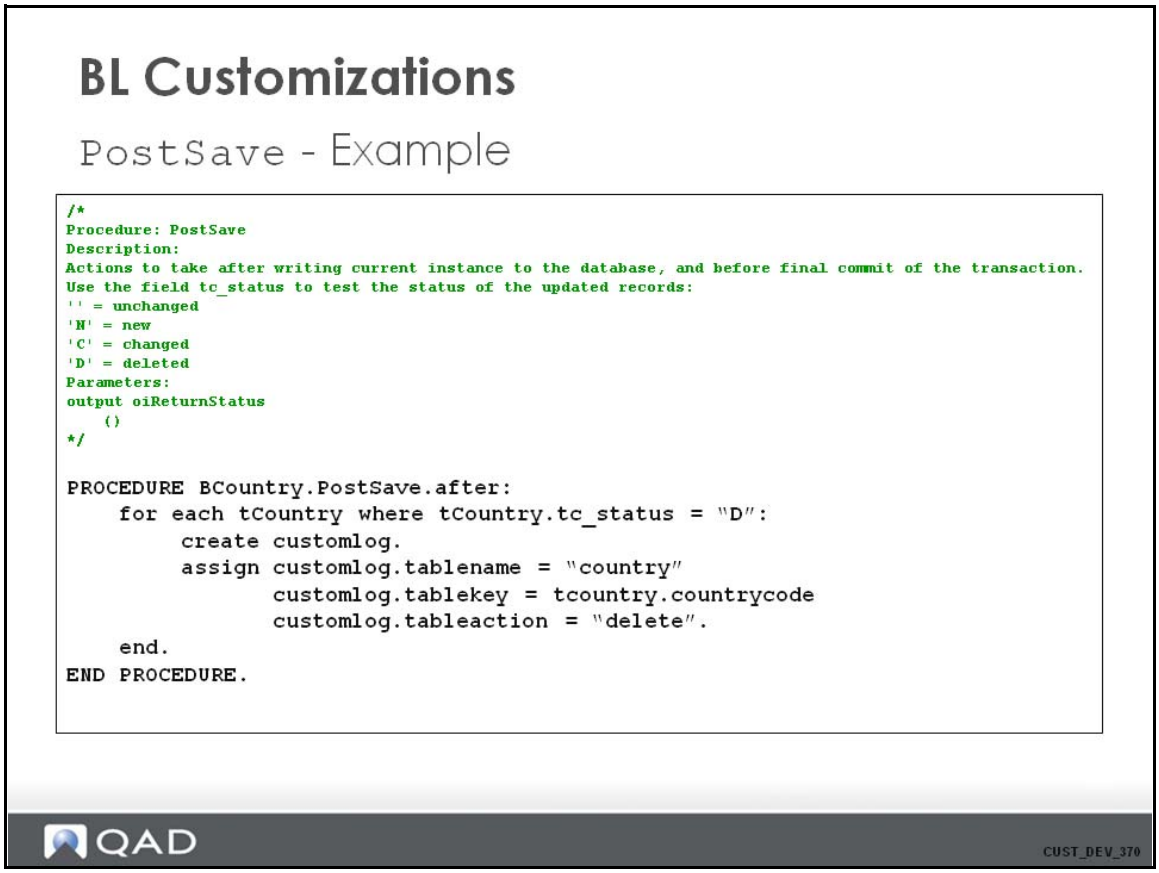


BL Customizations - PreSave and PostSave

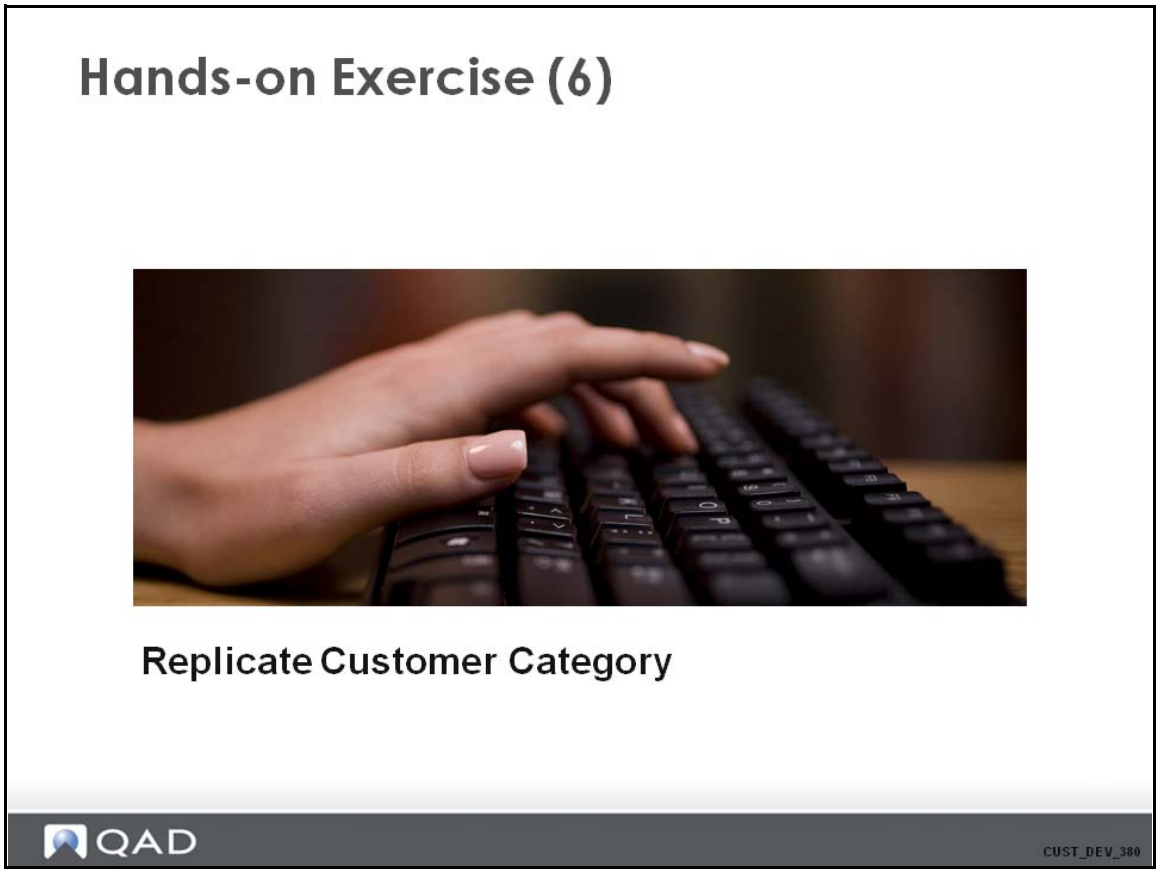


These methods are typically used to update records in the database and the update needs to be in the same physical transaction. An example is the update of custom tables in the database when the tCustomTable* tables are used in the object dataset.

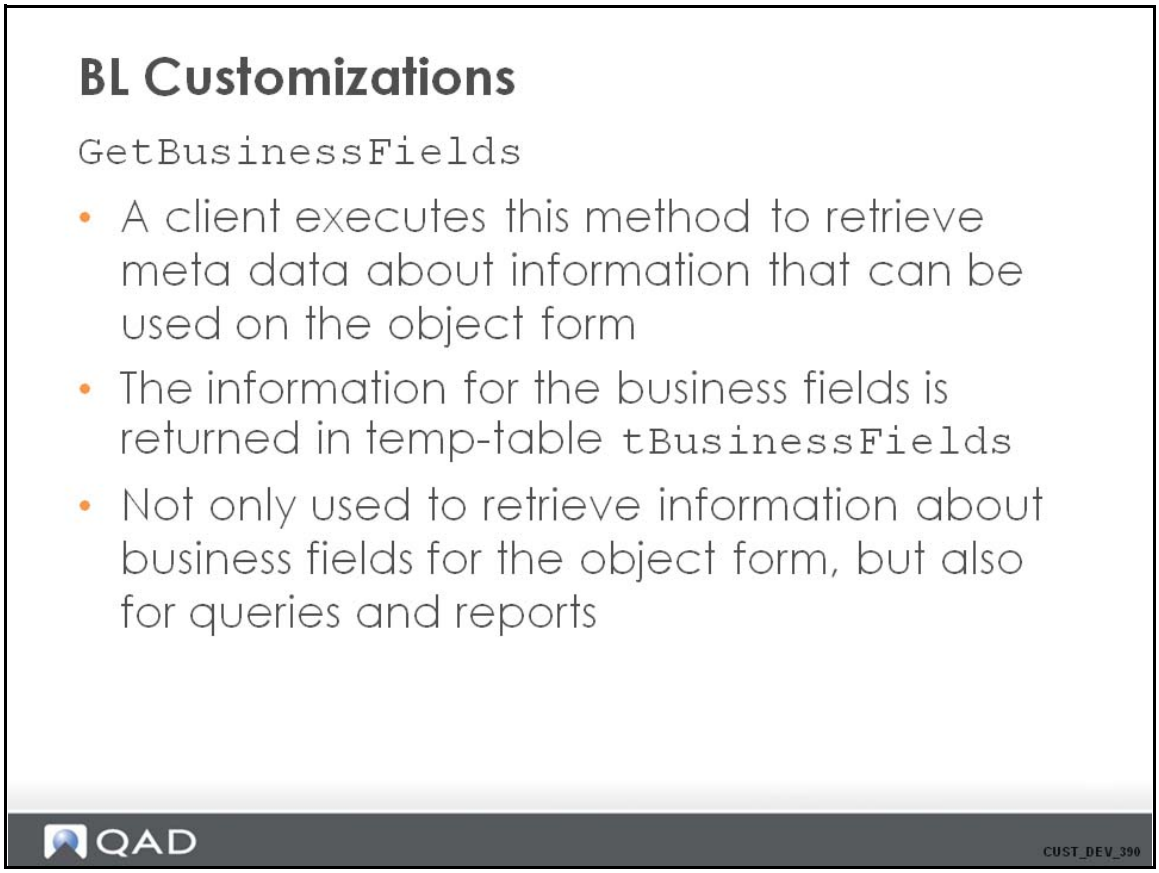
BL Customizations - PostSave



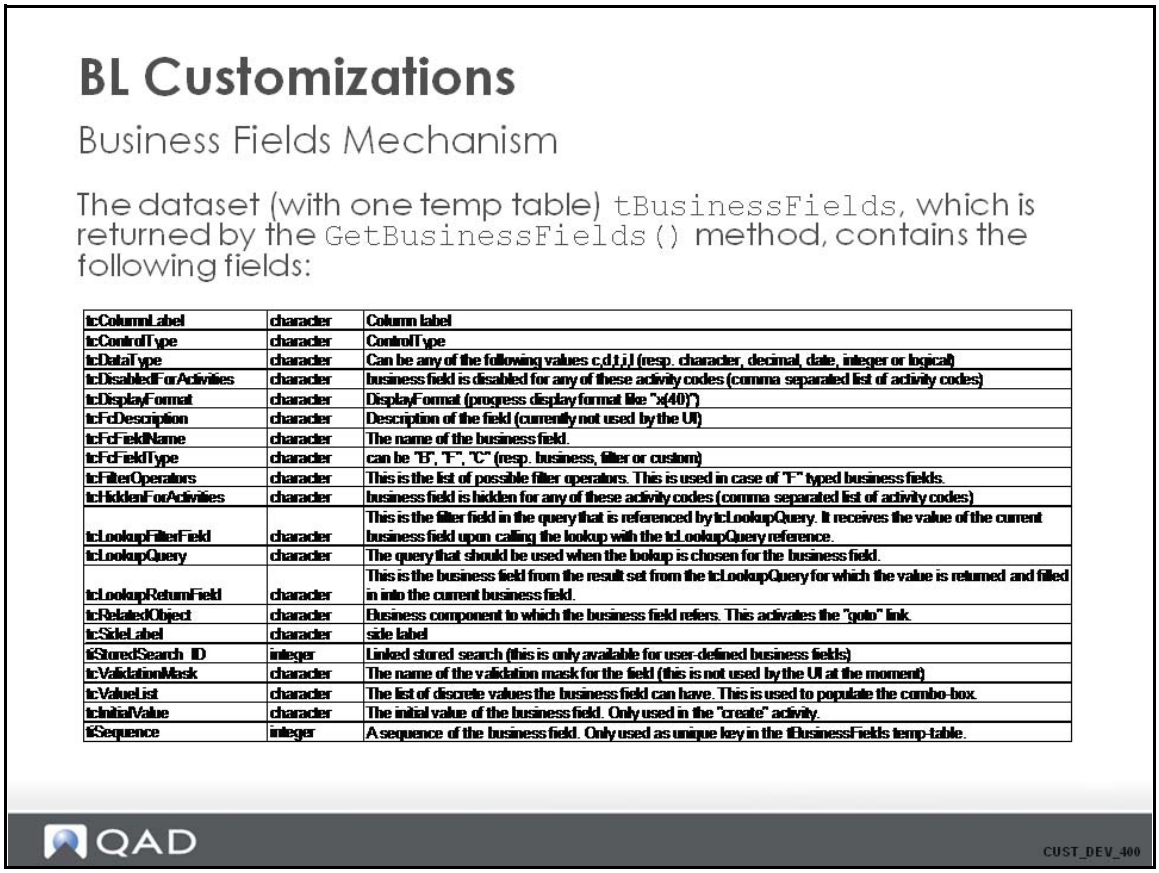
Hands-on Exercise (6)



BL Customizations - GetBusinessFields



BL Customizations - Business Fields Mechanism



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.

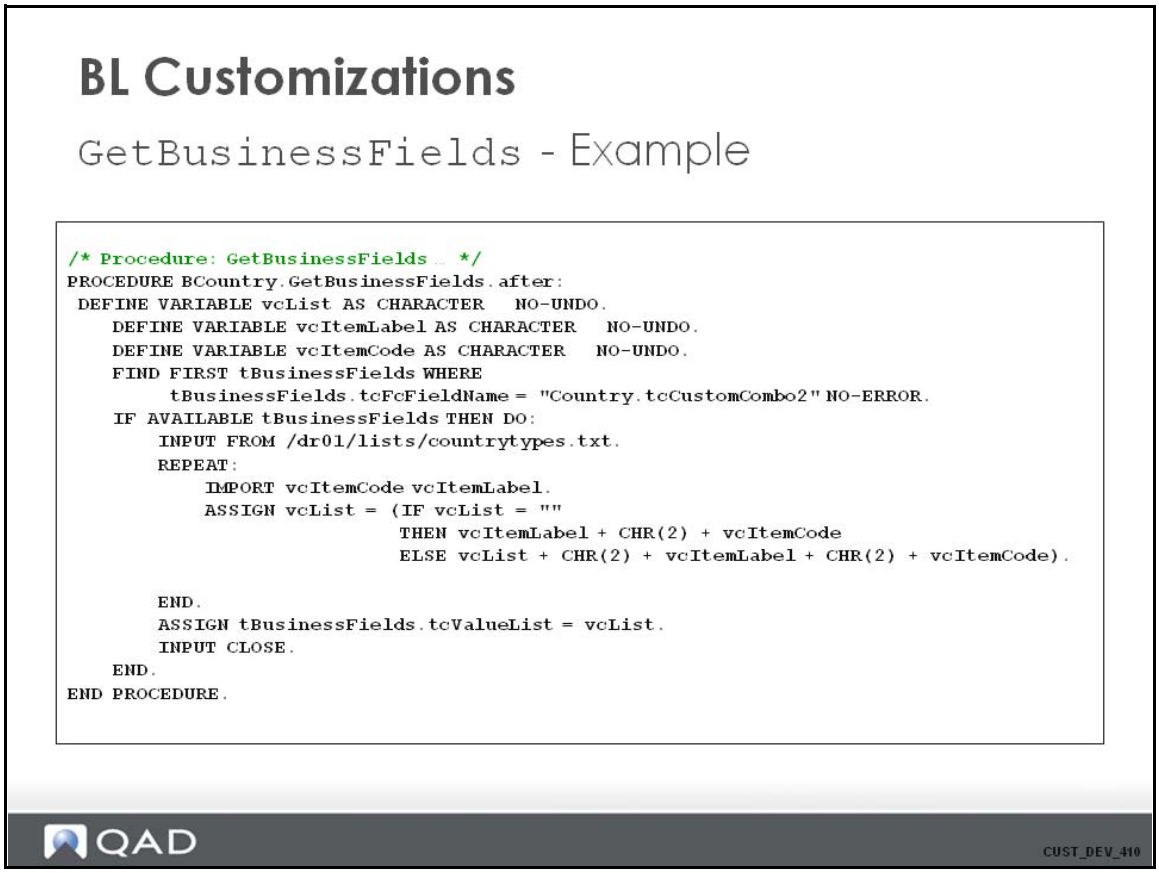
For a query, GetBusinessFields() is called from within the browse or lookup. There are three clear types of business field returned. Type "F" fields are the filter business fields. These fields are used in the browse / lookup in the filter area. Type "B" fields are the fields that are available in the 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.

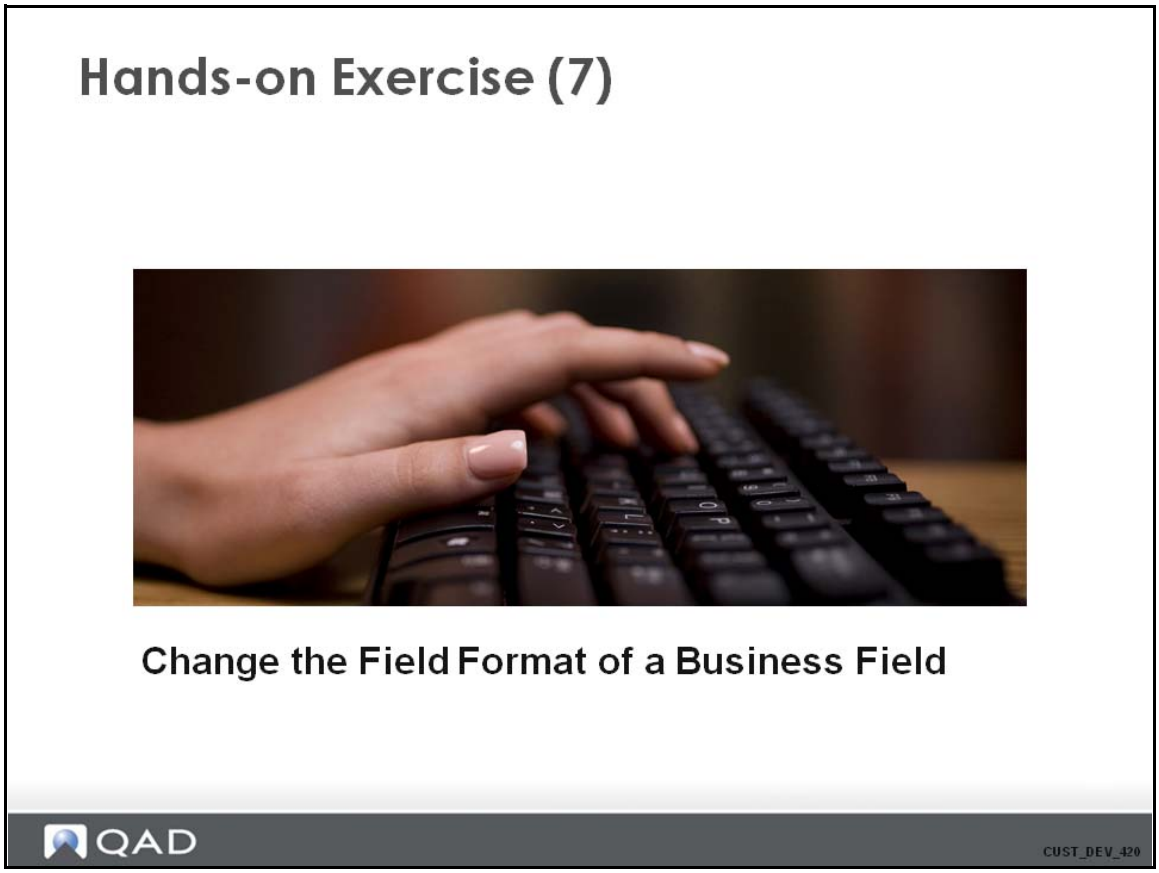
tcControlType can be any of these values: NumericDecimal / NumericInt / Bool / DateTime / TextBox / ComboBox

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) …

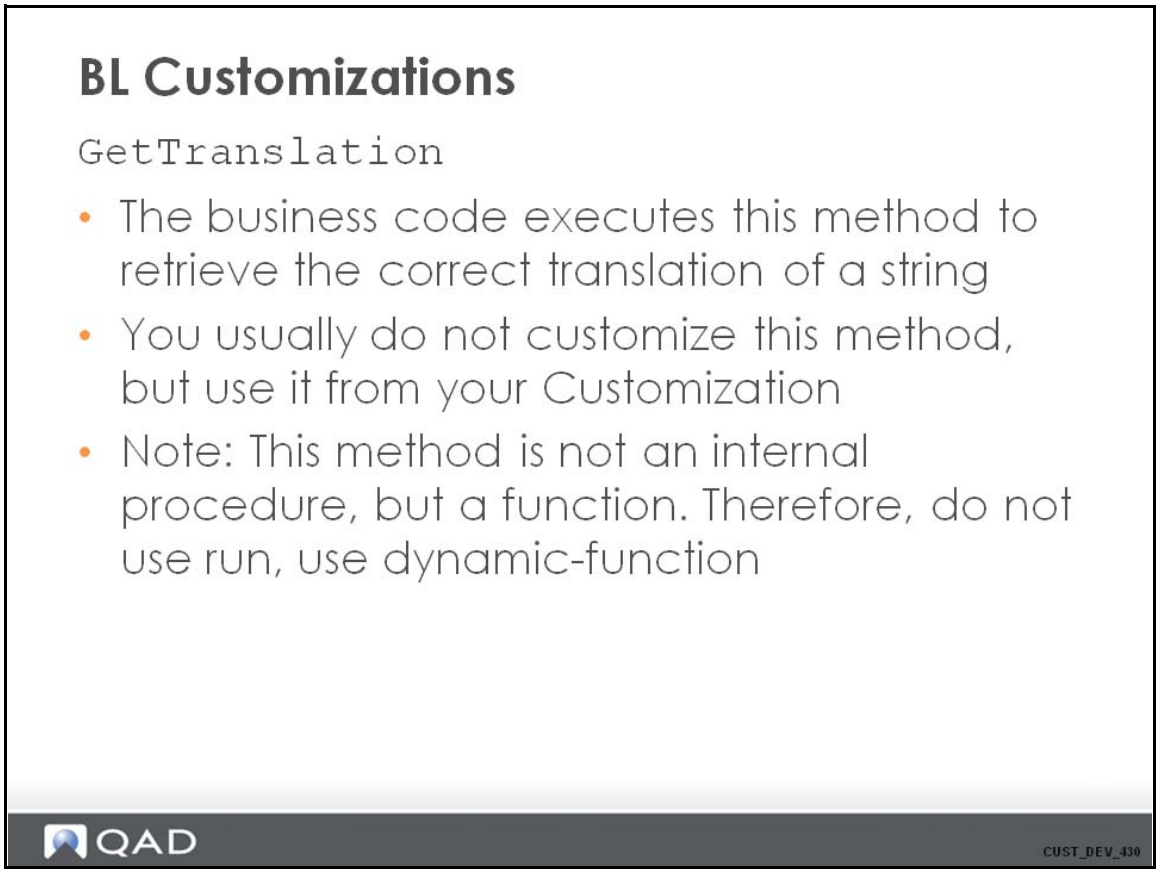
BL Customizations - GetBusinessFields



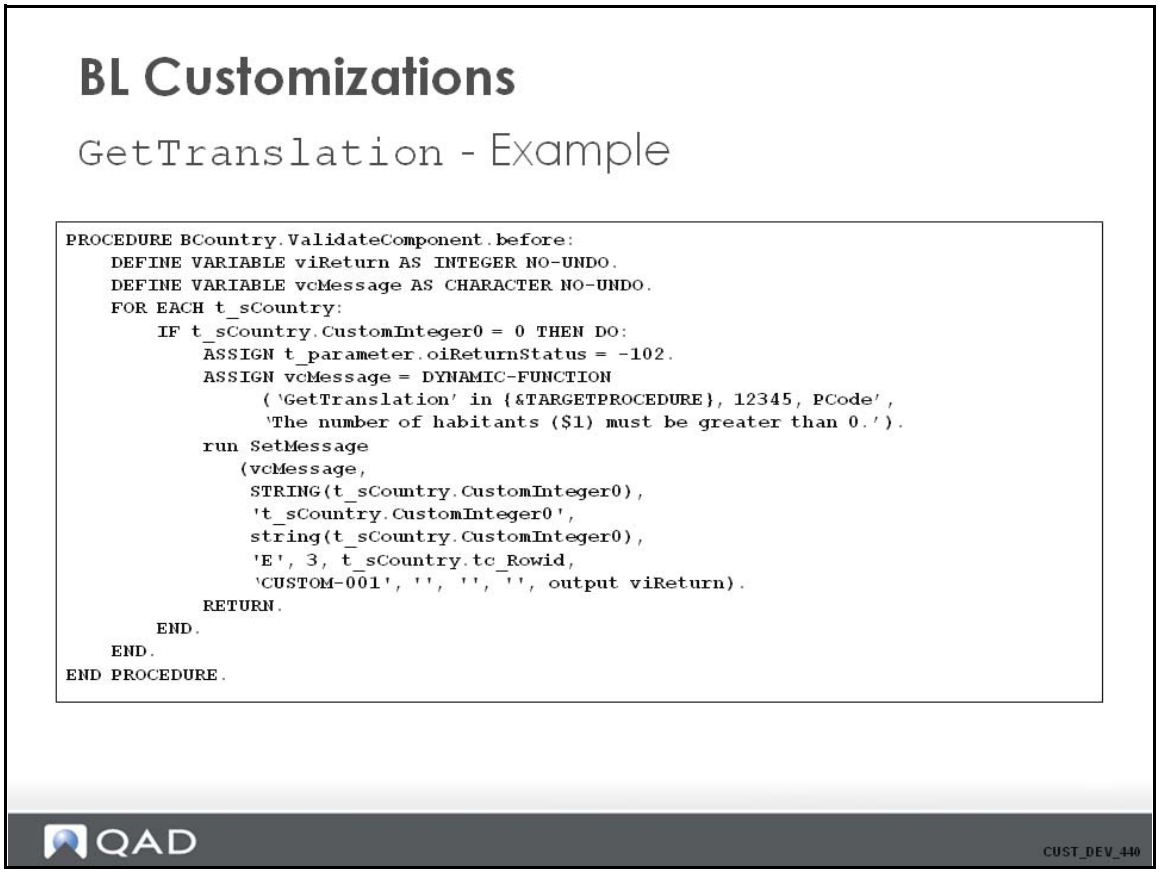
Hands-on Exercise (7)



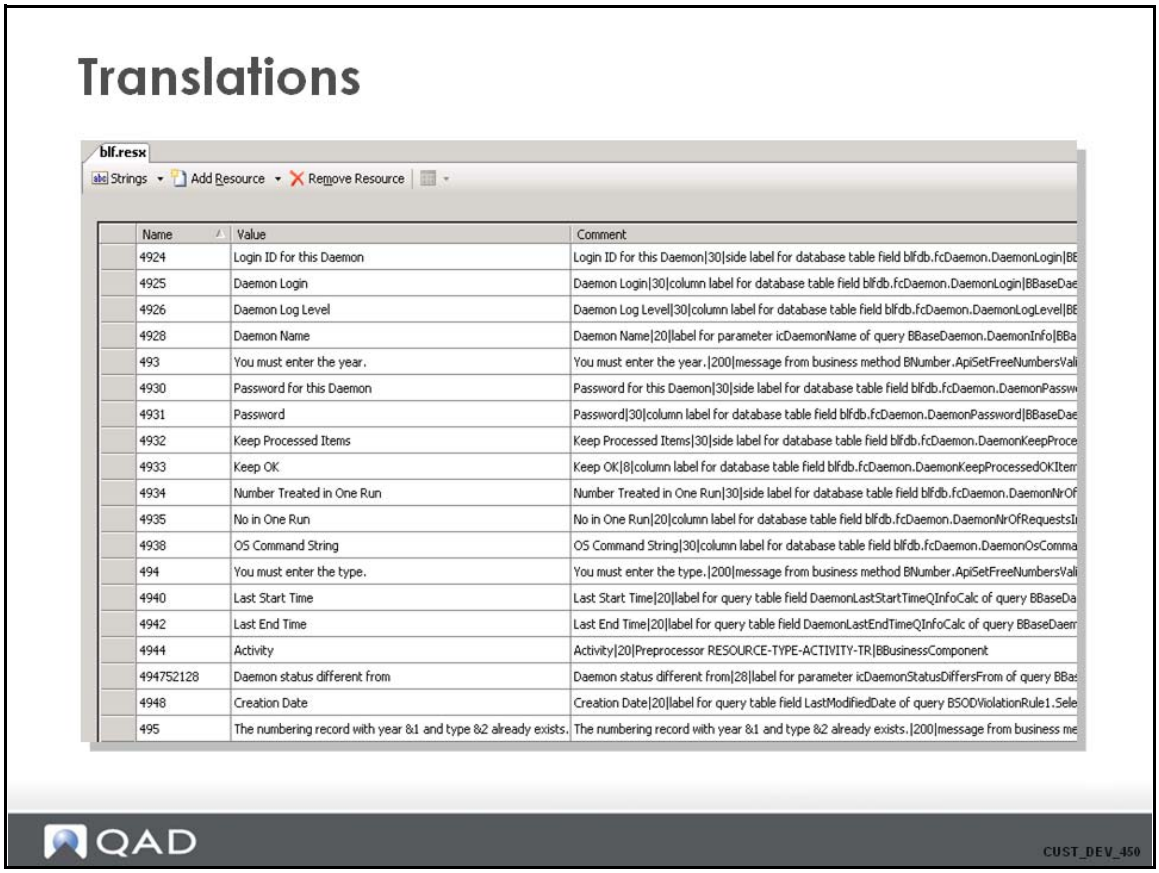
BL Customizations - GetTranslation



BL Customizations - GetTranslation



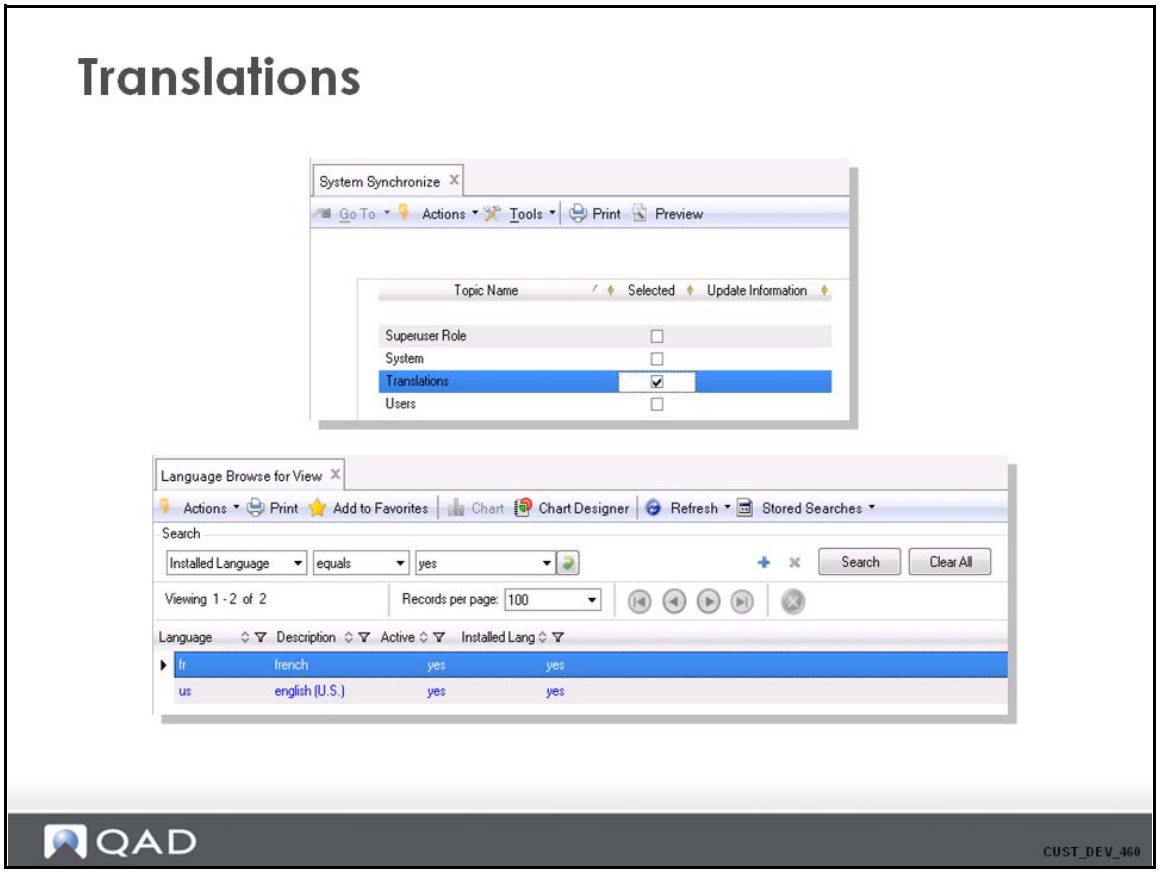
Translations



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

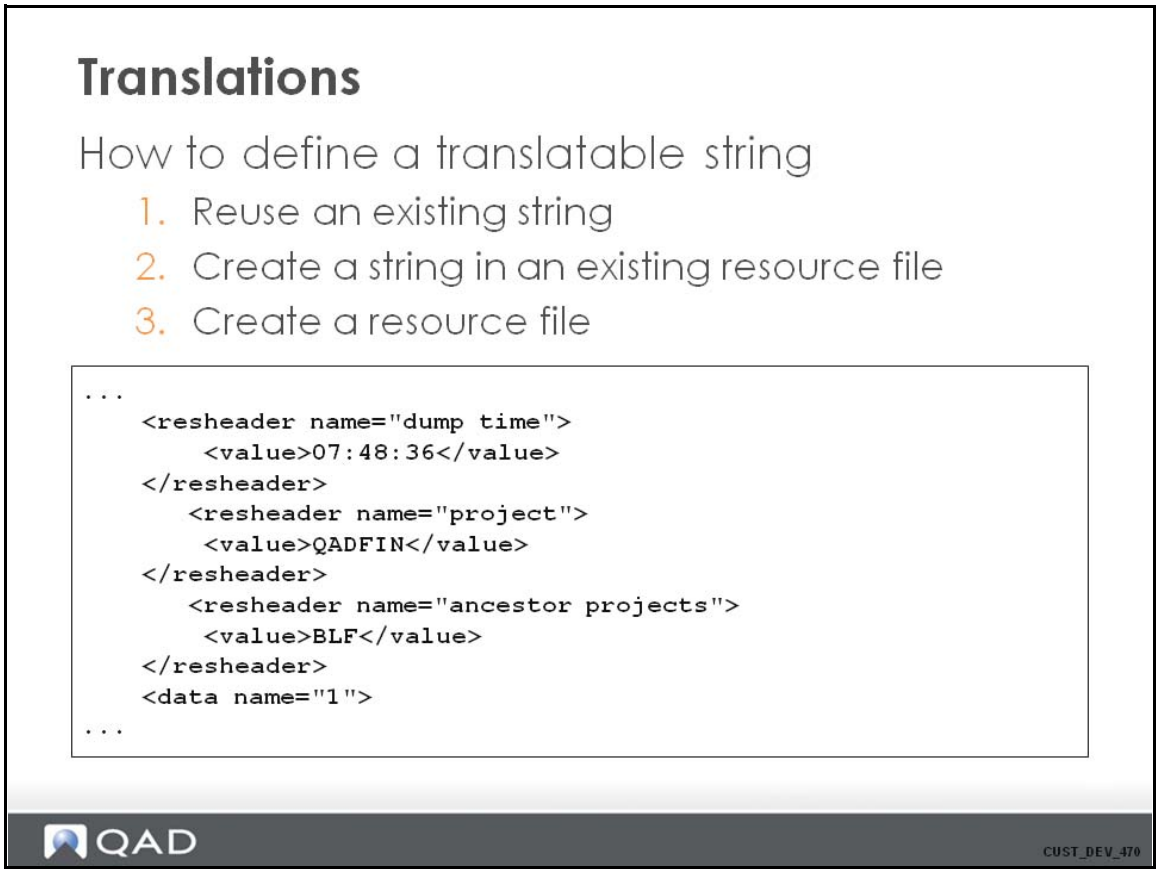
Inside a resource file, a unique number identifies each string. Each resource file is then translated into each required language and renamed as `<project>.<language>.resx` like `f.e. qadfin.fr.resx`.

Translations



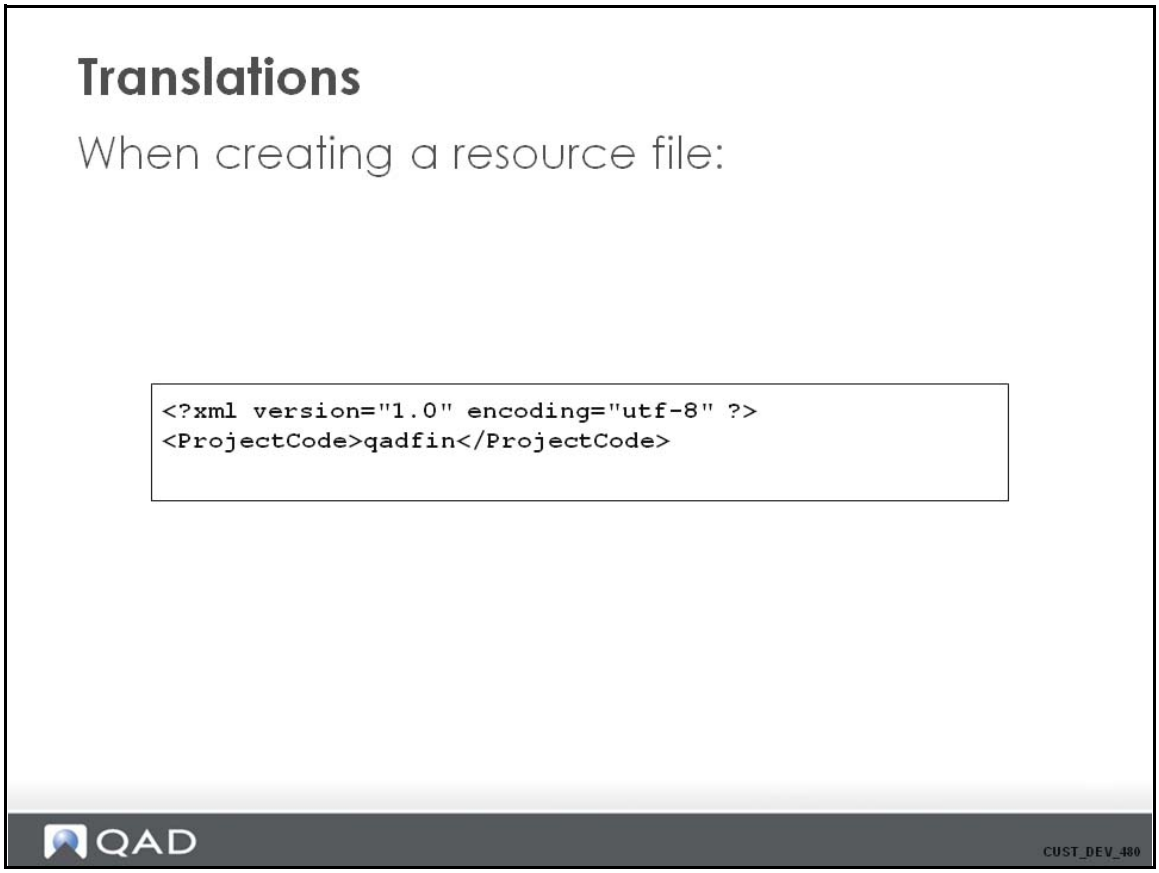
Translation resource files are loaded in the system synchronize function. Translations are loaded for all installed language codes.

Translations



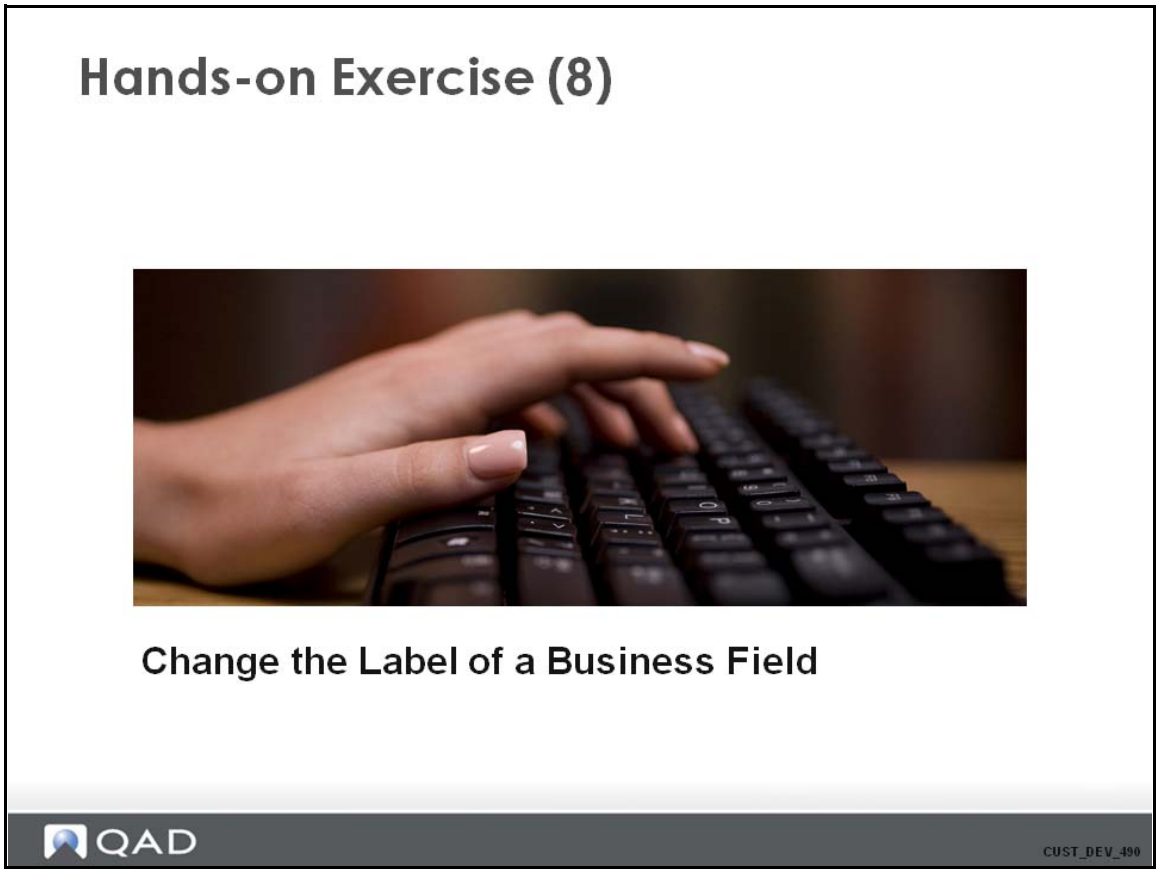
When modifying an existing resource file, you must redo your changes whenever a new version of the application is installed. Creating a resource file is upgrade-safe. You can use any editor you like to maintain resource files, like Visual Studio, or even Notepad. When creating a resource file, you can 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



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

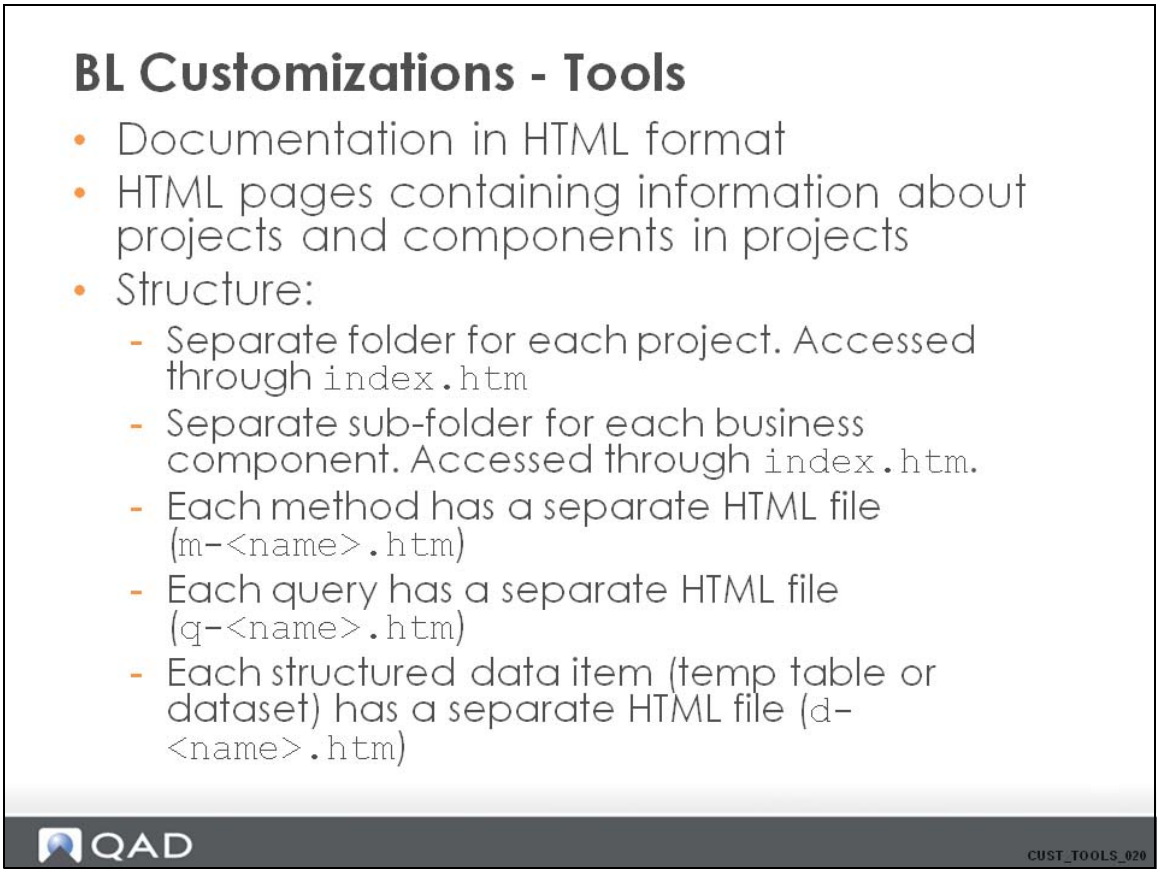
Hands-on Exercise (8)



Chapter 3

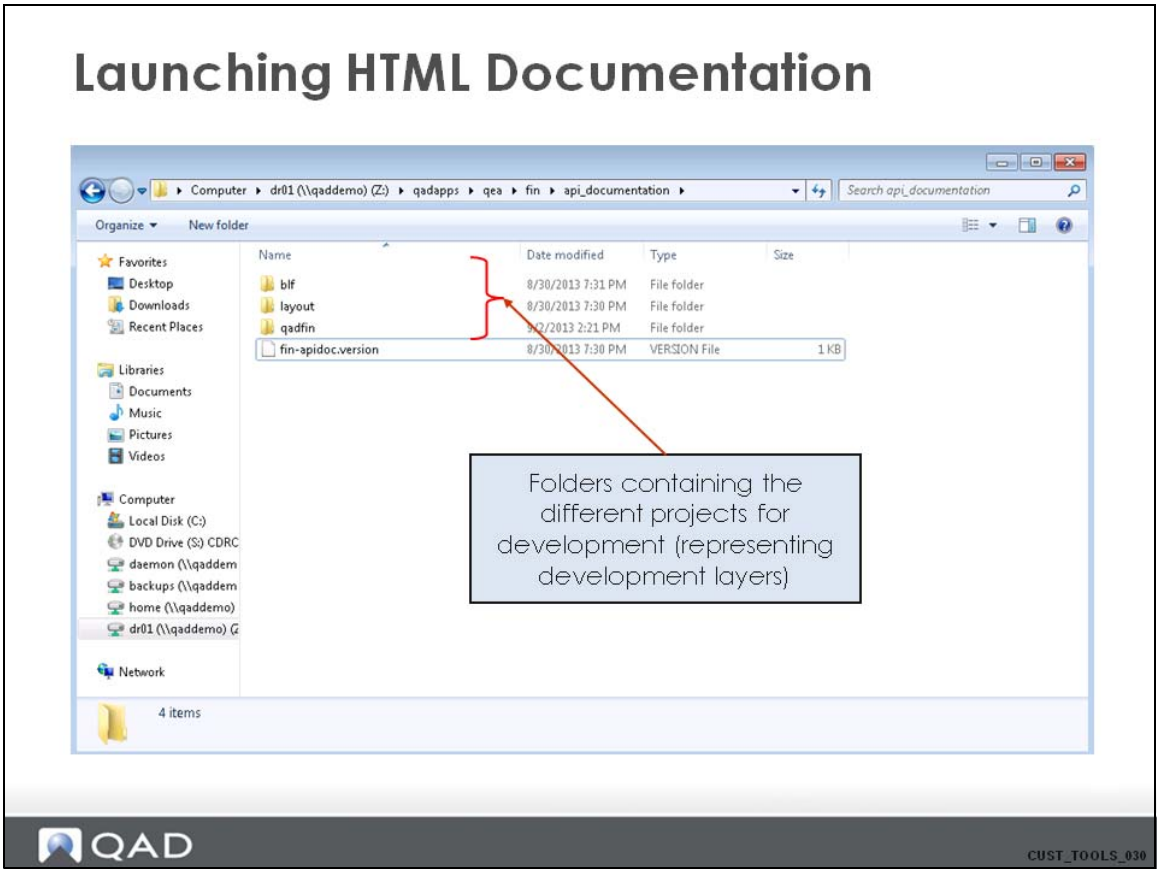
Non-intrusive Customization: Supporting Tools

BL Customization - Tools



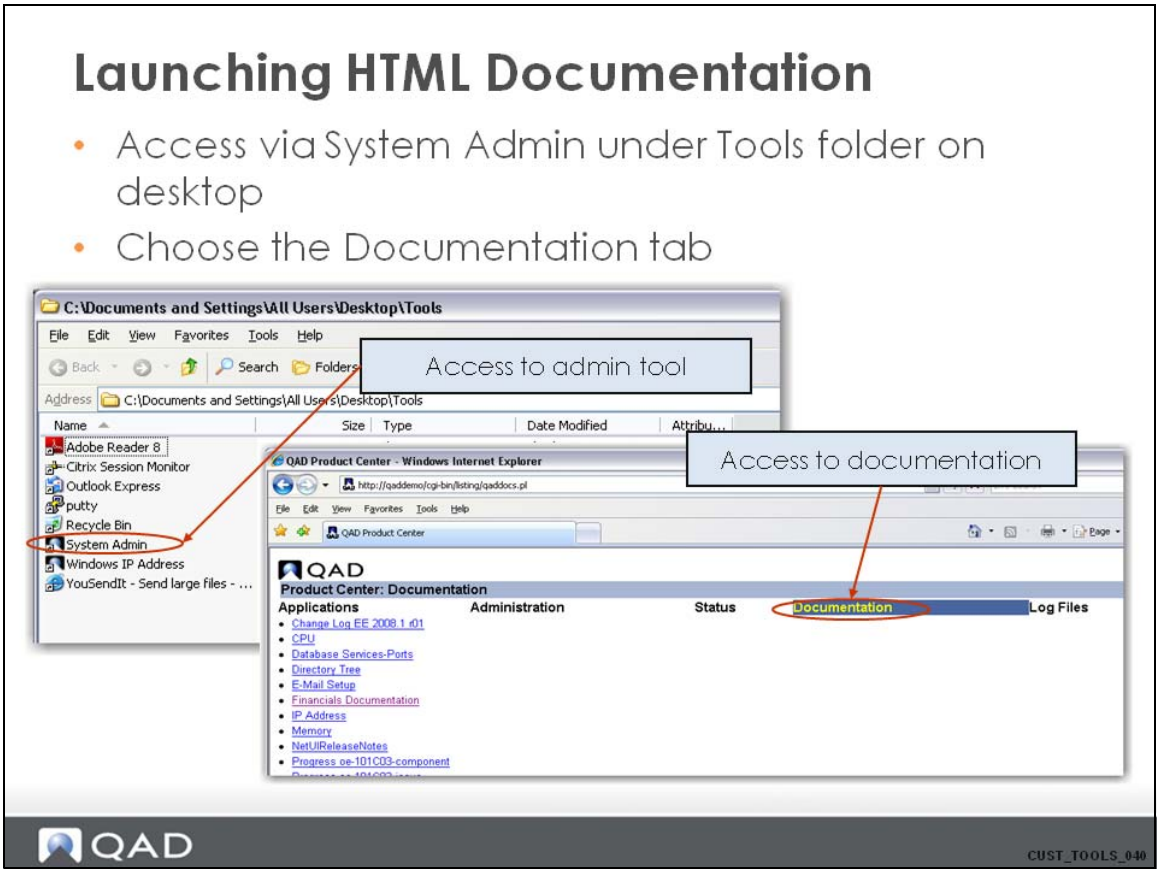
The HTML documentation represents the documented class model for the application business layer. The documentation is written by developers and generated in HTML by the development tool that they use (Component Builder tool).

Launching HTML Documentation



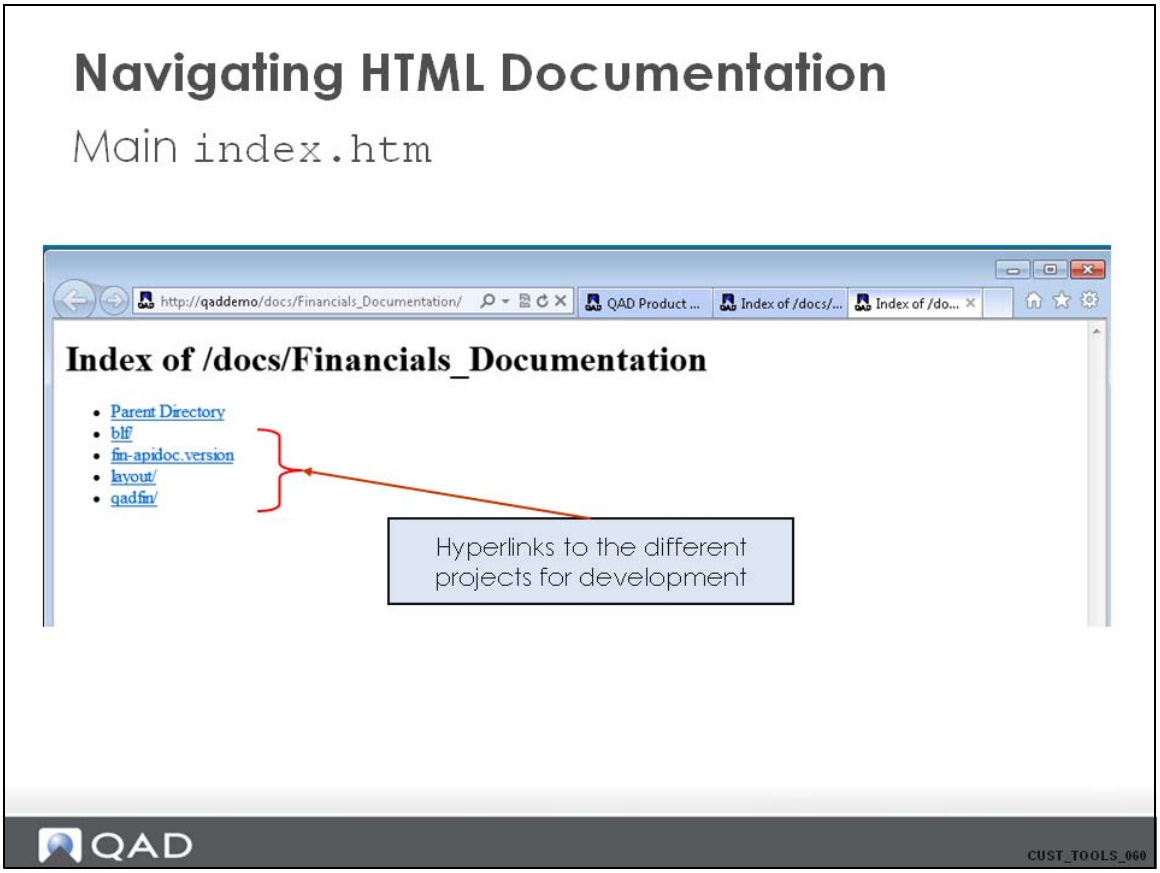
Launching the documentation from Windows.

Launching HTML Documentation



Launching the documentation from the admin tools.

Navigating HTML Documentation

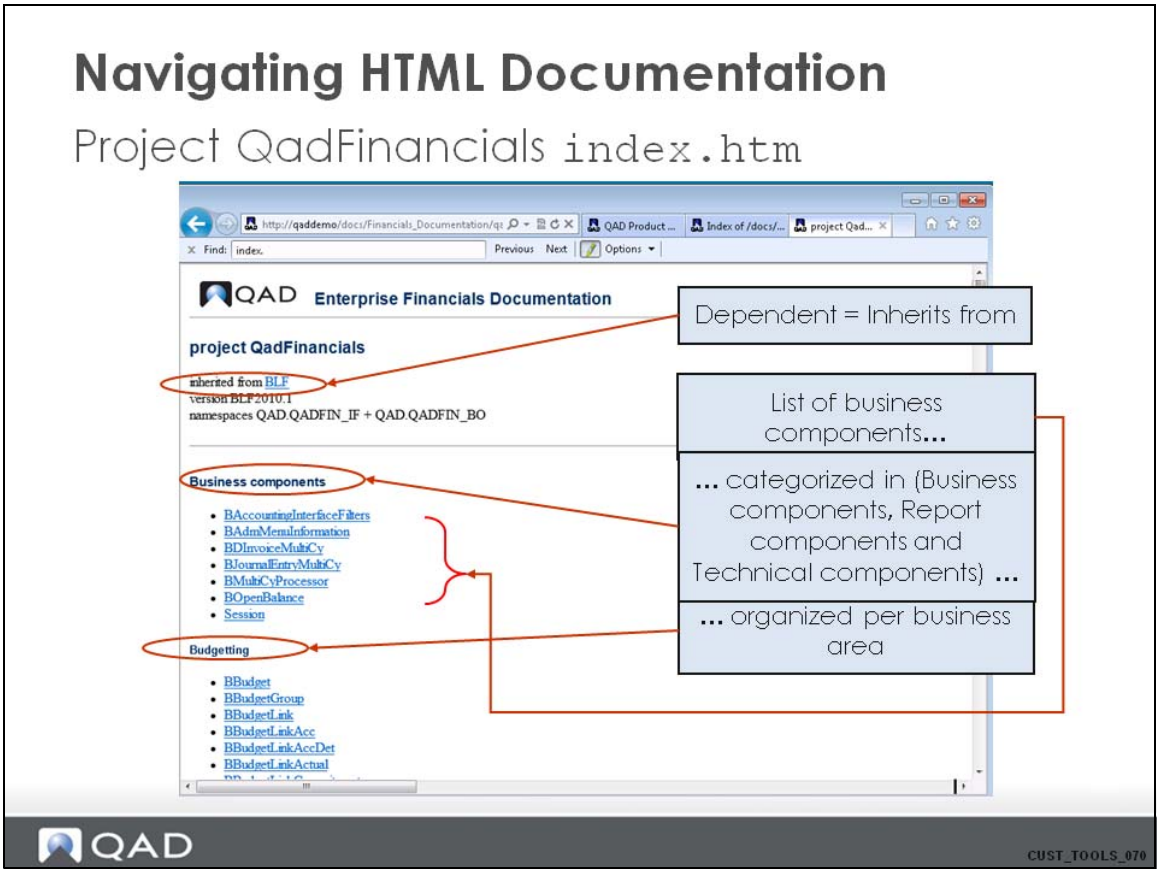


The main page for HTML documentation contains a hyperlink to each of the development projects that are used to create the final application code.

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

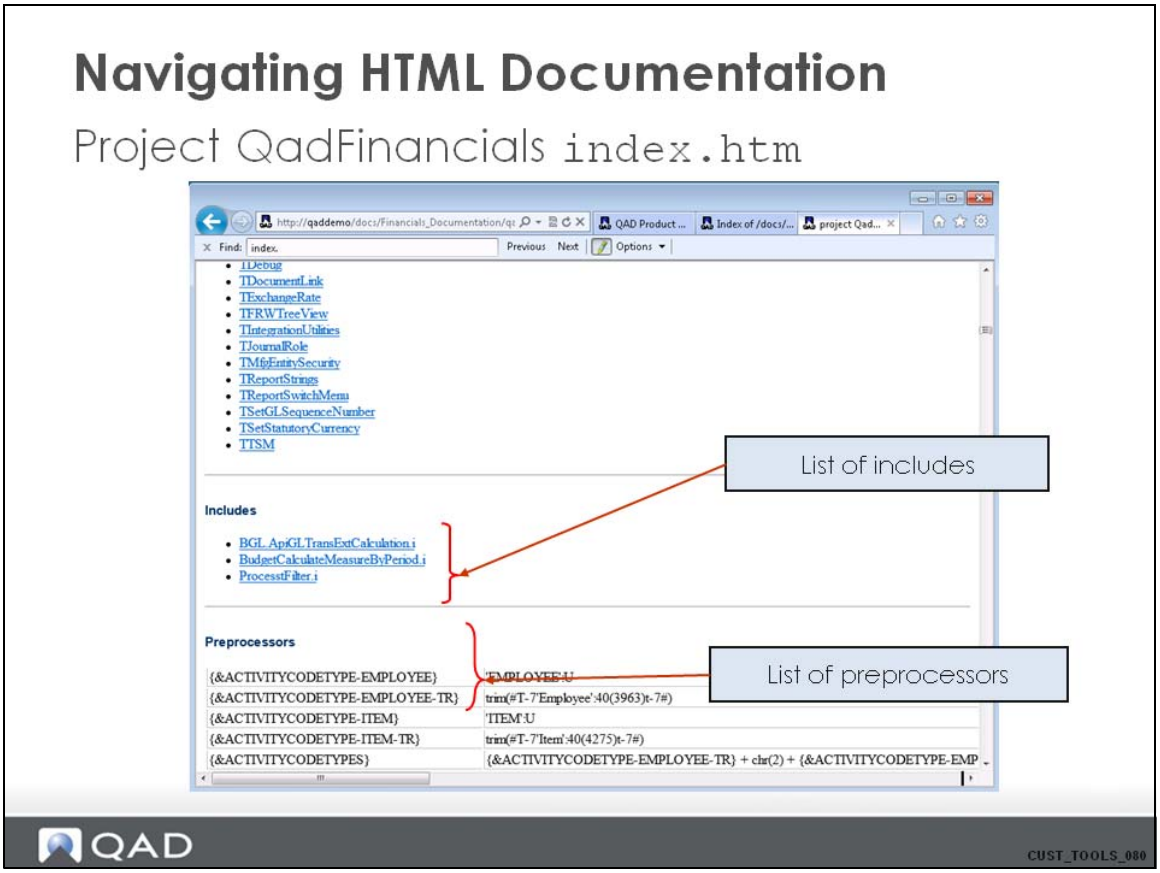


The main page for project documentation contains the following:

- 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



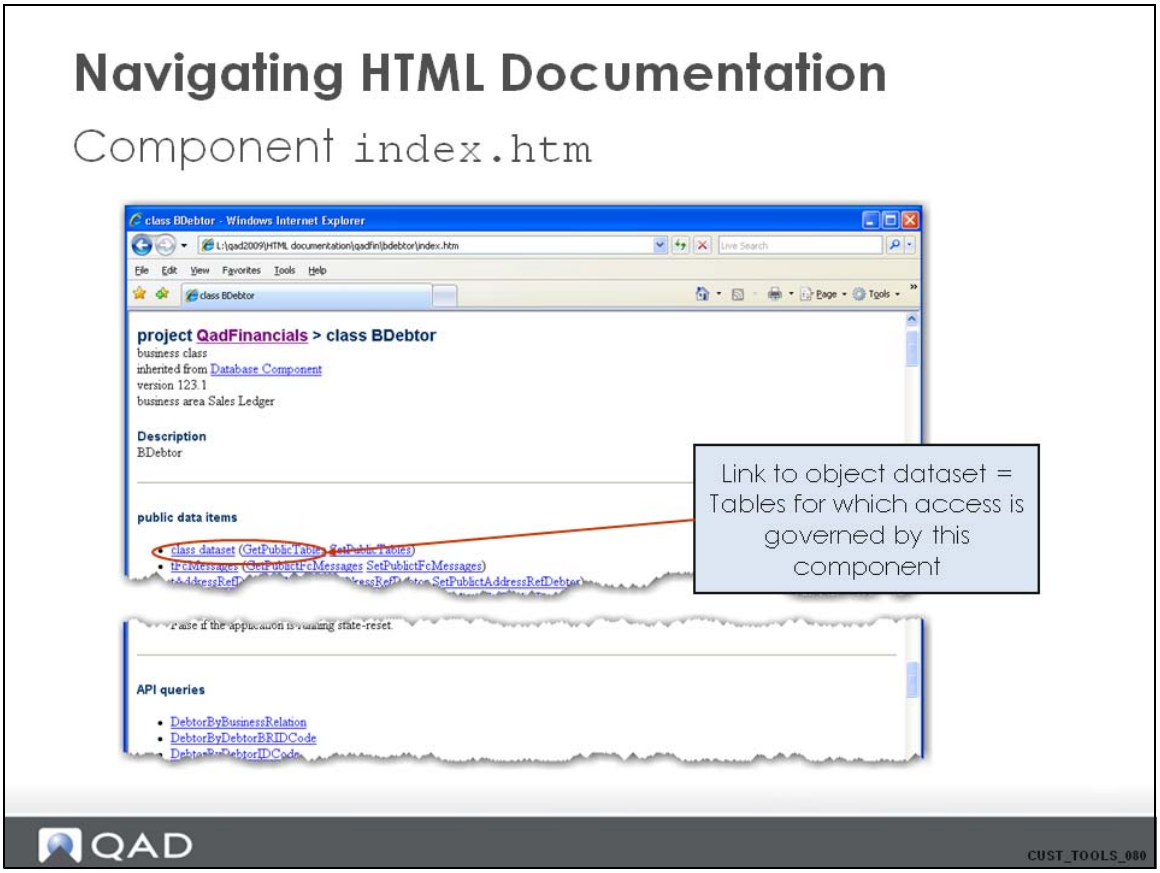
The main page for project documentation contains the following:

- …
- 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

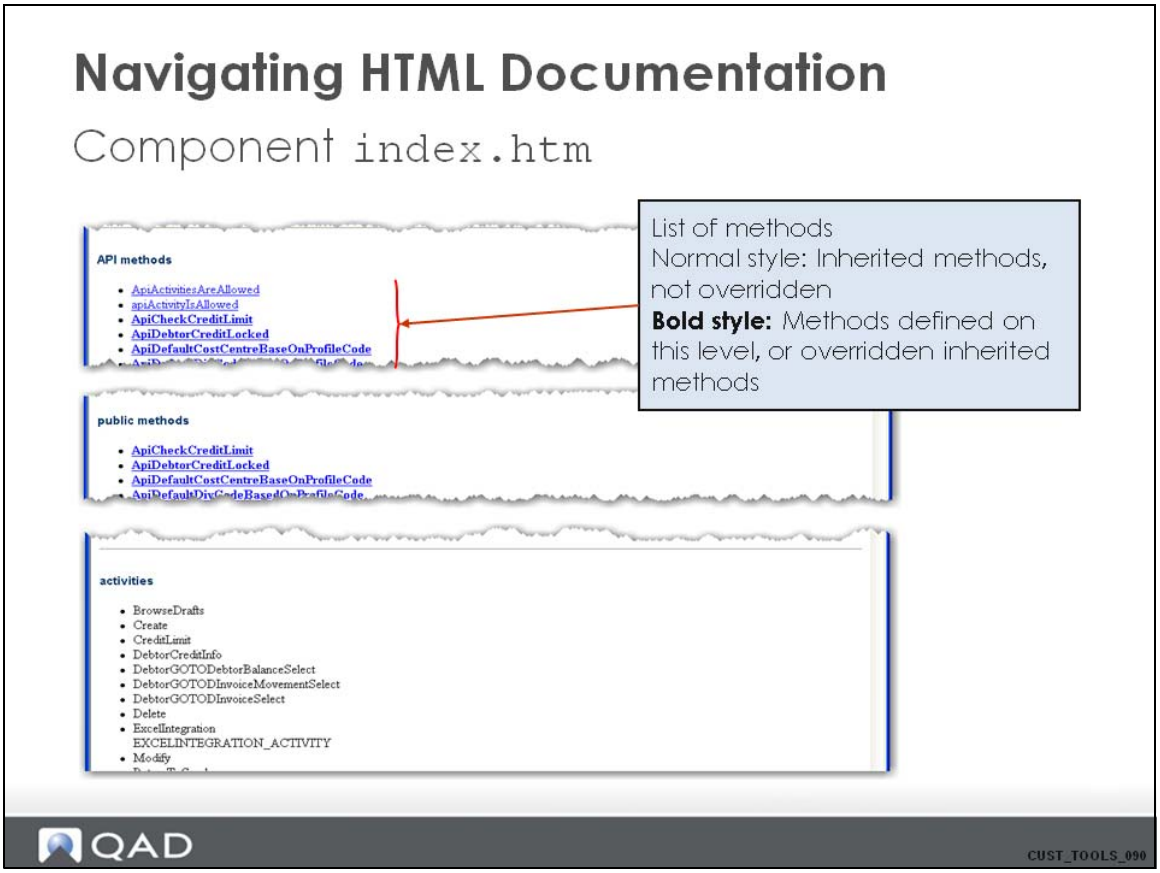


The main page for the class (component) documentation contains the following:

- 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



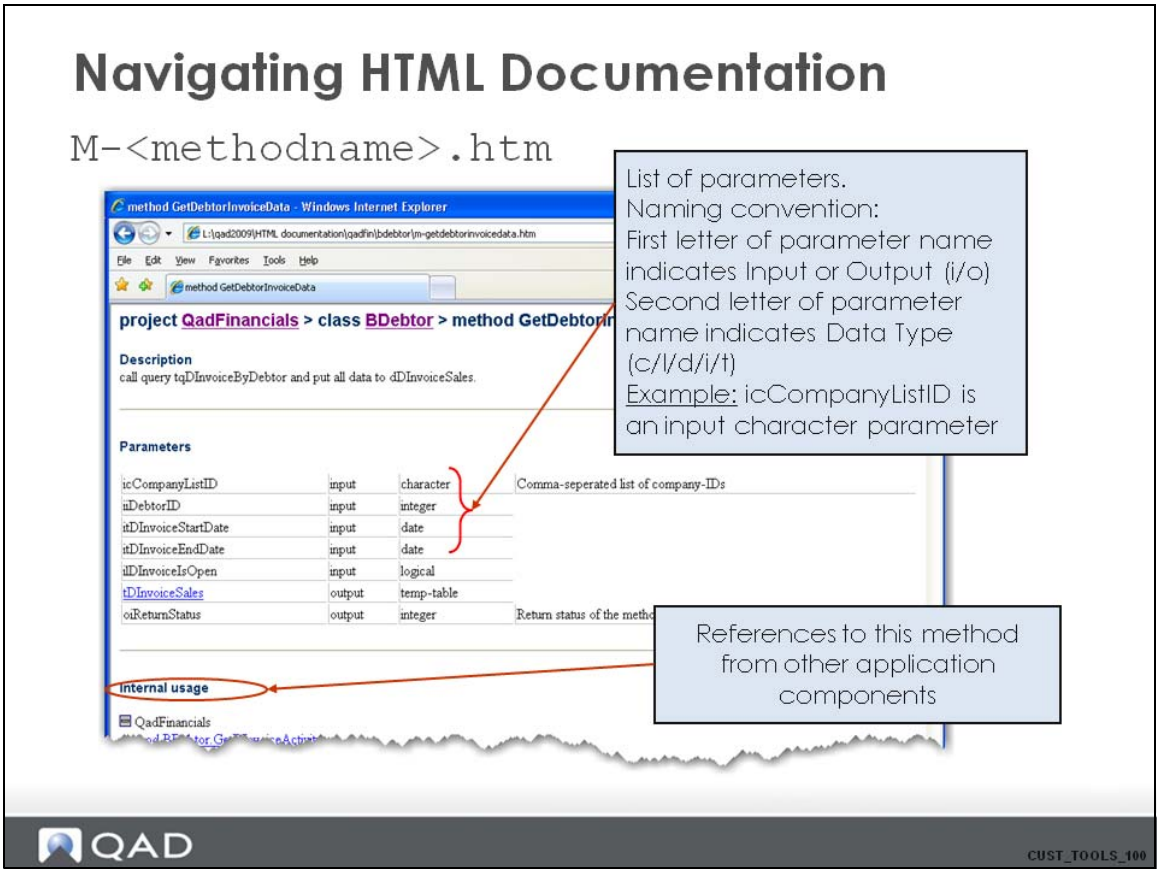
The main page for the class (component) documentation contains the following:

- 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.

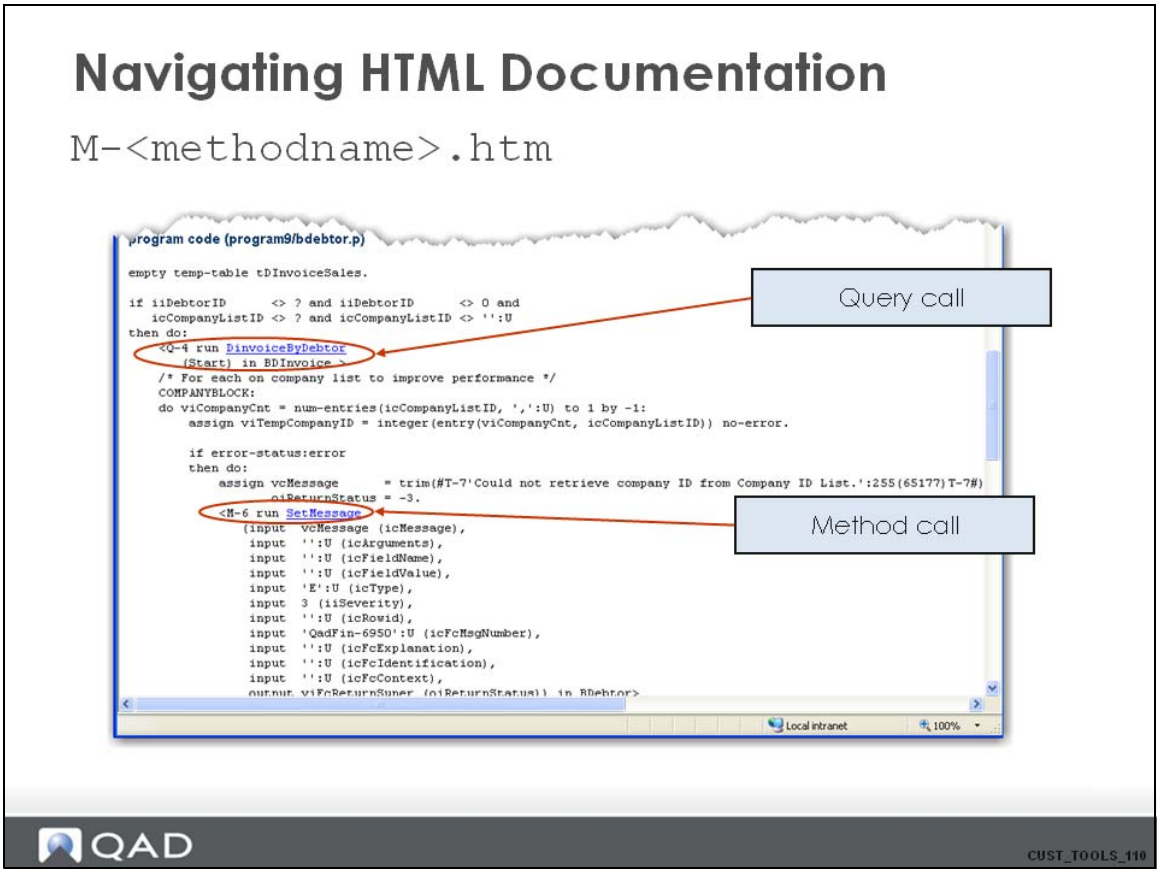
Navigating HTML Documentation



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

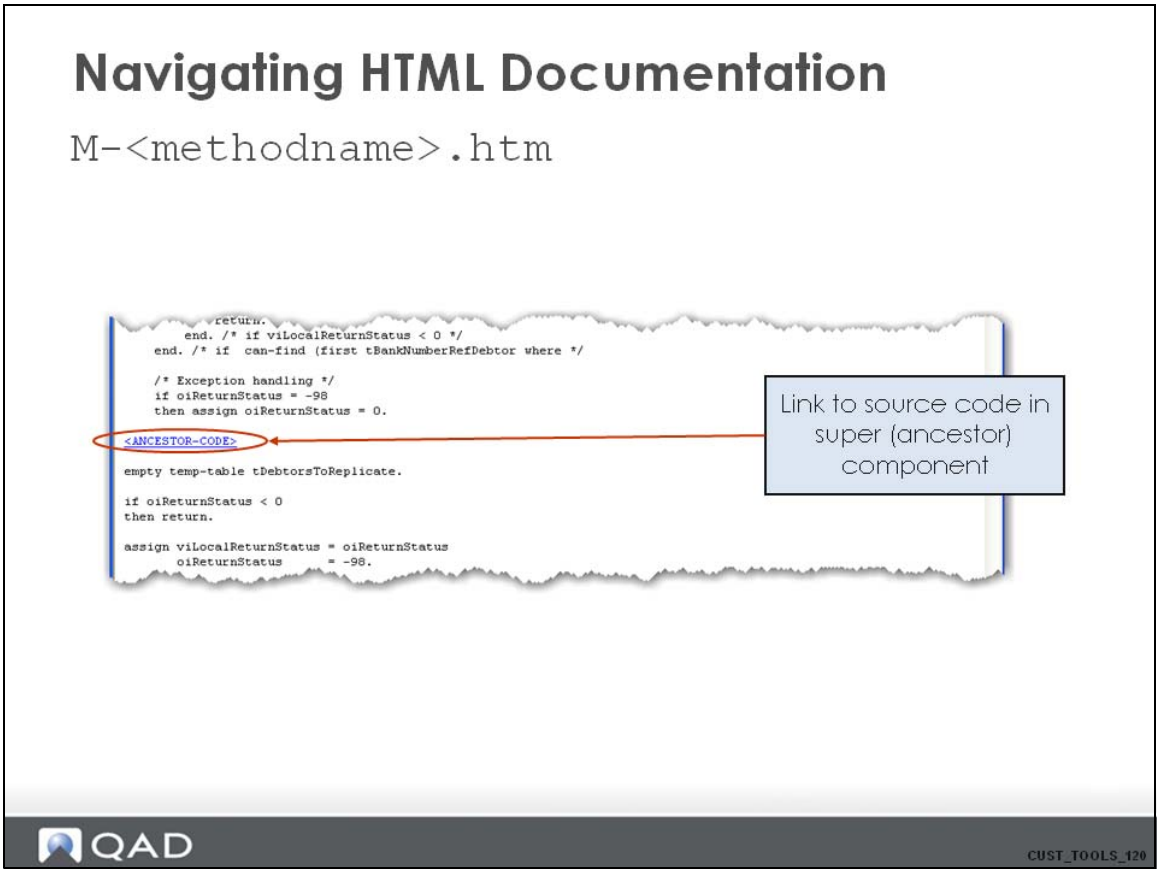


The method page contains the program code. The developer inserts this code as part of the implementation of the method in the Component Builder.

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

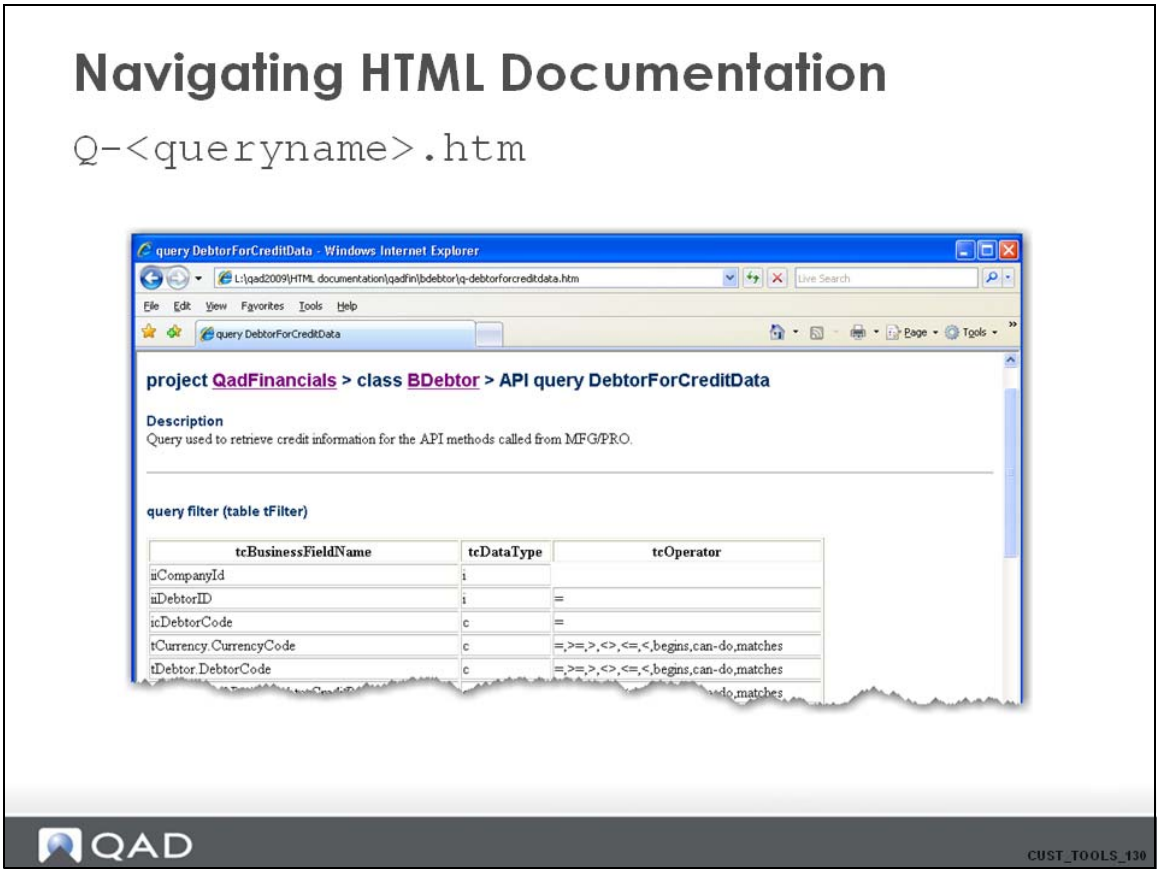
- Query calls. (<Q-…>) These represent execution of queries to retrieve data in records of a temp table. Values for the input parameters are also visible in the query call tag. The name of the query within the tag is hyperlinked and can be used to drill down to the specific documentation for the called 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



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.

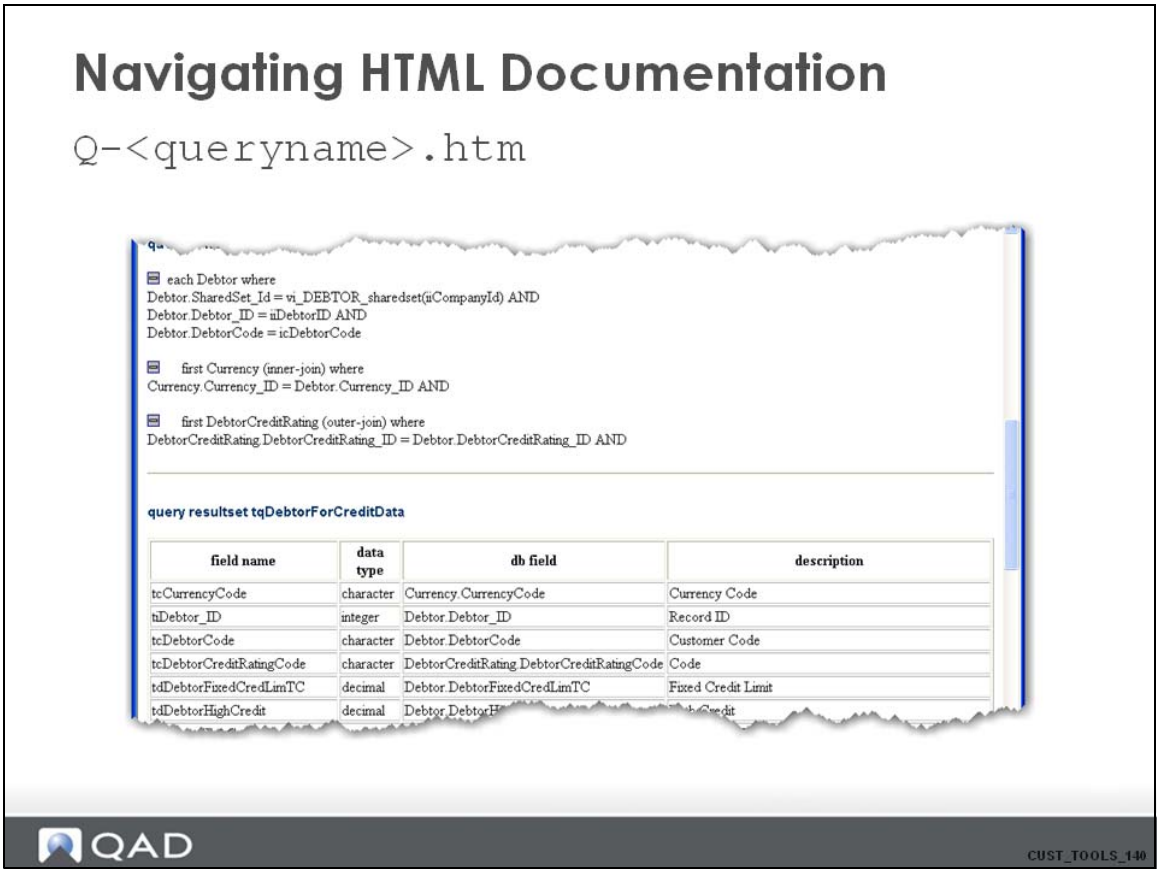
Navigating HTML Documentation



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.

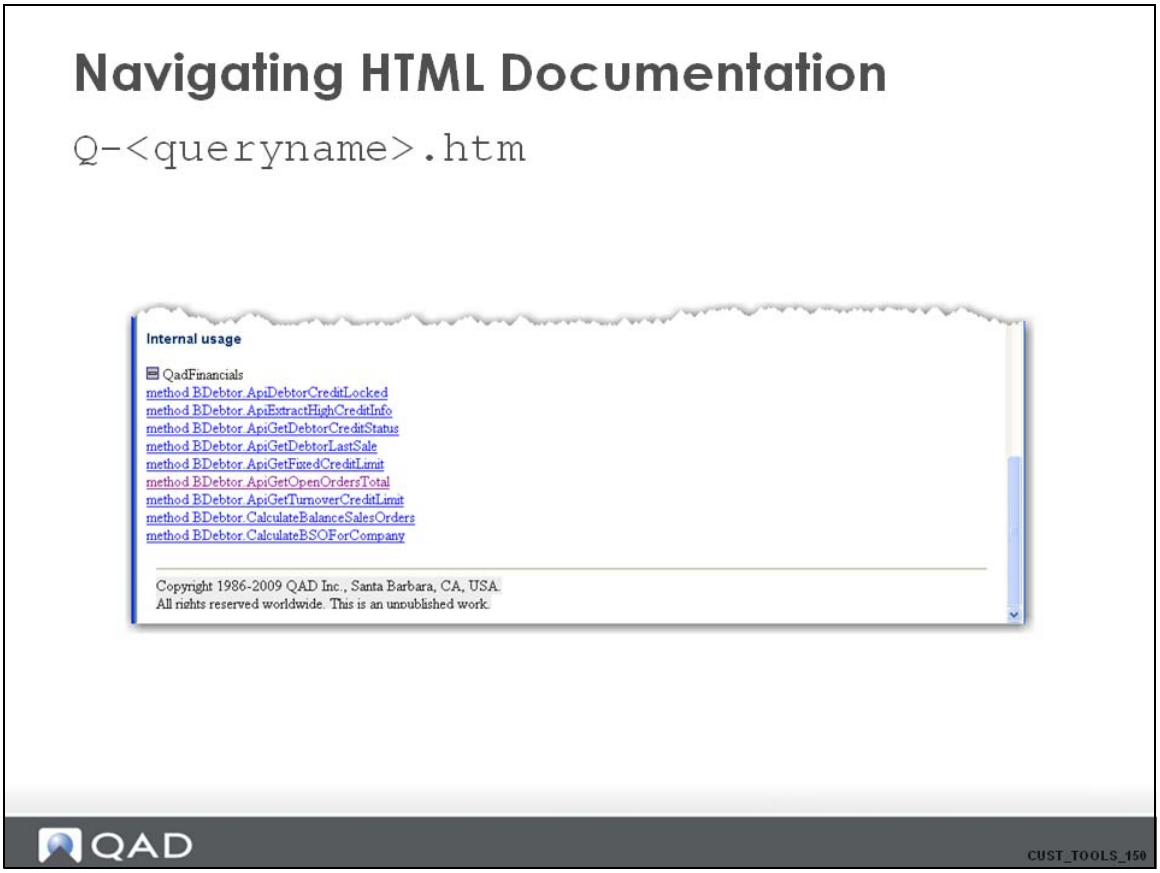
Navigating HTML Documentation



The query page contains the following information:

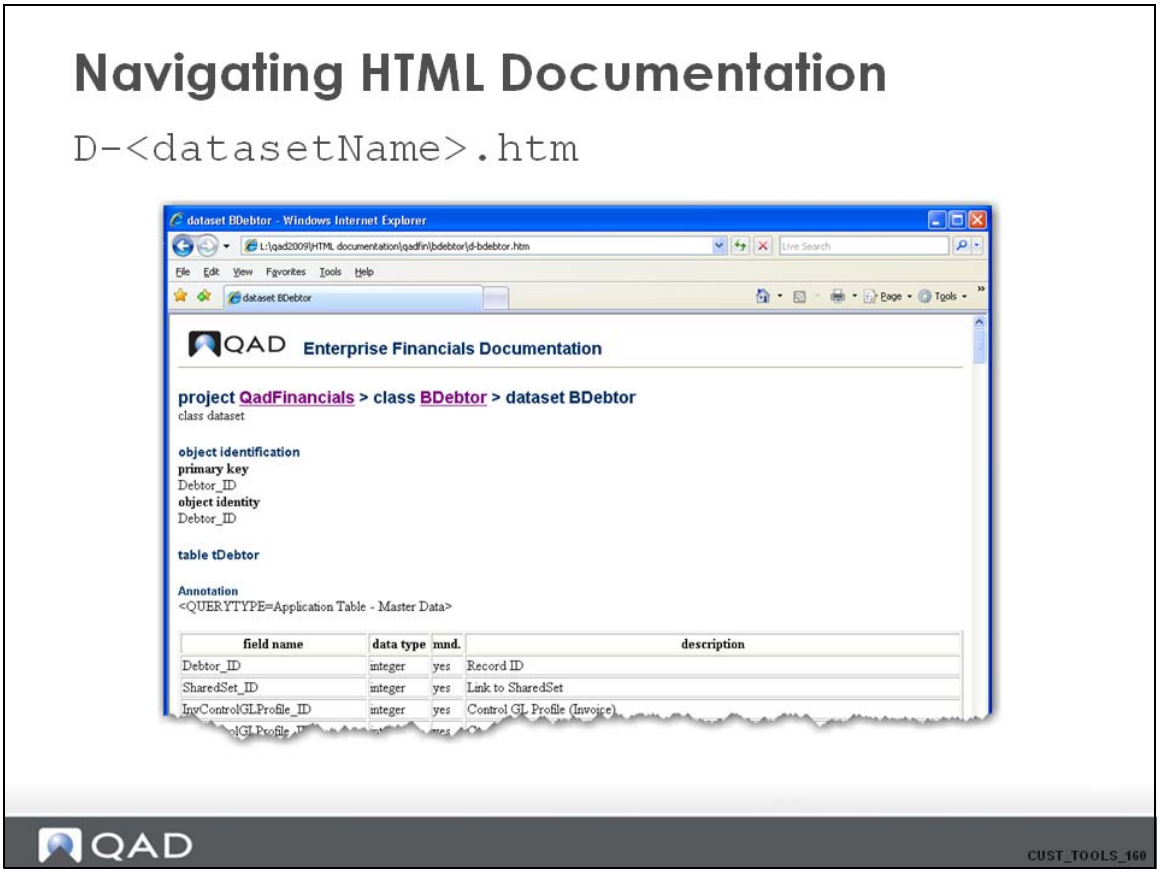
- The query (condition).
- The description of the query result set.

Navigating HTML Documentation



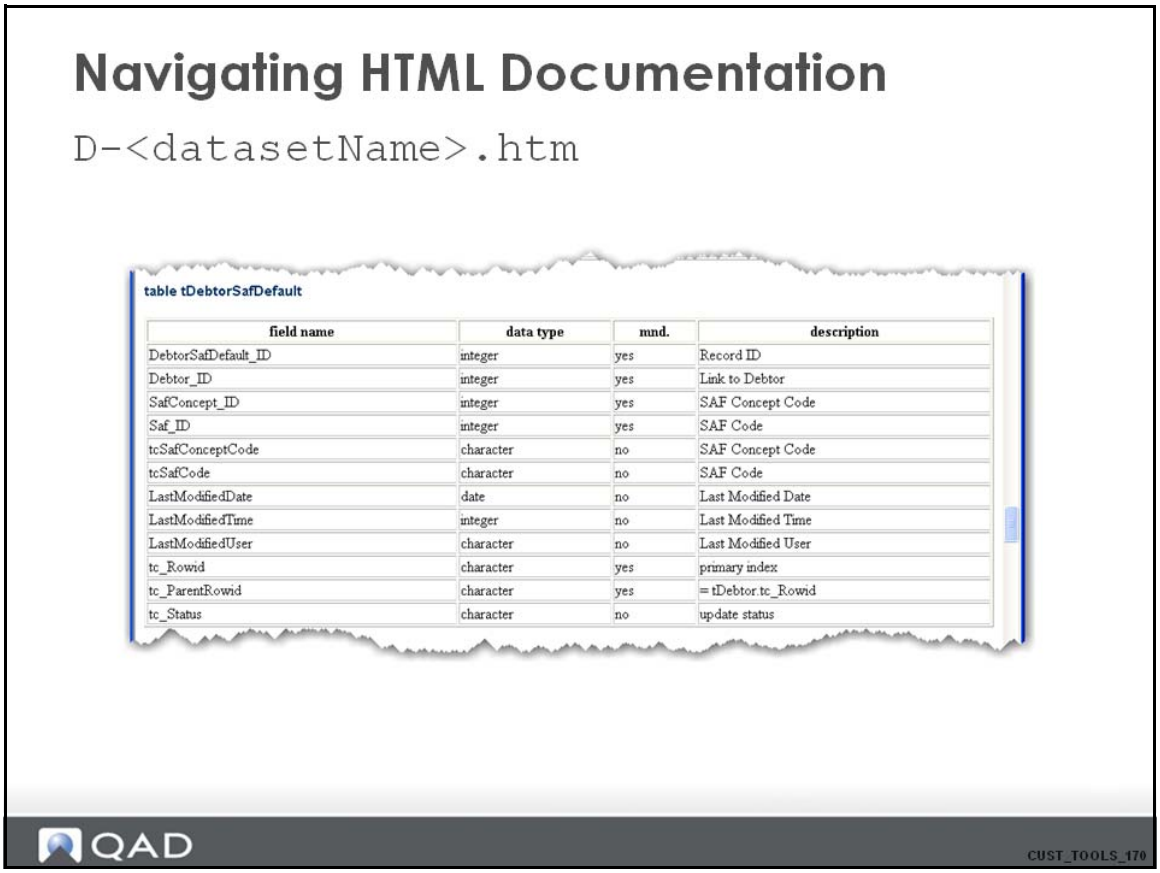
The query page contains the places in the code that reference (call) the query.

Navigating HTML Documentation



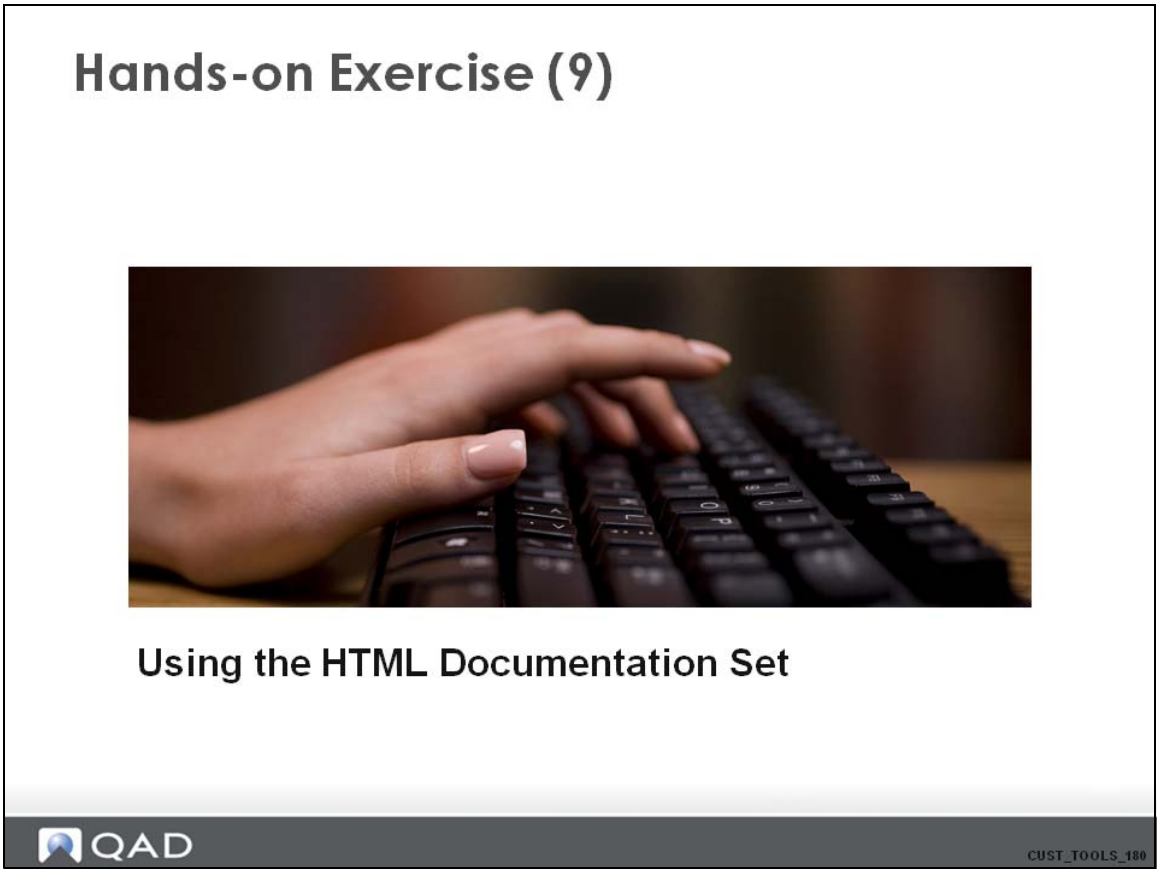
The dataset page contains the list of fields for each table in the dataset.

Navigating HTML Documentation

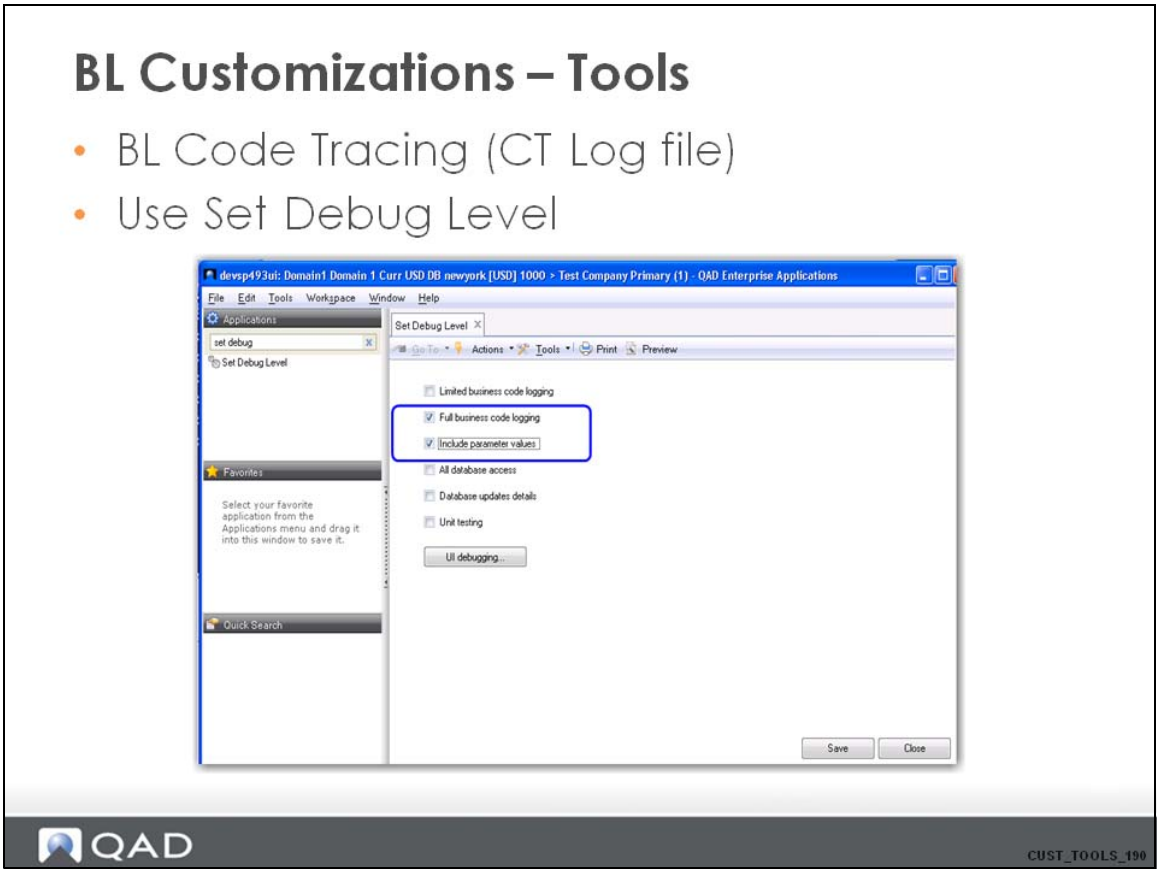


For an object dataset (for example, BDebtor), the tc_parentrowid shows the relation to the parent table in the object dataset.

Hands-on Exercise (9)



BL Customizations - Tools

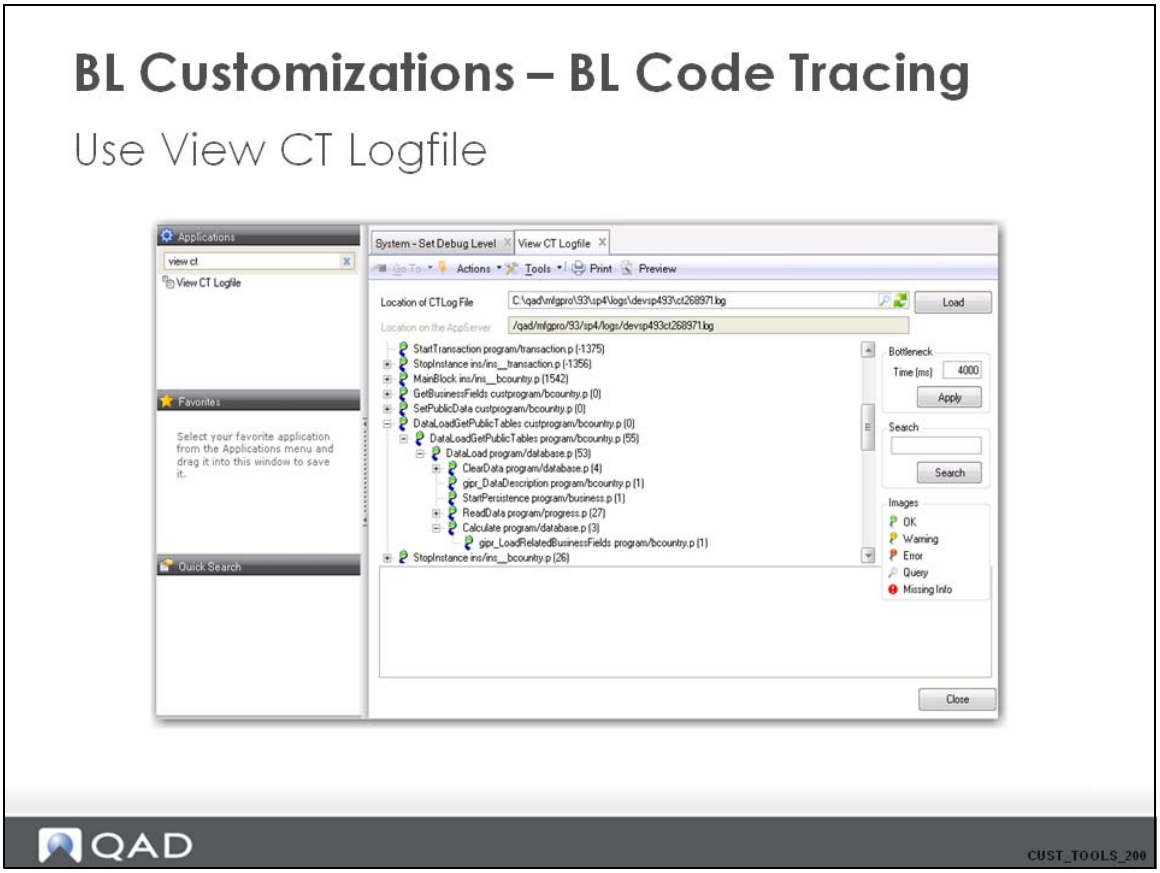


The ability to have a full code execution trace on the backend BL is indispensable for more complex customization work. All components follow the same standard patterns for the normal maintenance functionality, and for most other functions too. But some more complex components have many sub-flows or 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.

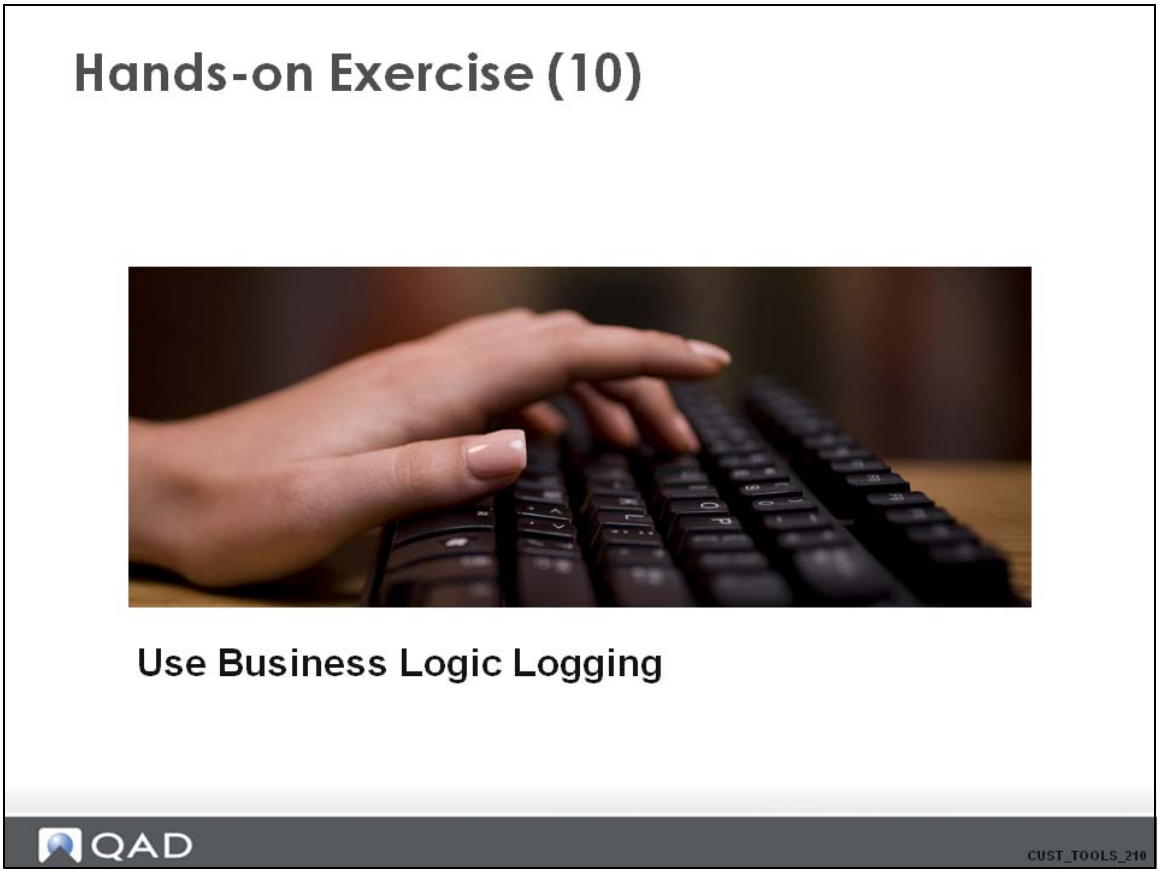
When the logging is on, a file named `ct<sessionID>.log` is created on the server. All methods and queries executed on the backend are logged.

BL Customizations - BL Code Tracing

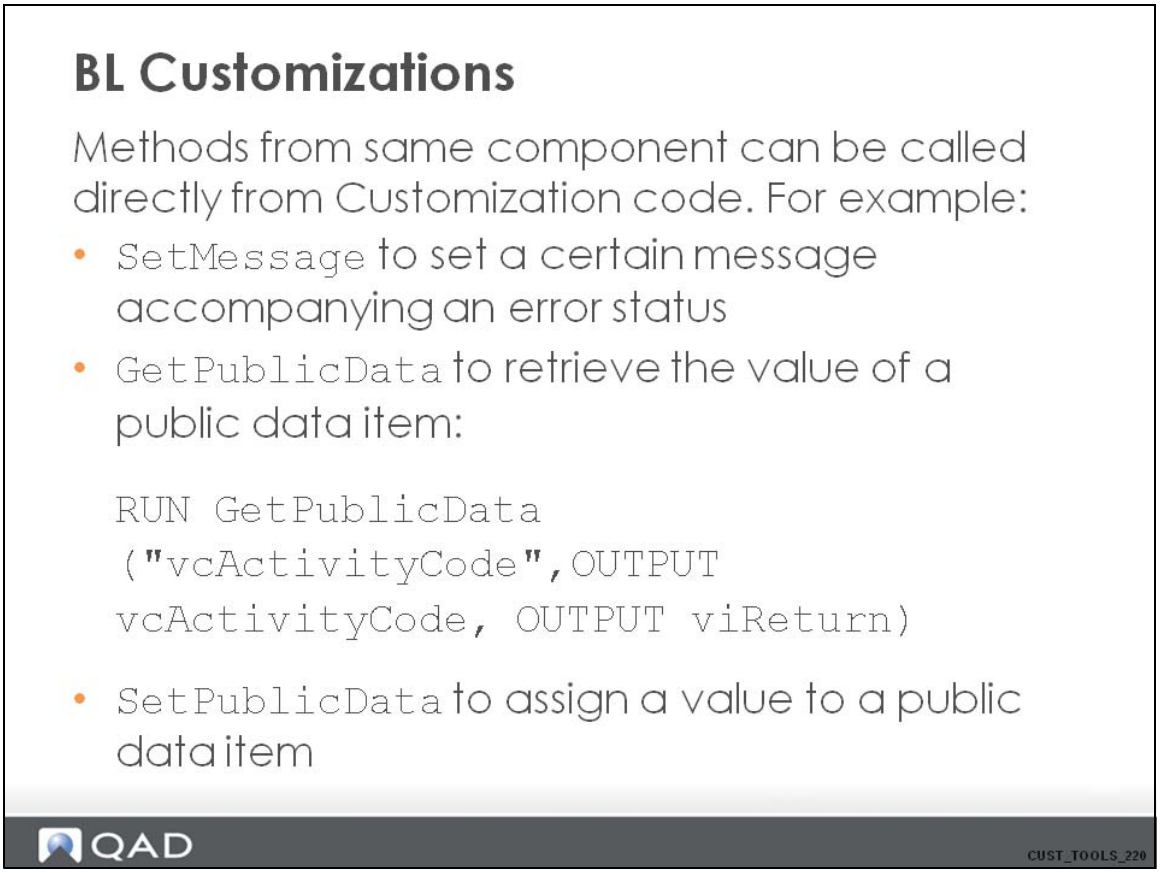


To view the contents of the current CT log file (containing the business code trace), use “View CTLog” from the menu.

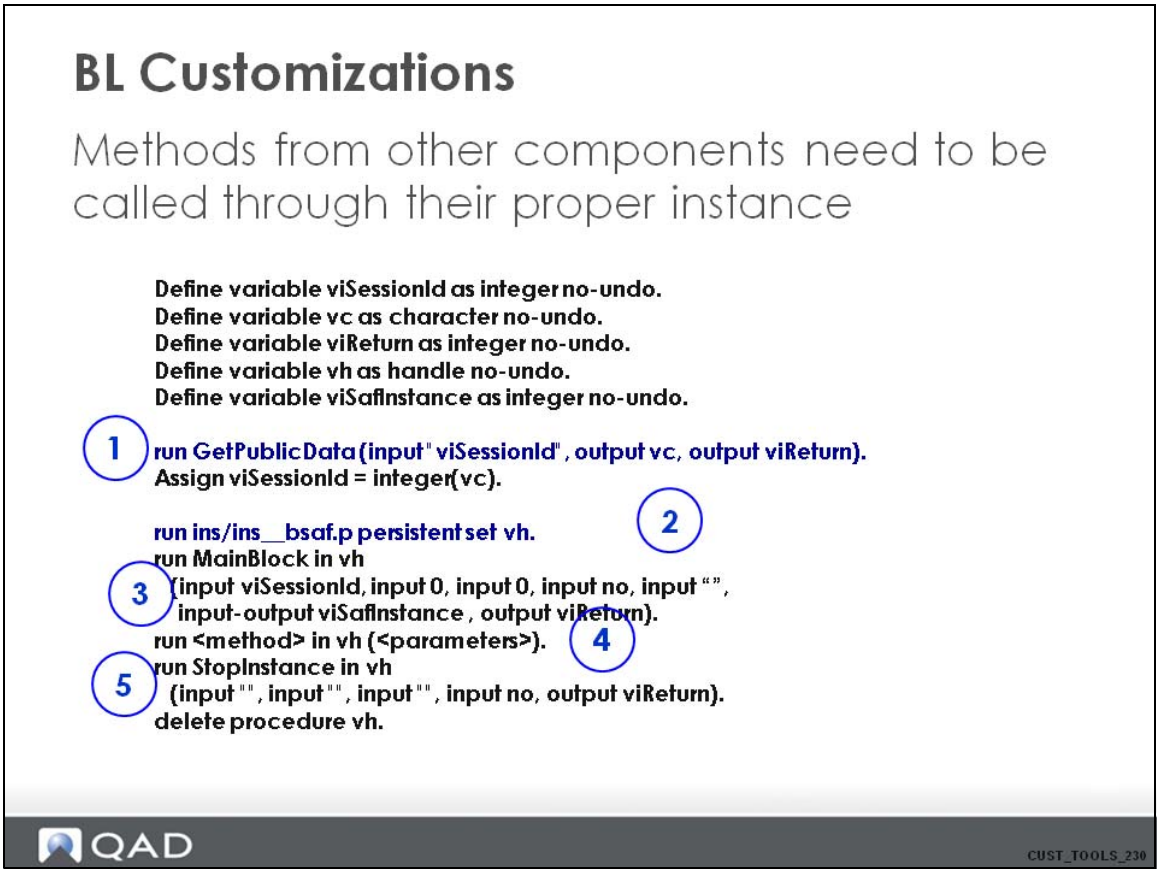
Hands-on Exercise (10)



BL Customizations



BL Customizations



1. Retrieve the current session ID. This is required to start an instance of the business component.

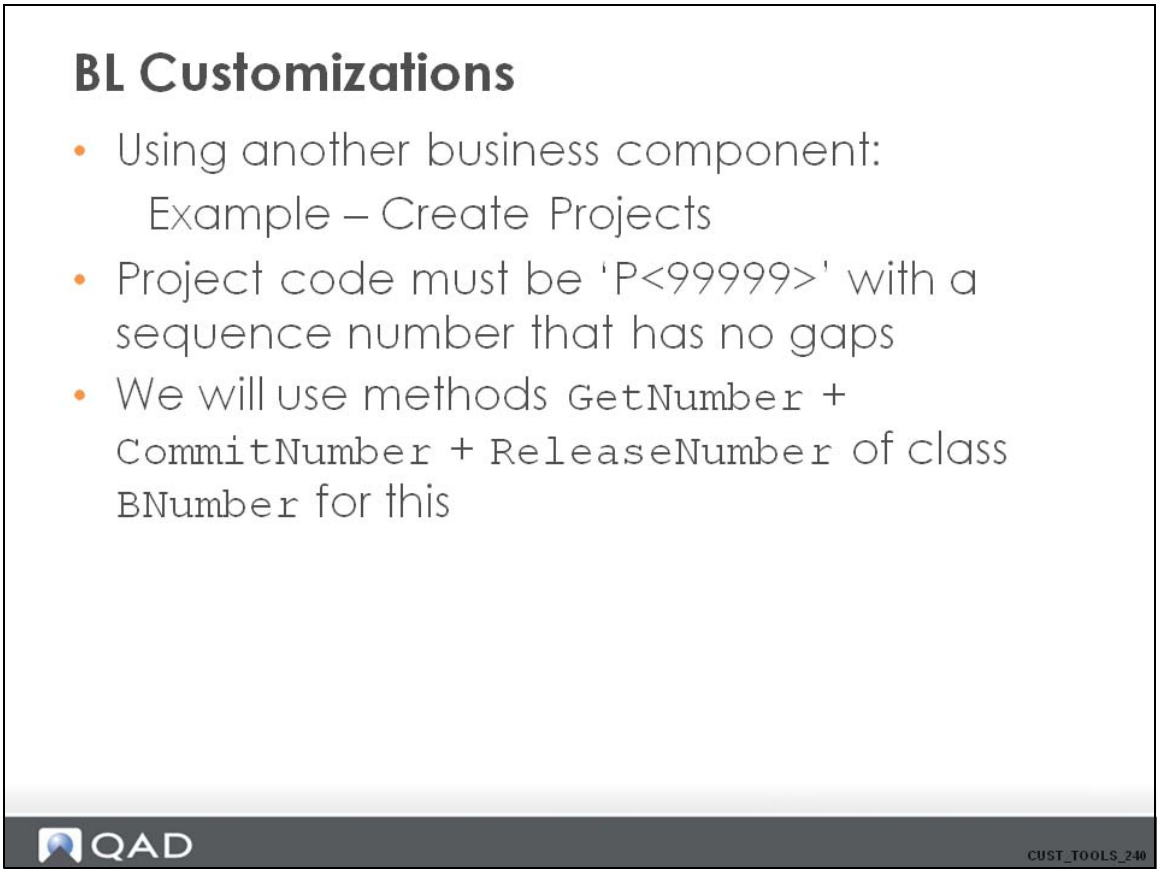
2. Run the instance procedure of the business component persistently to make sure that the methods become available.

3. Run the MainBlock in the instance procedure. This ensures that the state of the business component instance is OK. Remark the viSafIntance. 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)!

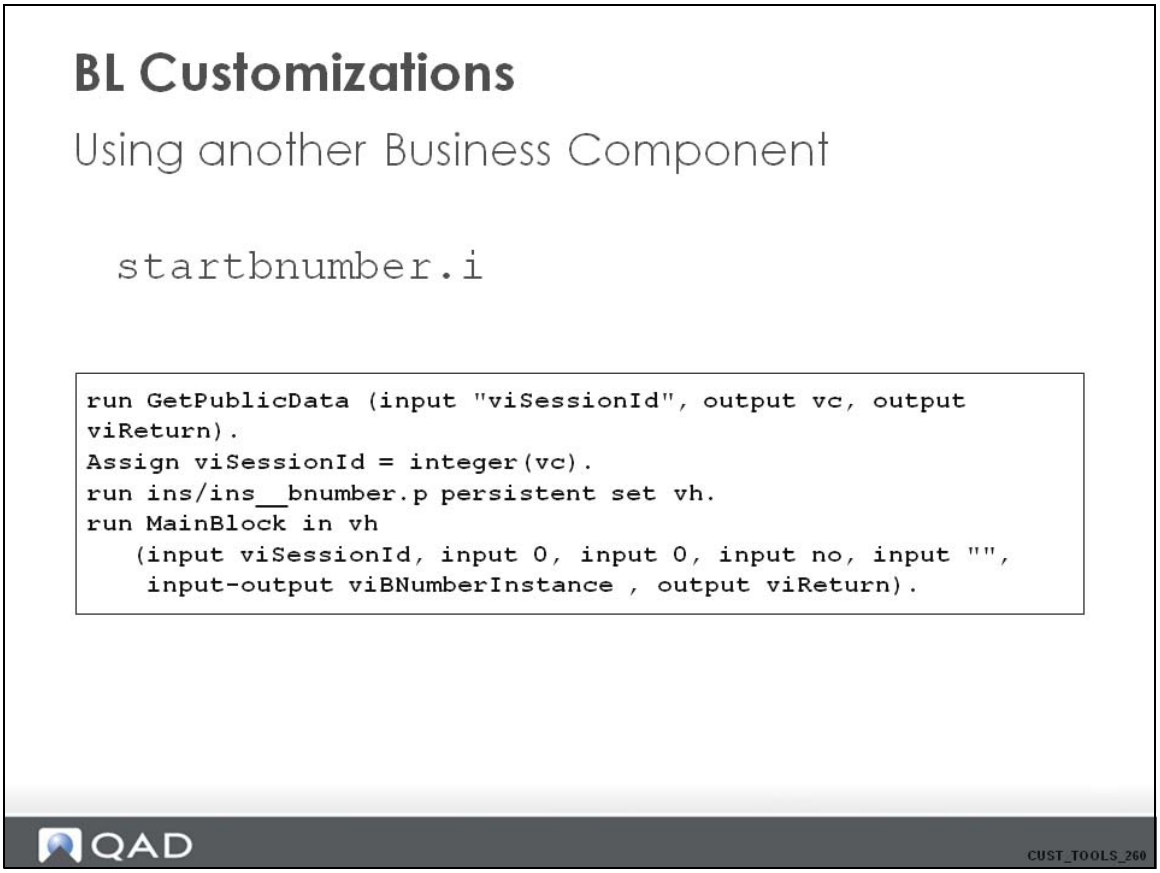
BL Customizations



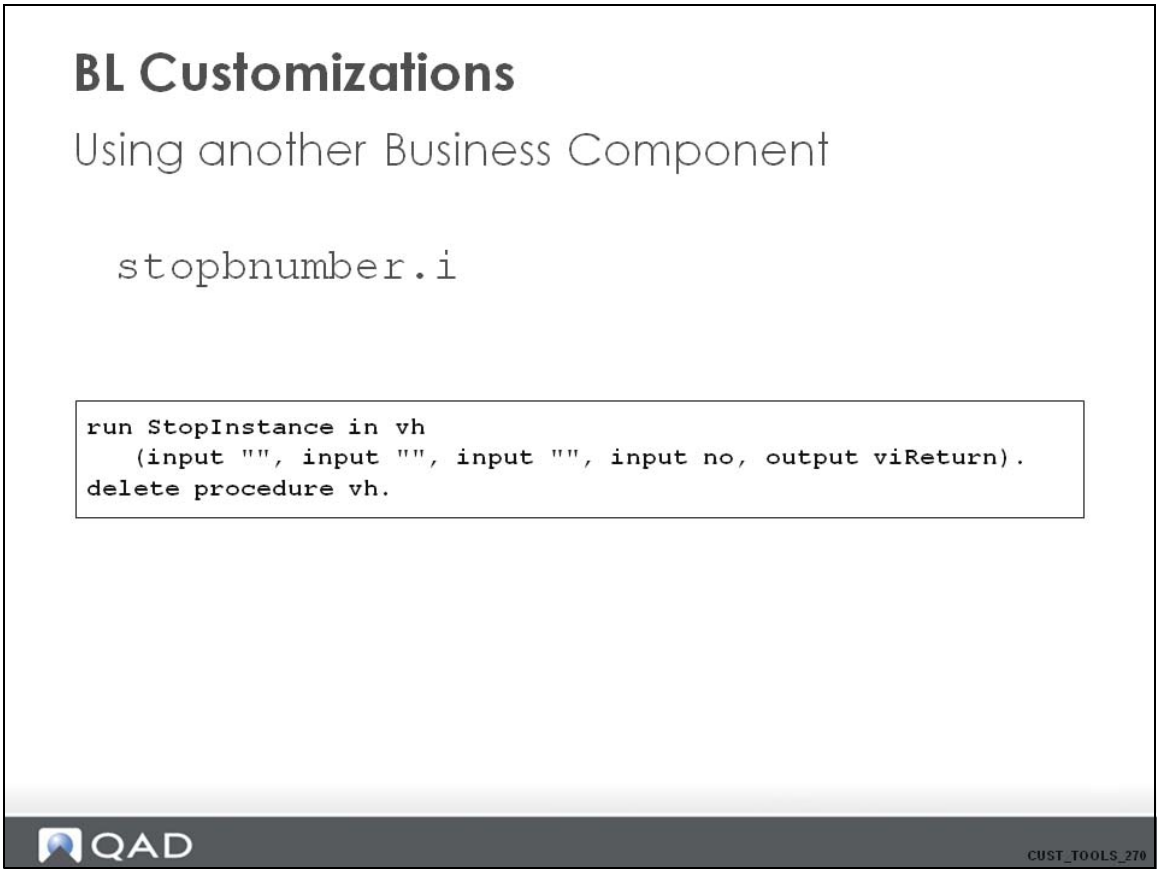
For this case, we assume that projects cannot be deleted and that projects are not created through QXtend or other integration. Read the HTML documentation on the business methods used.

BL Customizations

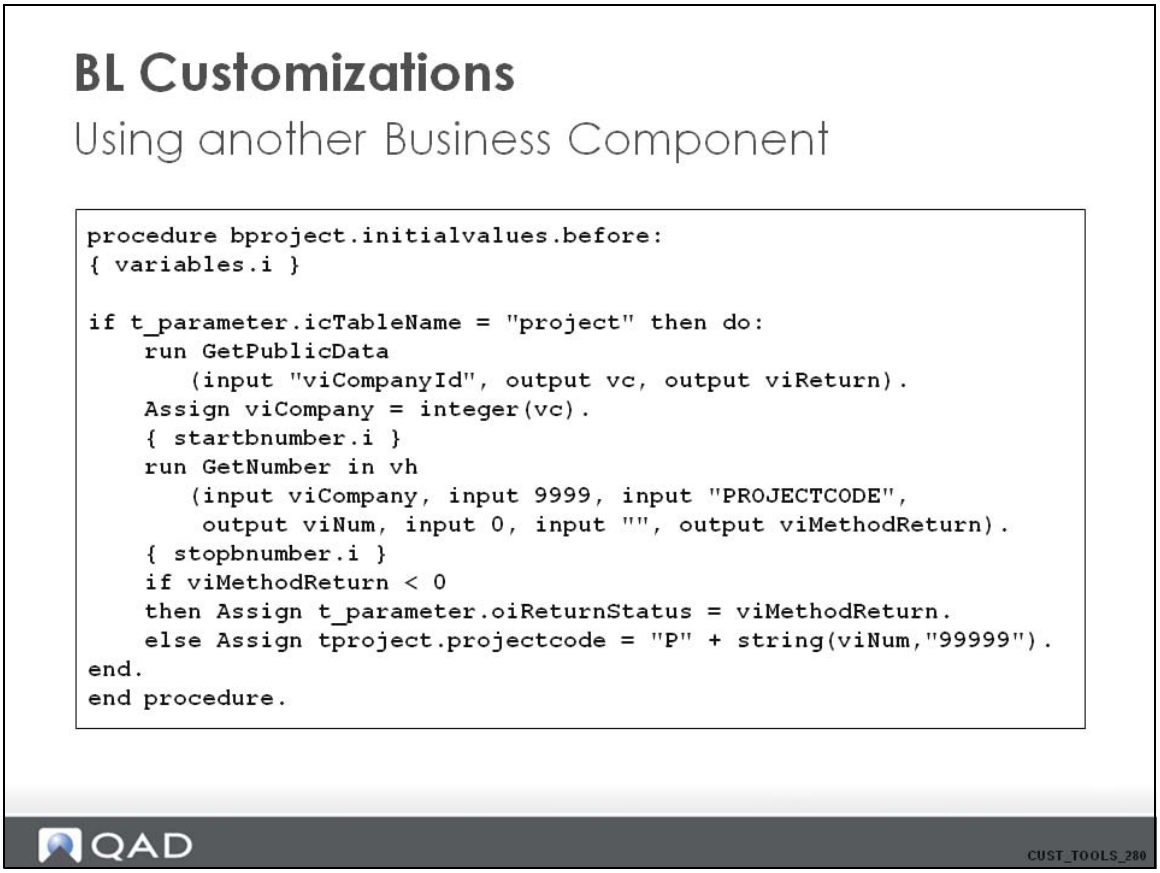
BL Customizations



BL Customizations

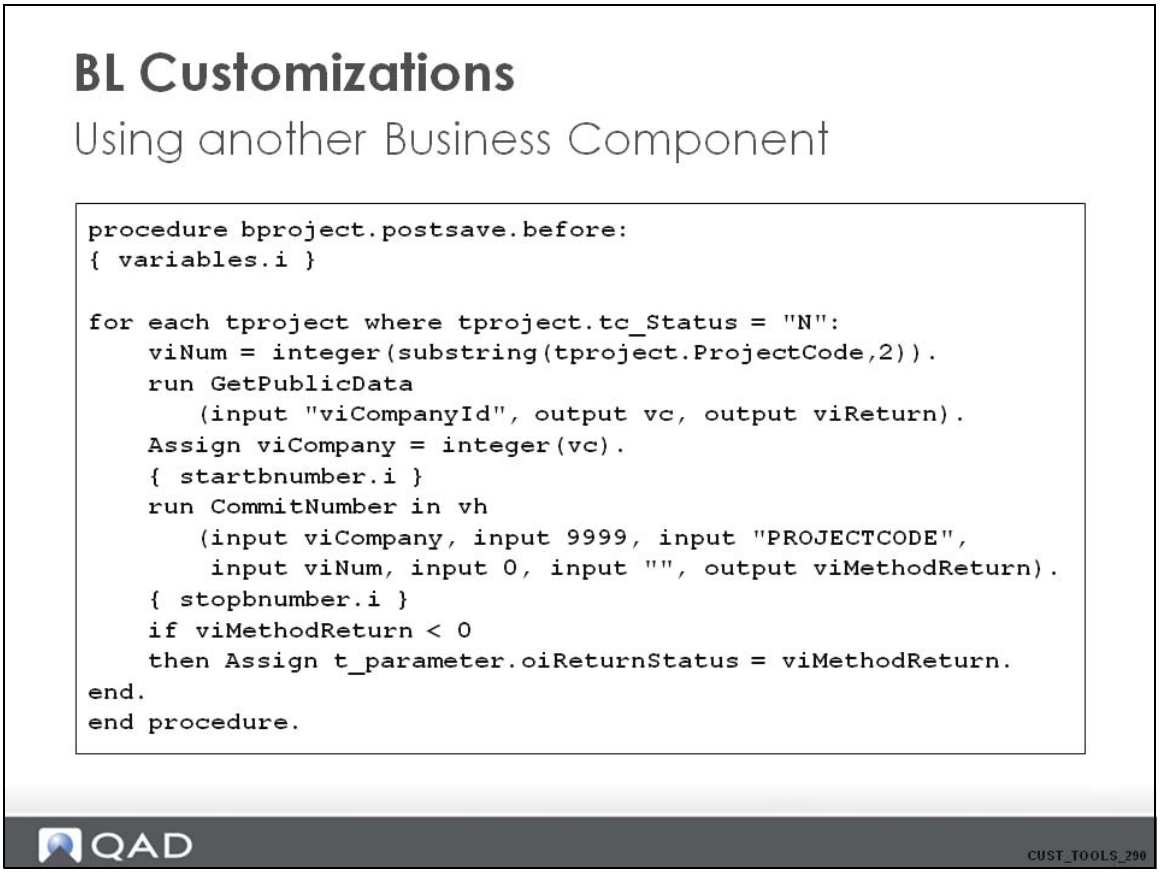


BL Customizations



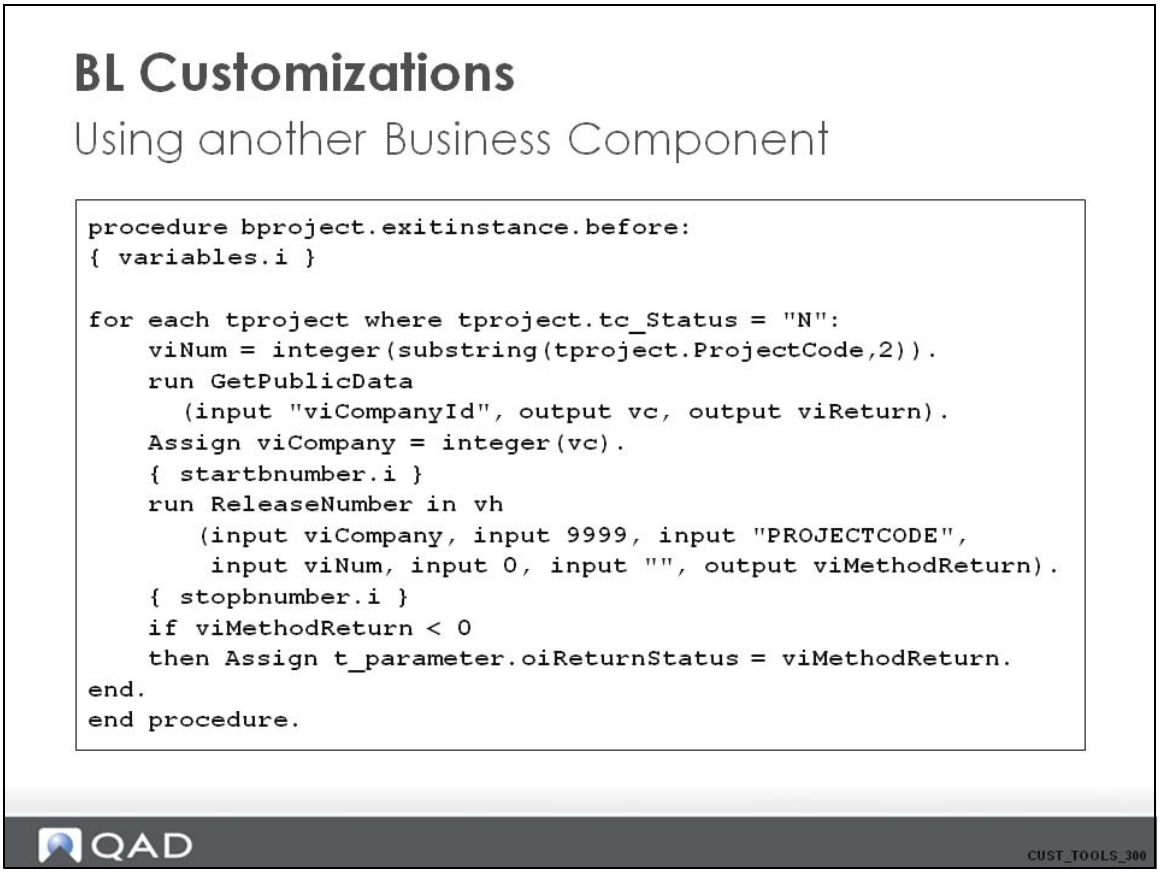
GetNumber reserves the project code number; if another user starts project create before we save our project, the user does not get the same number.

BL Customizations



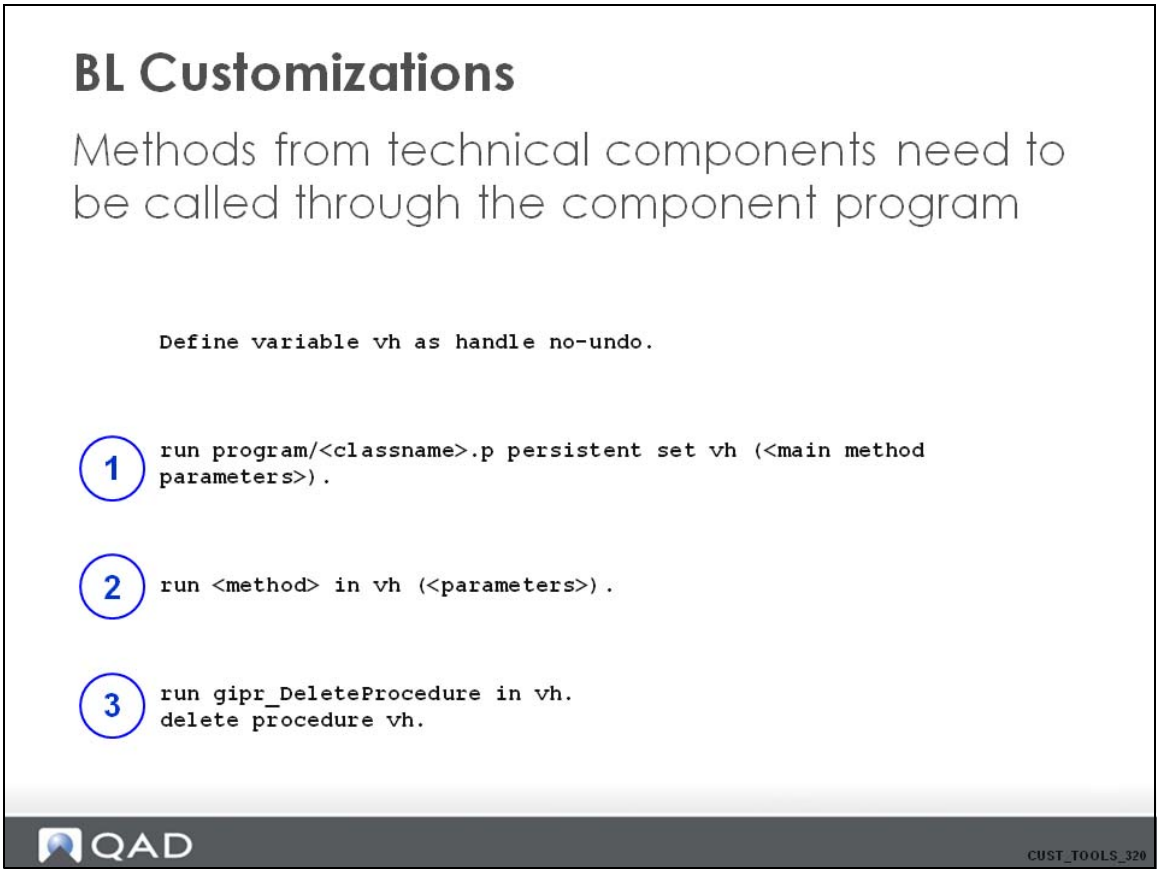
CommitNumber confirms that the number was used and is never used again.

BL Customizations



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

BL Customizations



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.

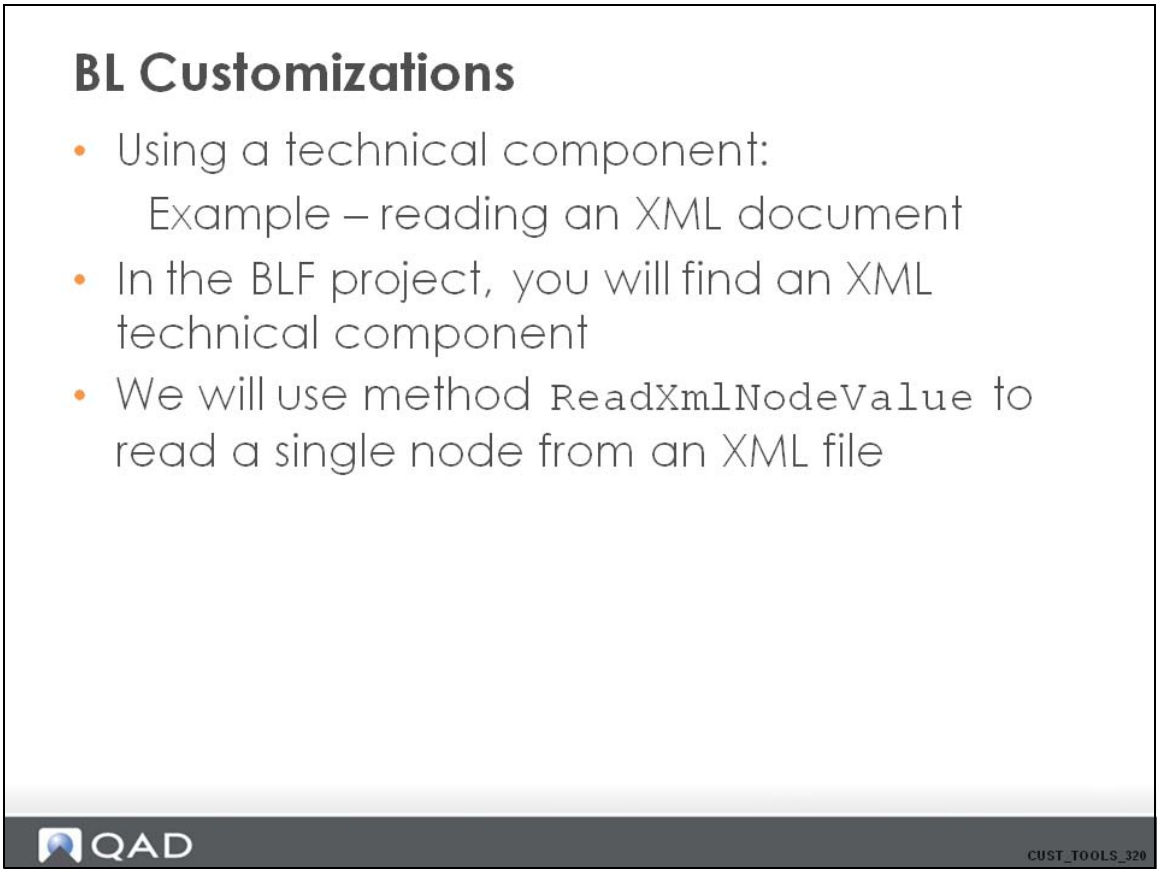
Some exceptions could be exposed to run from an external consumer, but this must be specifically written and documented.

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.

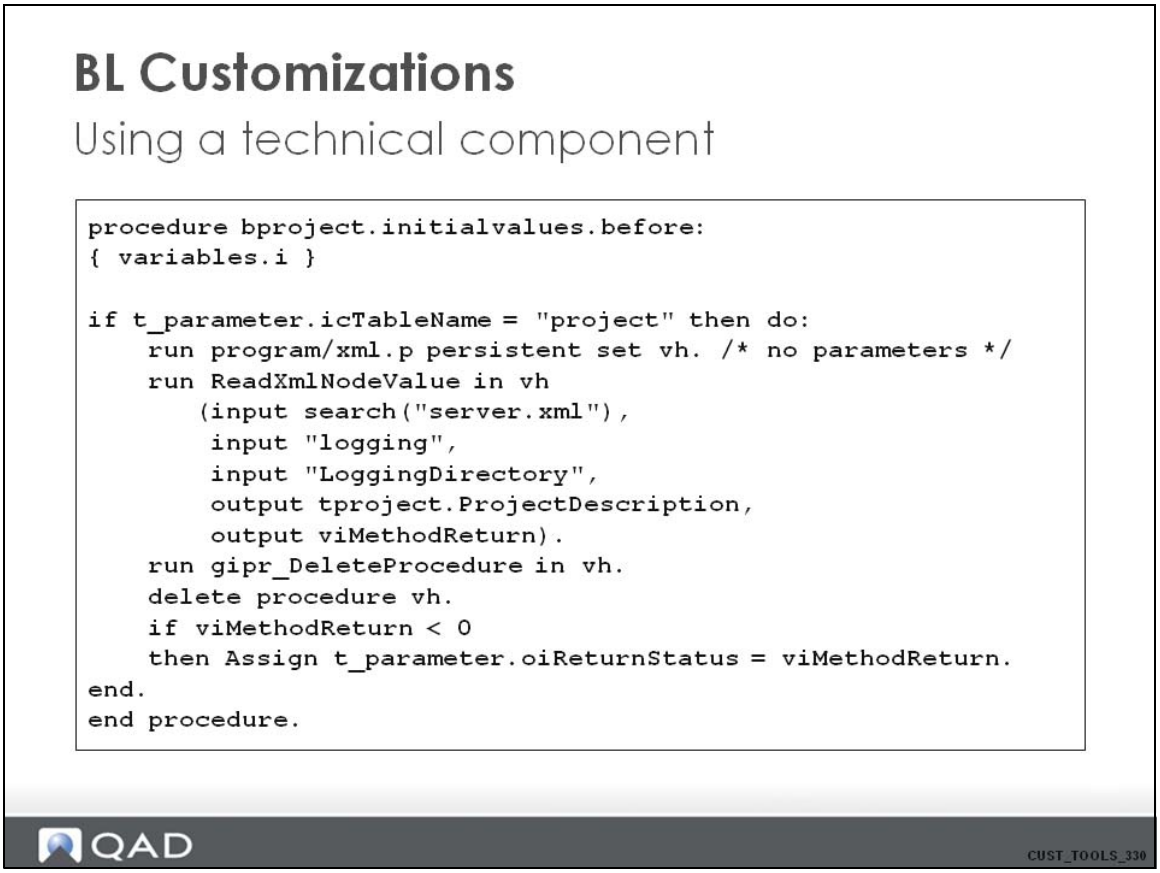
2. Run the method.

3. Stop the technical component, but first run a (generated) internal procedure to clean up any resources used by the component.

BL Customizations

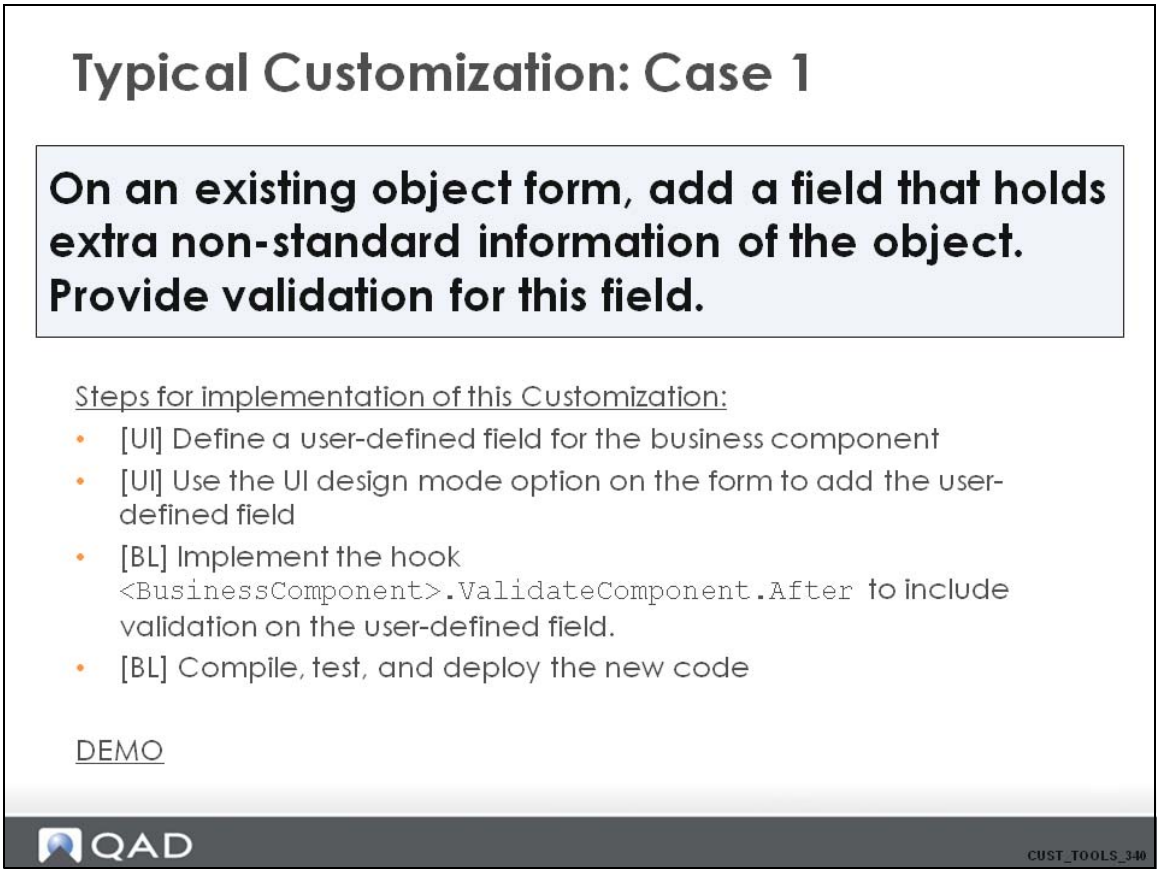


BL Customizations



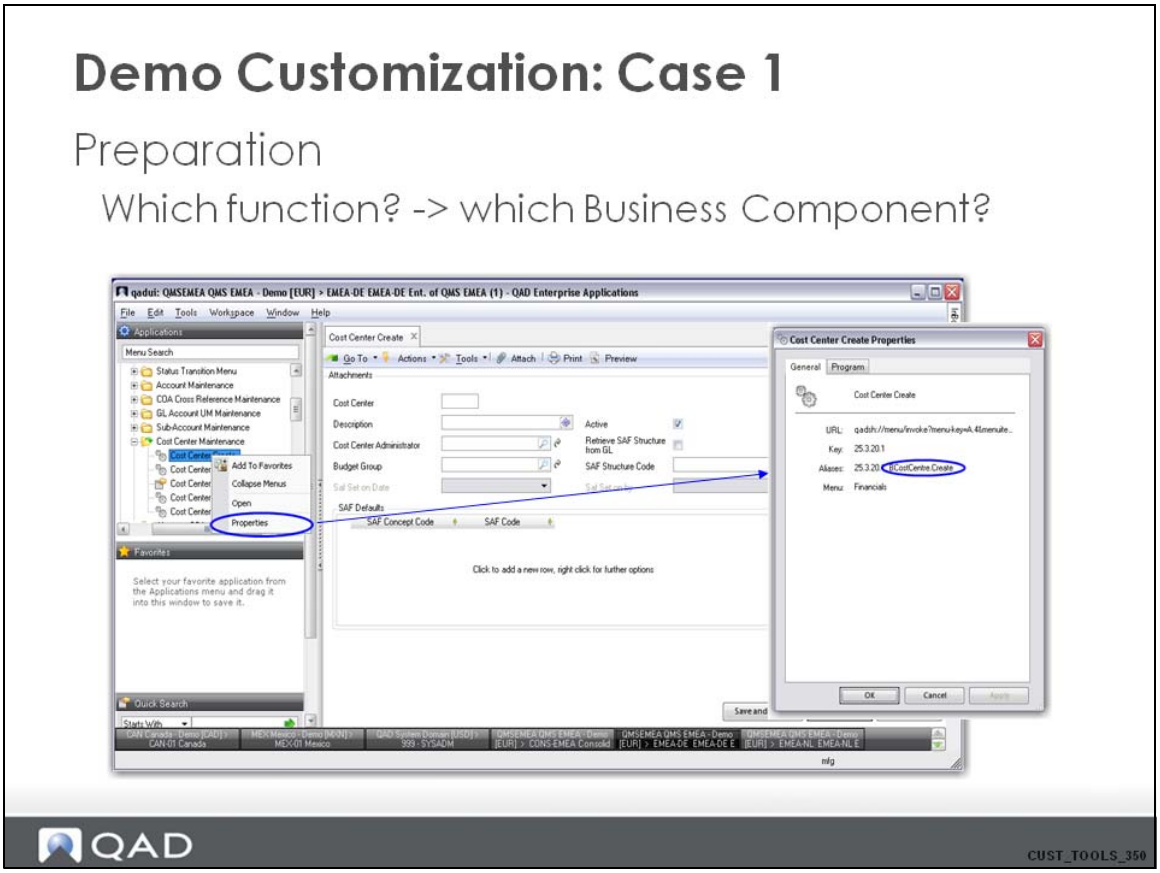
`server.xml` is a random XML file that is available. LoggingDirectory is a random node in this file. What would happen if you forget “gipr_DeleteProcedure”? Find out using unit test logging.

Typical Customization: Case 1

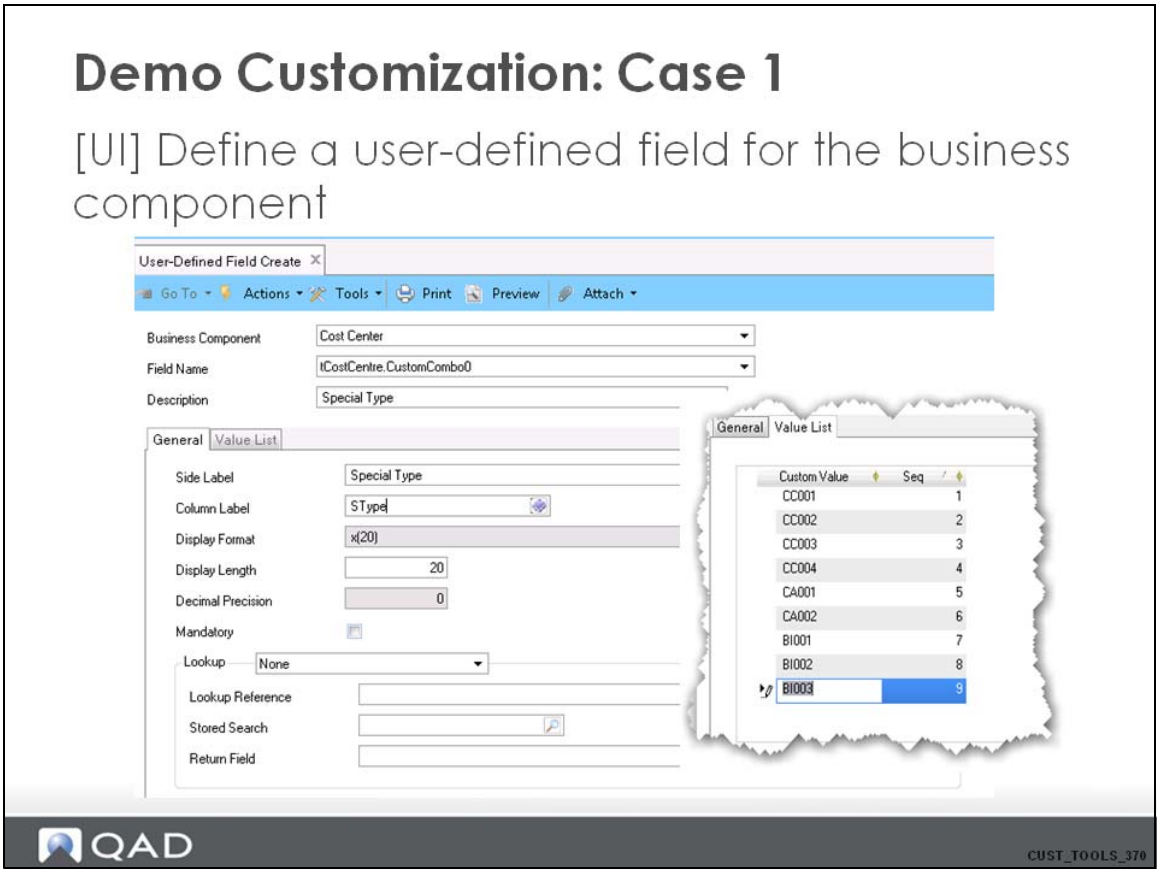


- Most of the forms in the client UI represent application objects. These forms are called “object forms”.
- Object forms are typically built up of business fields representing data for the object.
- Every object in the Financials has a set of predefined fields (called “Custom*”) that can be used as 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

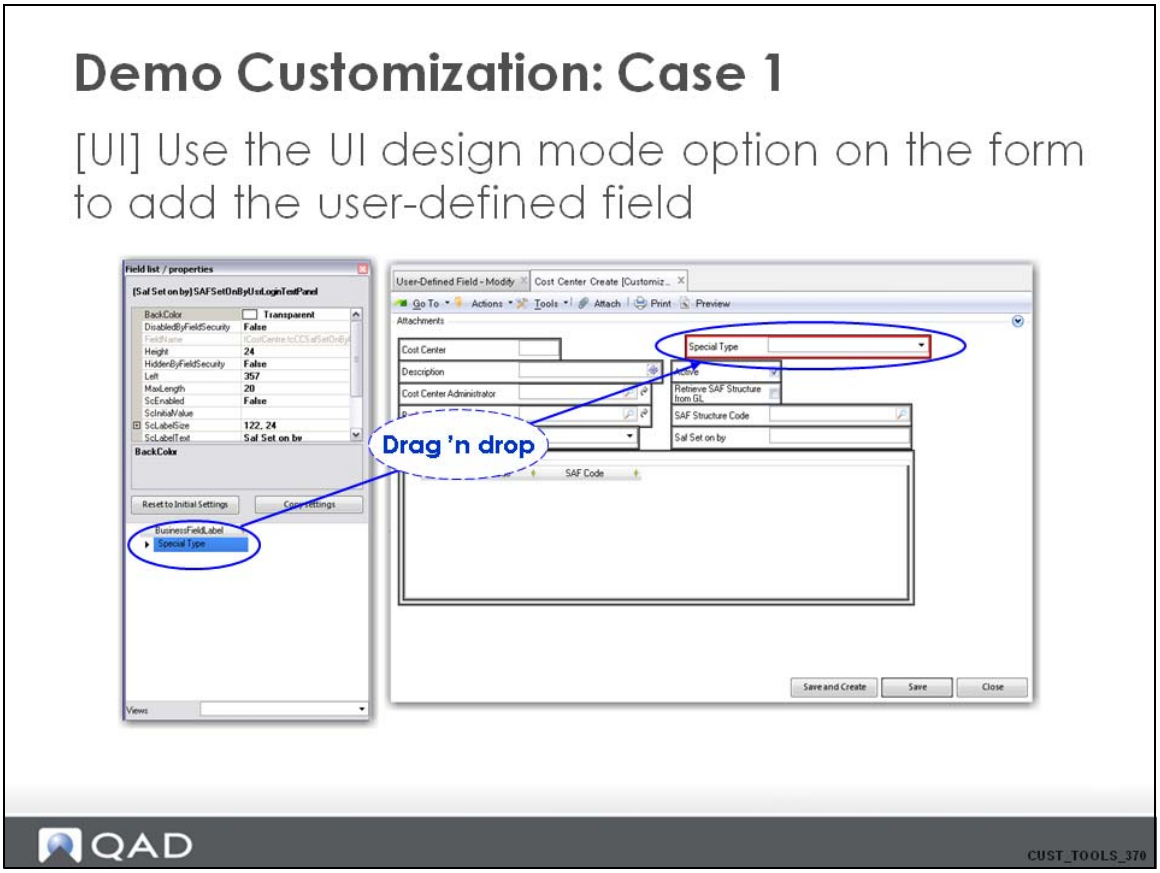


Demo Customization: Case 1



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

Demo Customization: Case 1

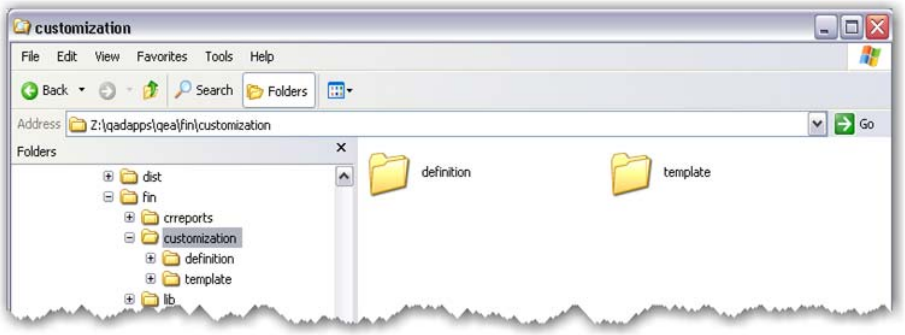


Demo Customization: Case 1

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



QAD

CUST_TOOLS_380

Demo Customization: Case 1

Demo Customization: Case 1

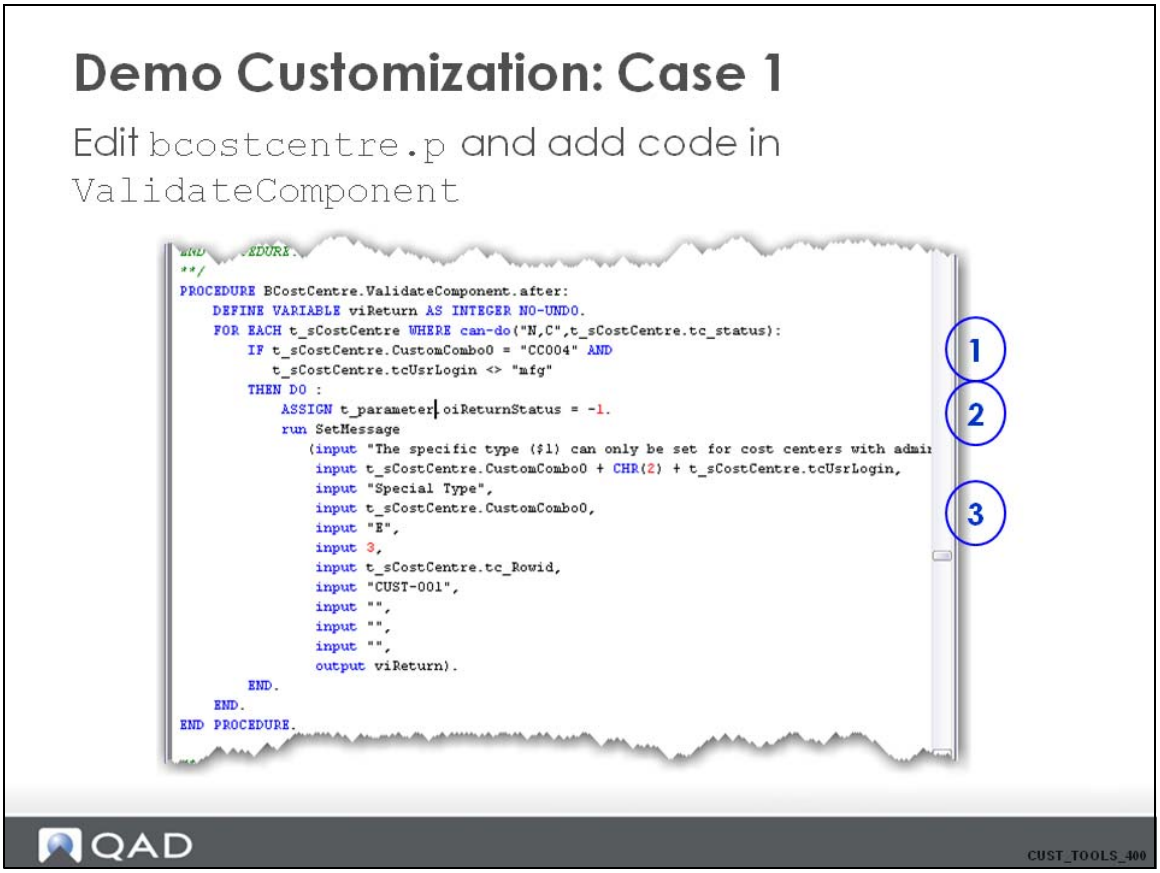
[BL] Implement the hook `<BusinessComponent>.ValidateComponent.After` to include validation on the user-defined field

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

QAD

CUST_TOOLS_390

Demo Customization: Case 1



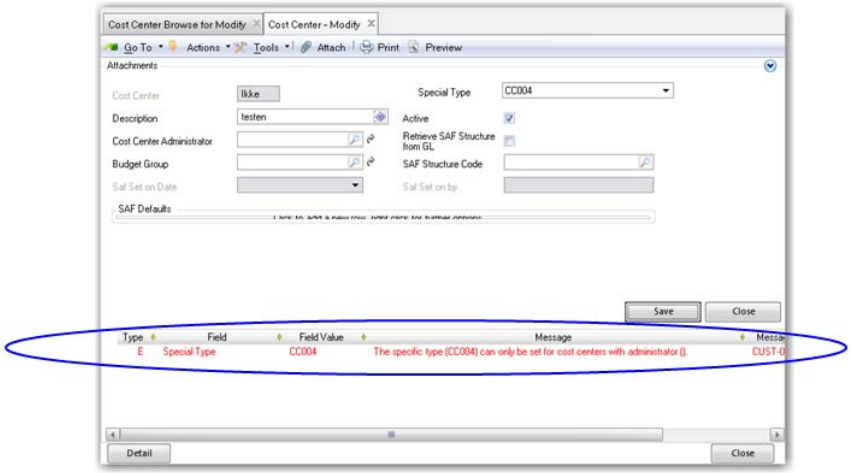
Uncomment the code for BCostCentre.ValidateComponent.After.

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

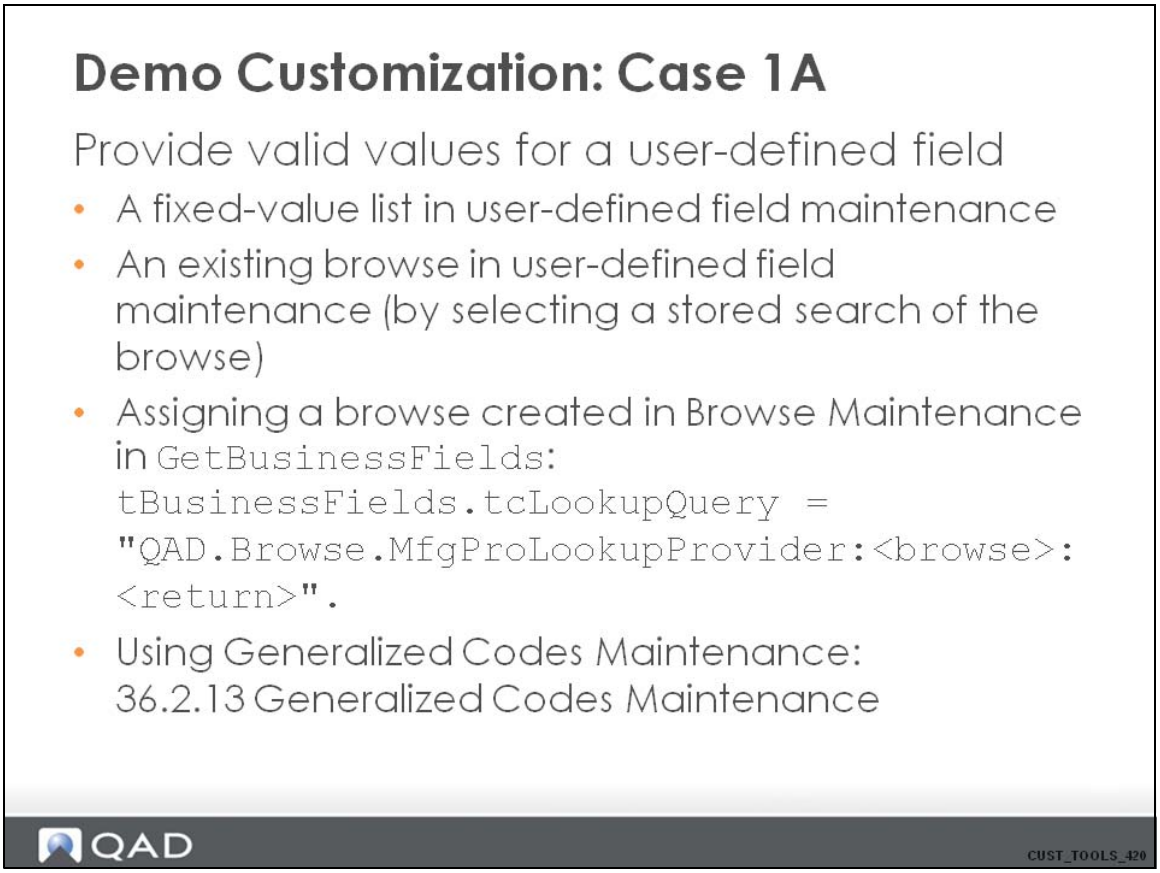
Compile, deploy customization, trim the AppServer, and run the Cost Center Create or Modify again



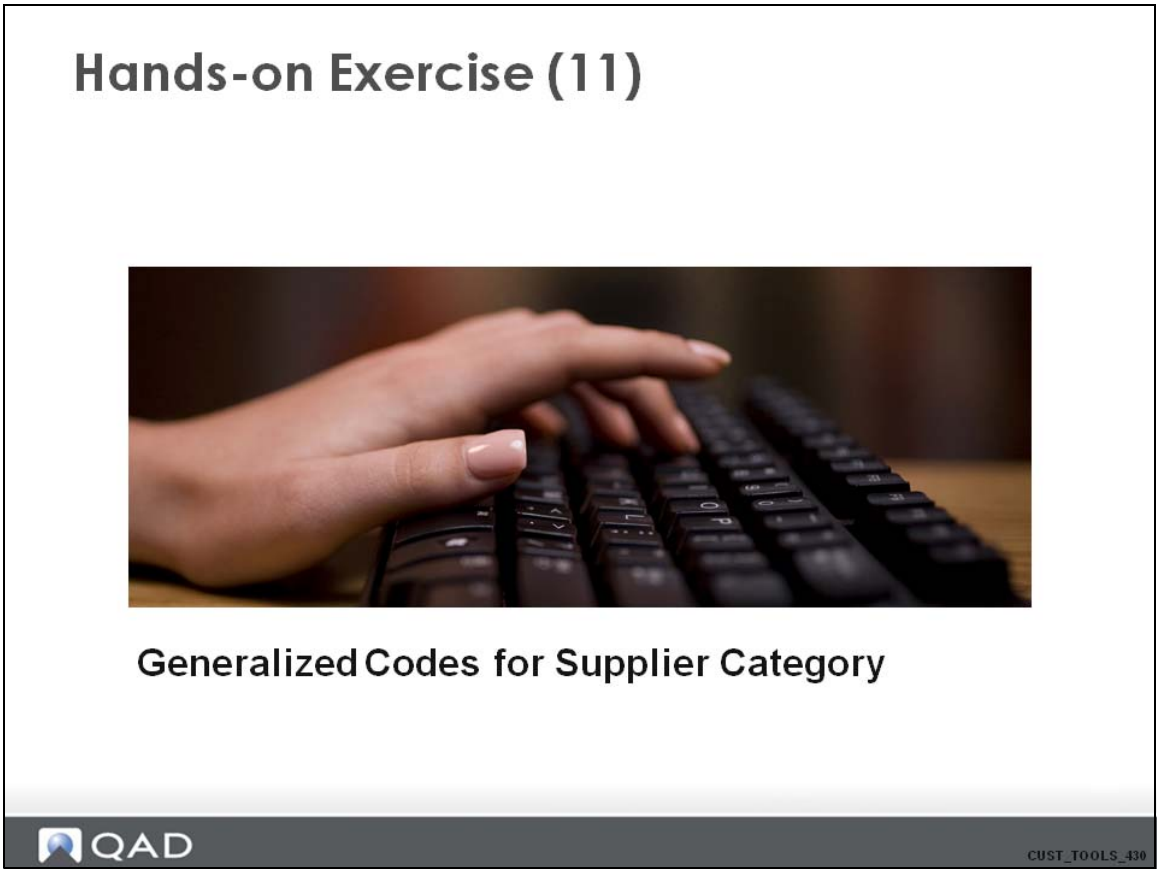
QAD

CUST_TOOLS_410

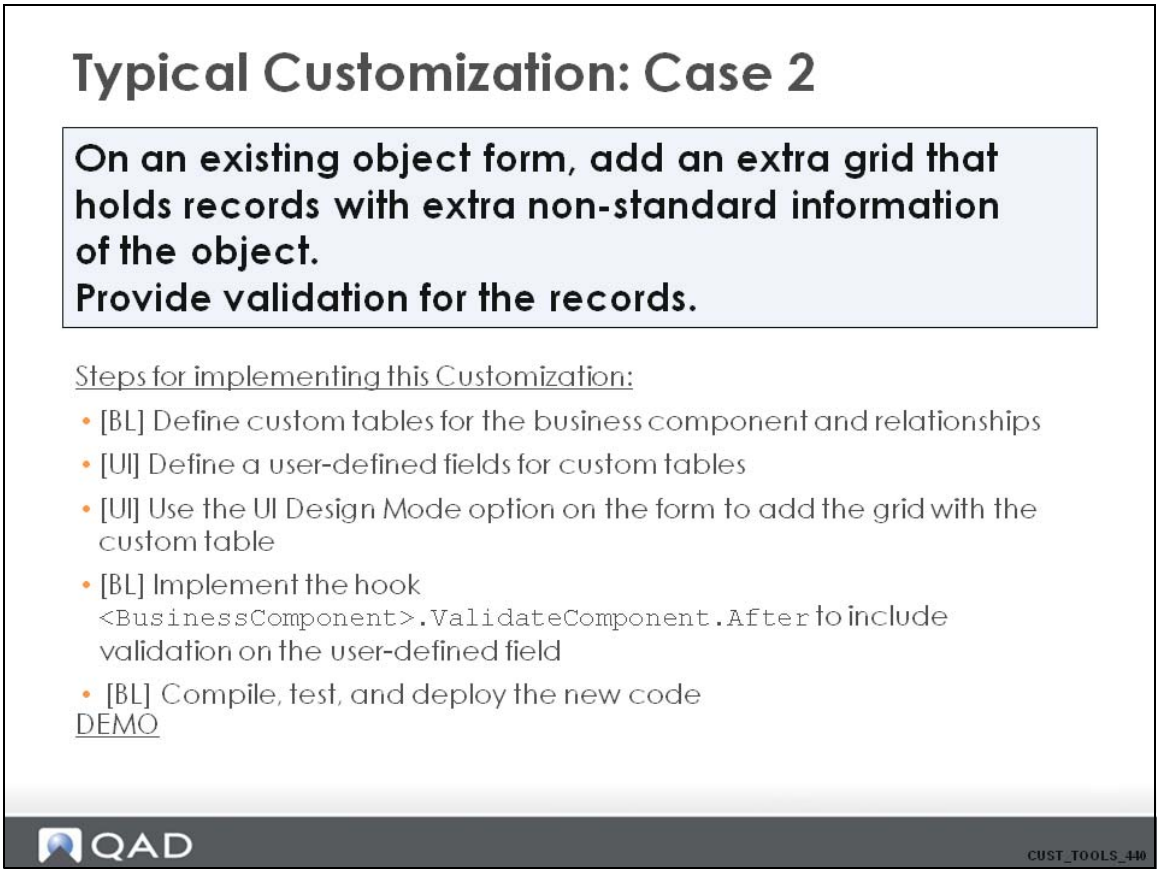
Demo Customization: Case 1A



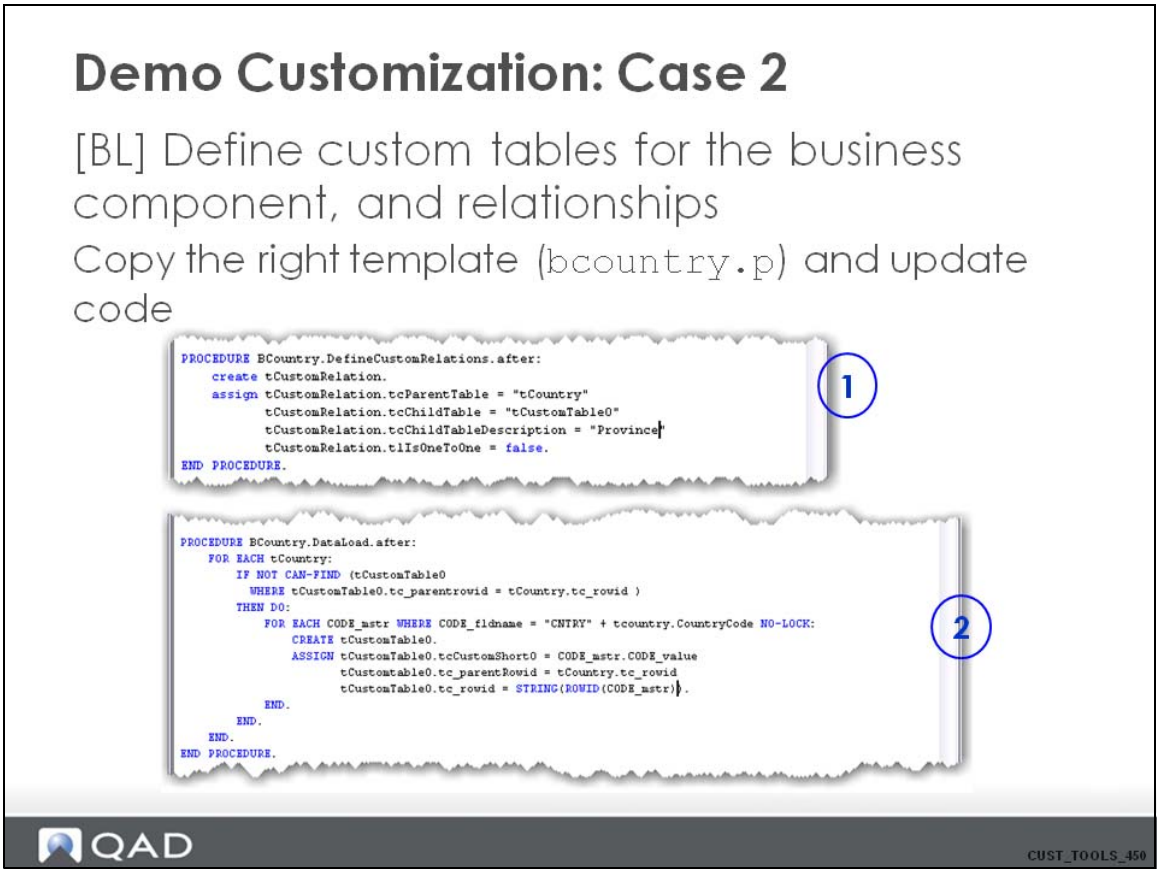
Hands-on Exercise (11)



Typical Customization: Case 2



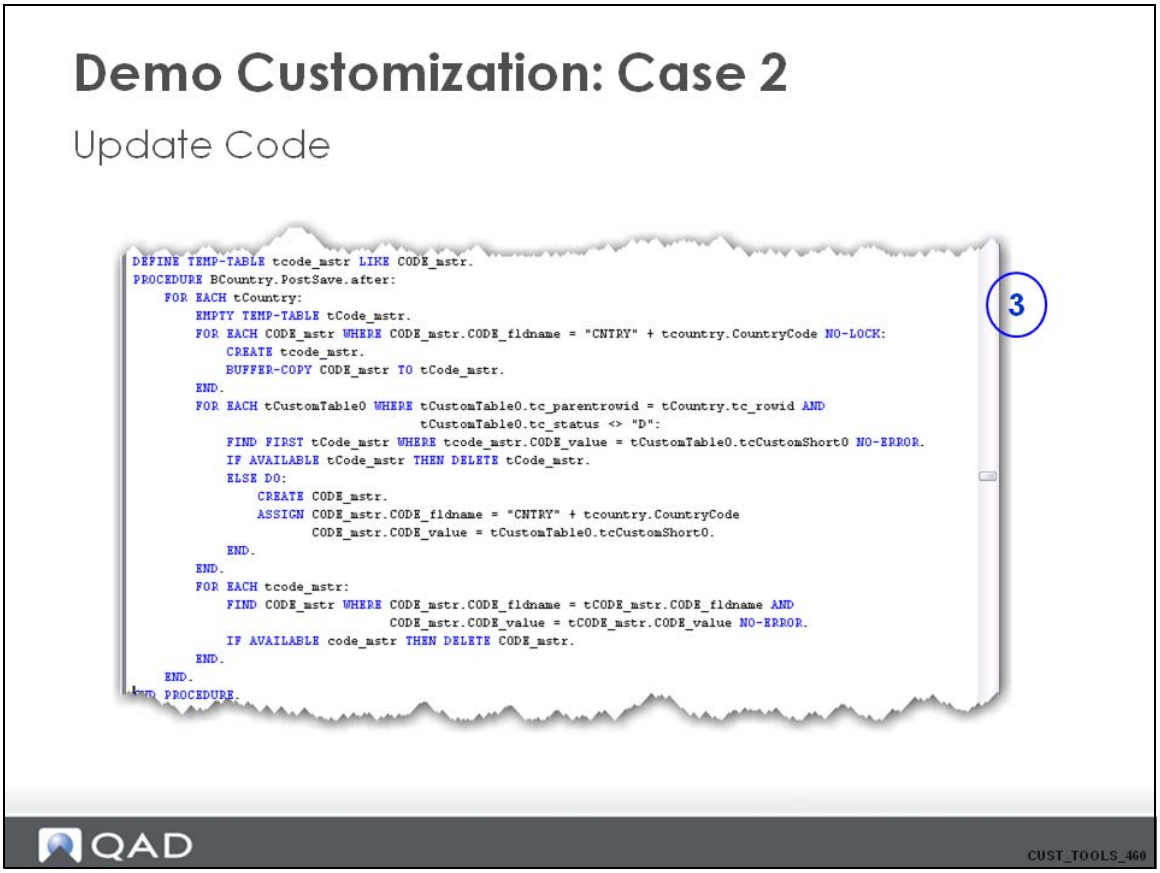
Demo Customization: Case 2



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



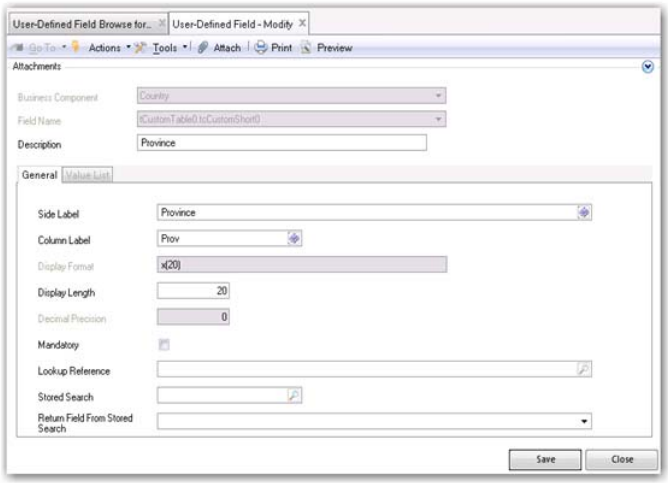

```
DEFINE TEMP-TABLE tcode_mstr LIKE CODE_mstr.
PROCEDURE BCountry.PostSave.after:
    FOR EACH tCountry:
        EMPTY TEMP-TABLE tCode_mstr.
        FOR EACH CODE_mstr WHERE CODE_mstr.CODE_fldname = "CNTRY" + tcountry.CountryCode NO-LOCK:
            CREATE tcode_mstr.
            BUFFER-COPY CODE_mstr TO tCode_mstr.
        END.
        FOR EACH tCustomTable0 WHERE tCustomTable0.tc_parentrowid = tCountry.tc_rowid 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.
```

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

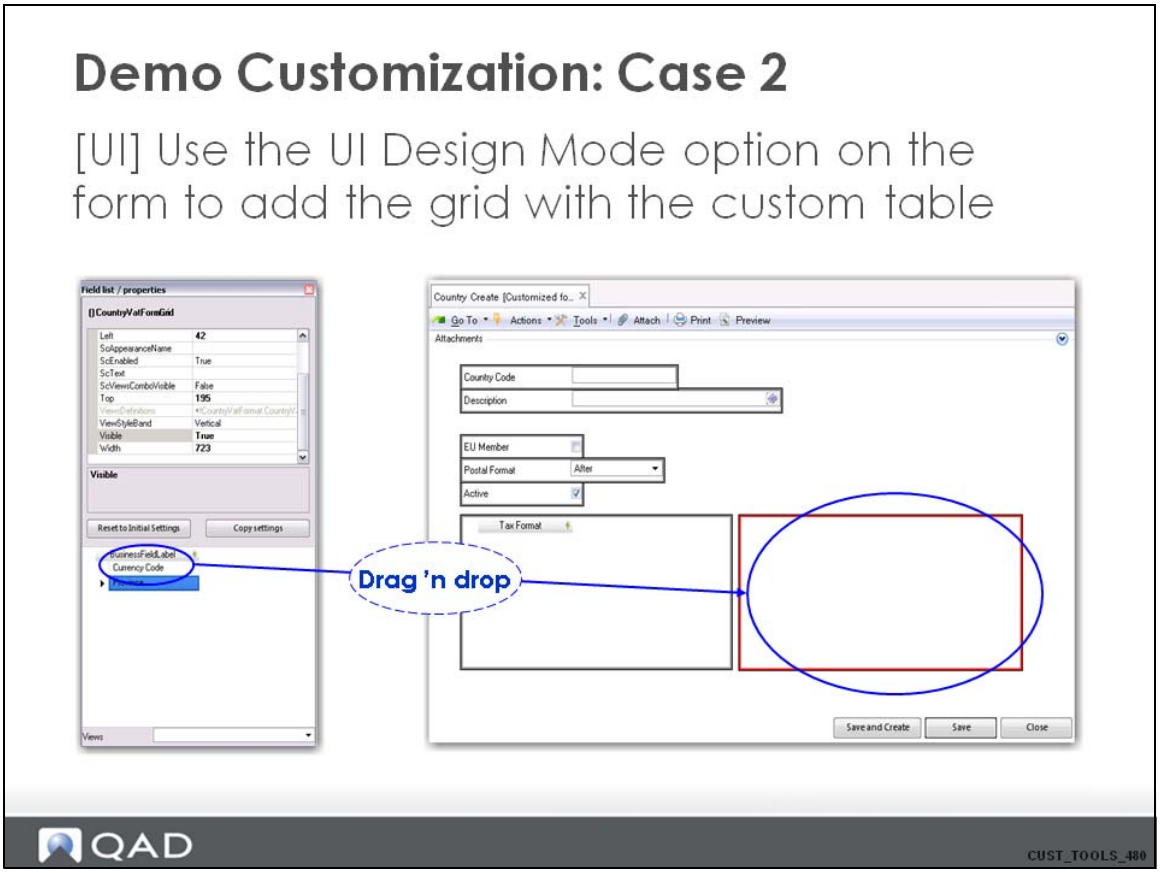
- Compile, deploy Customization, and trim the AppServer
- [UI] Define a user-defined fields for custom tables



QAD

CUST_TOOLS_470

Demo Customization: Case 2

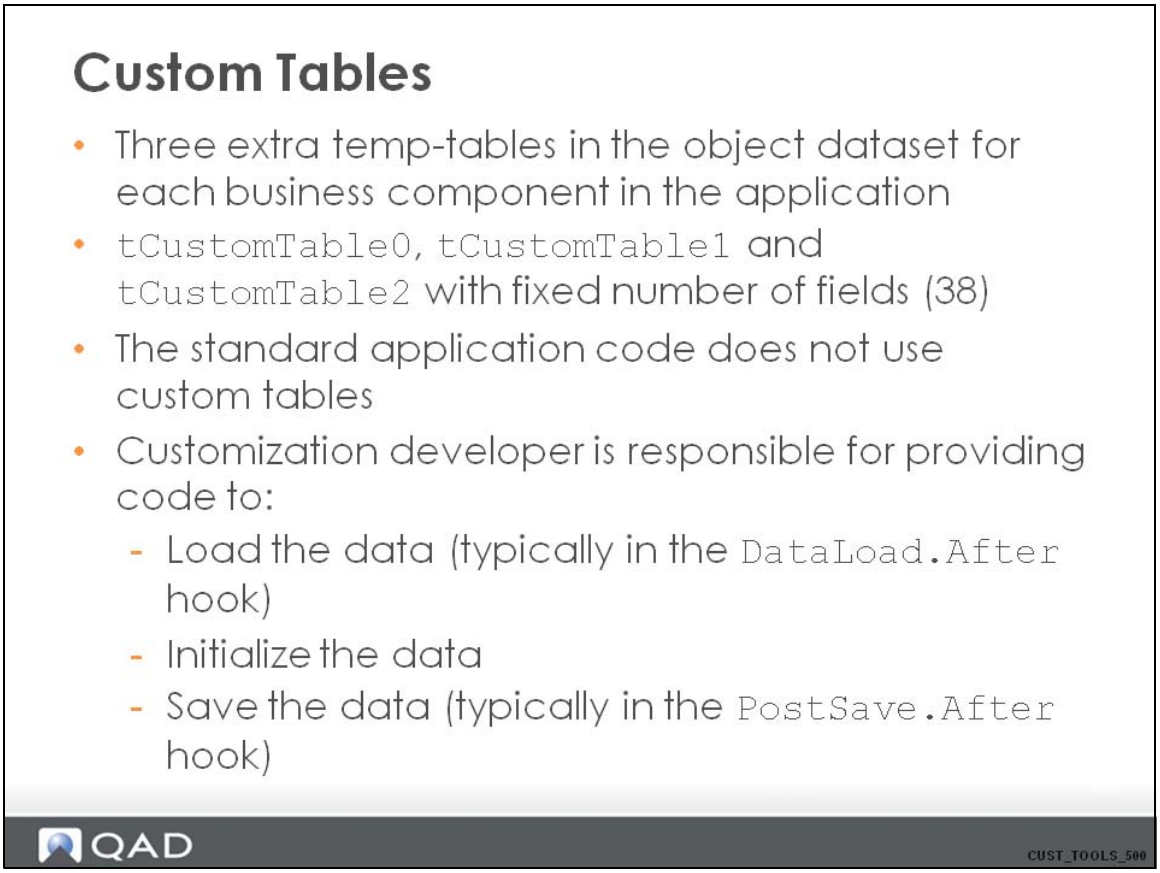


Set the property “AllowAddNew” to true in the properties panel.

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

Demo Customization: Case 2

Custom Tables



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

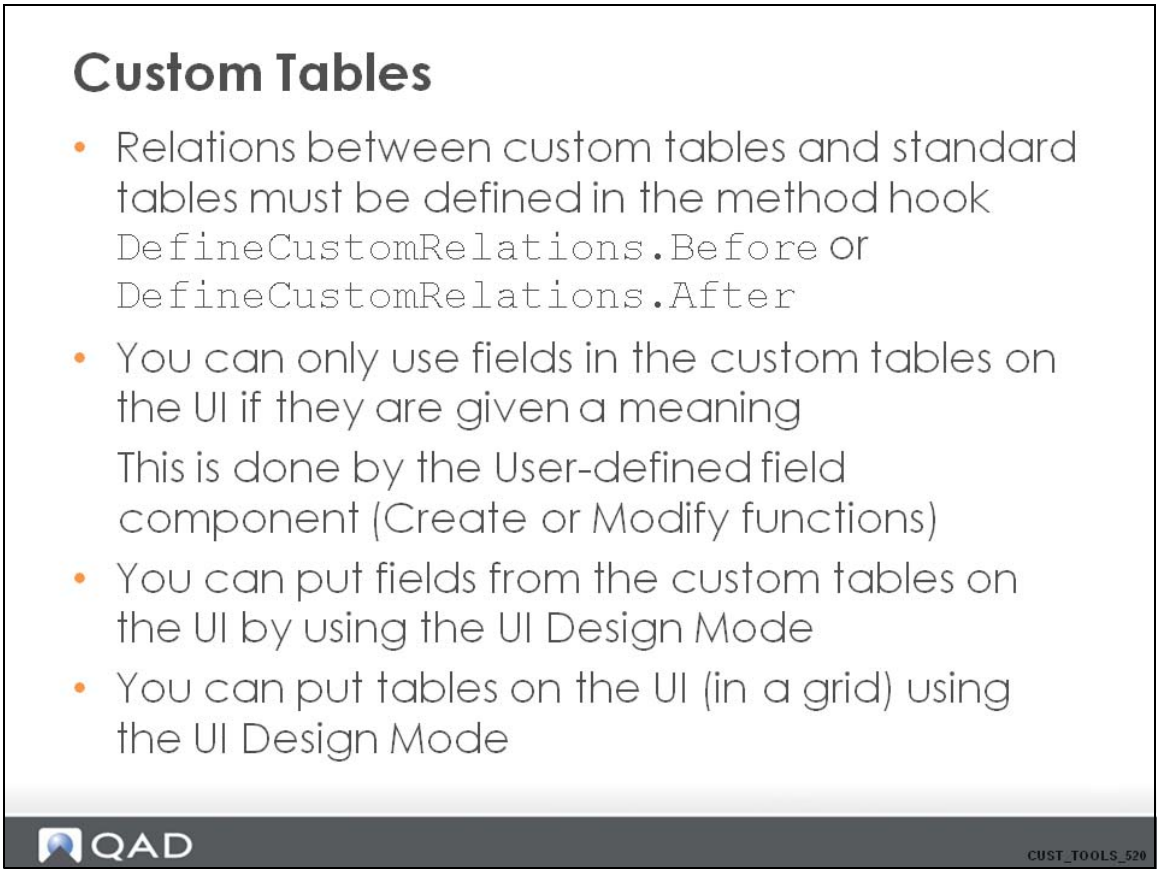
- Three extra temp-tables:

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

QAD

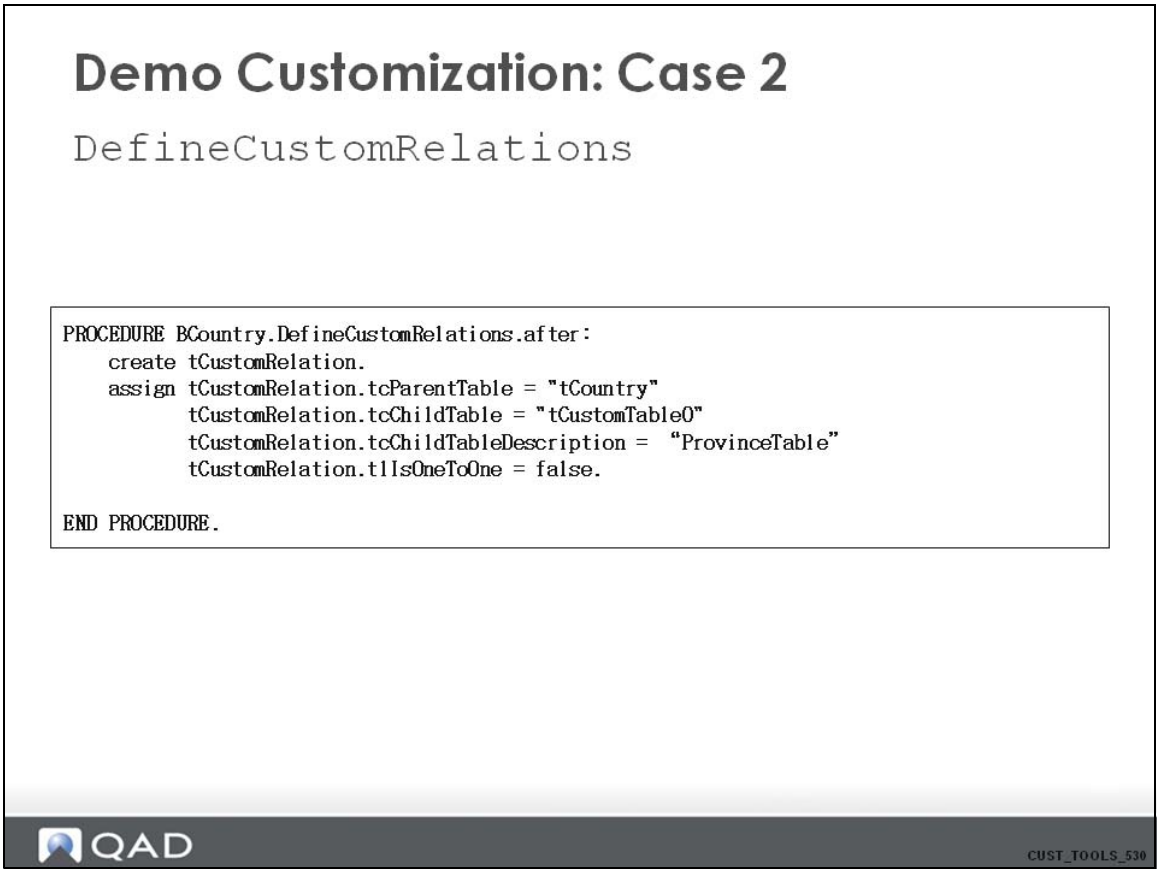
CUST_TOOLS_510

Custom Tables 3



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

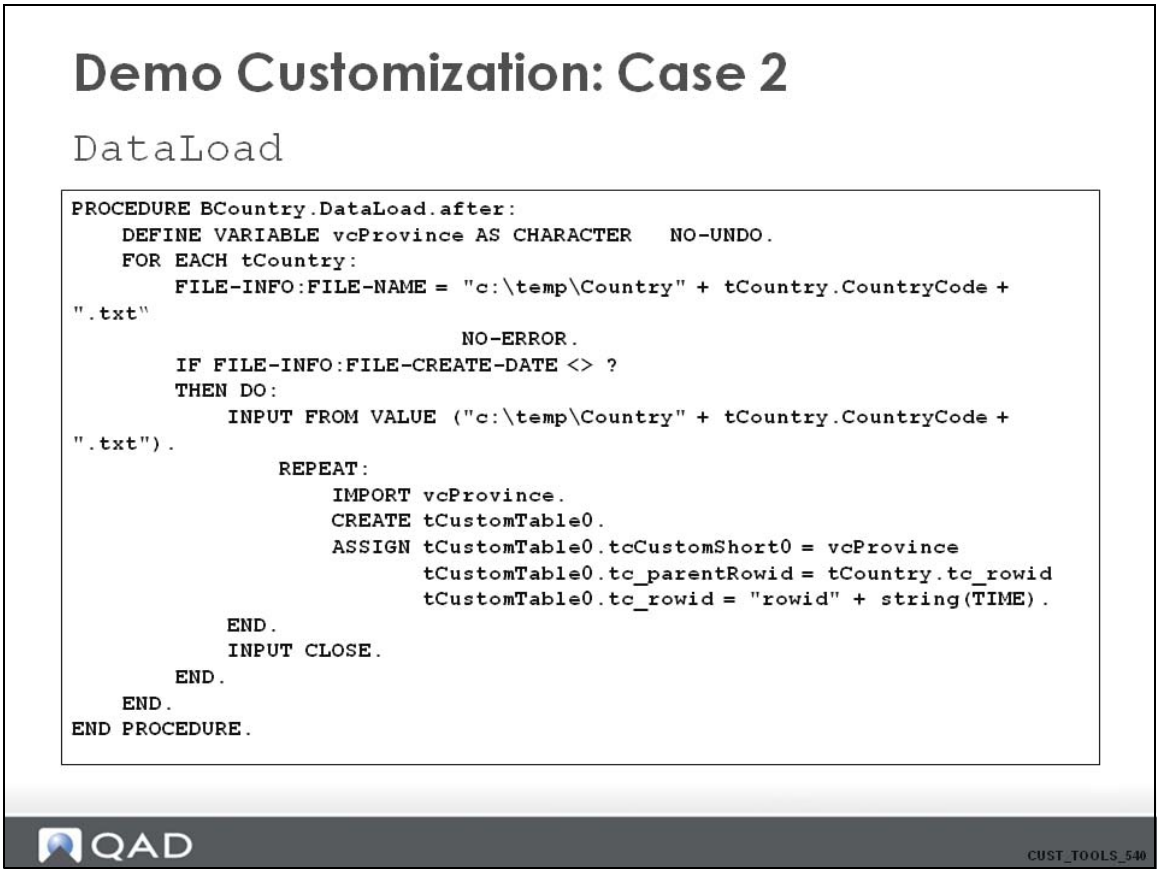


The only way to make sure that the system recognizes the custom table is to add at least one record in tCustomRelation temp-table.

The record fields are:

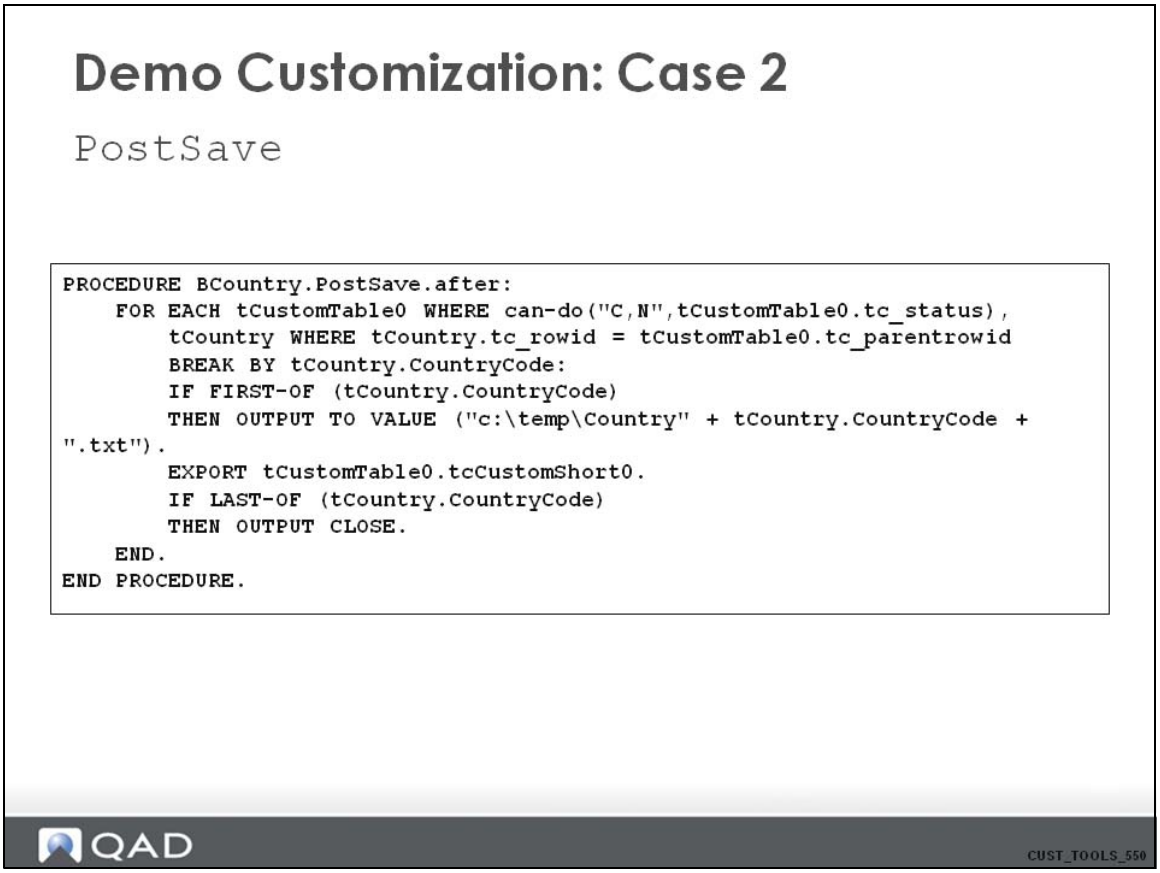
- tcParentTable: Name of the temp-table from the object dataset. Typically “t<table>”.
- tcChildTable: Name of the custom table from the object dataset that you want to give a meaning. This can be “tCustomTable0”, “tCustomTable1” or “tCustomTable2”.
- tcChildTableDescription: The logical name to use 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



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

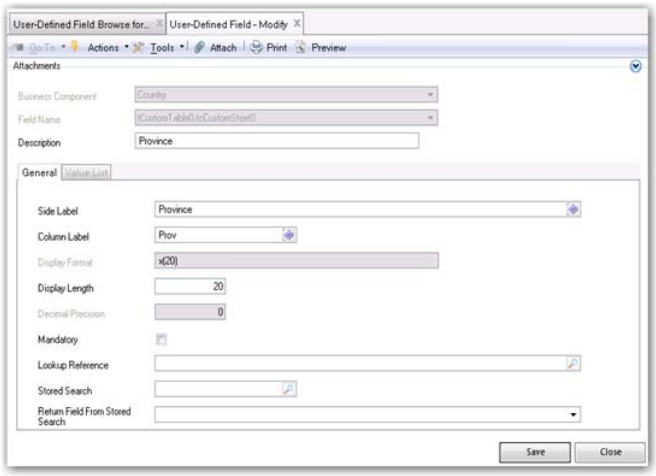
Demo Customization: Case 2 - PostSave



Demo Customization: Case 2 - Compile and Deploy

Demo Customization: Case 2

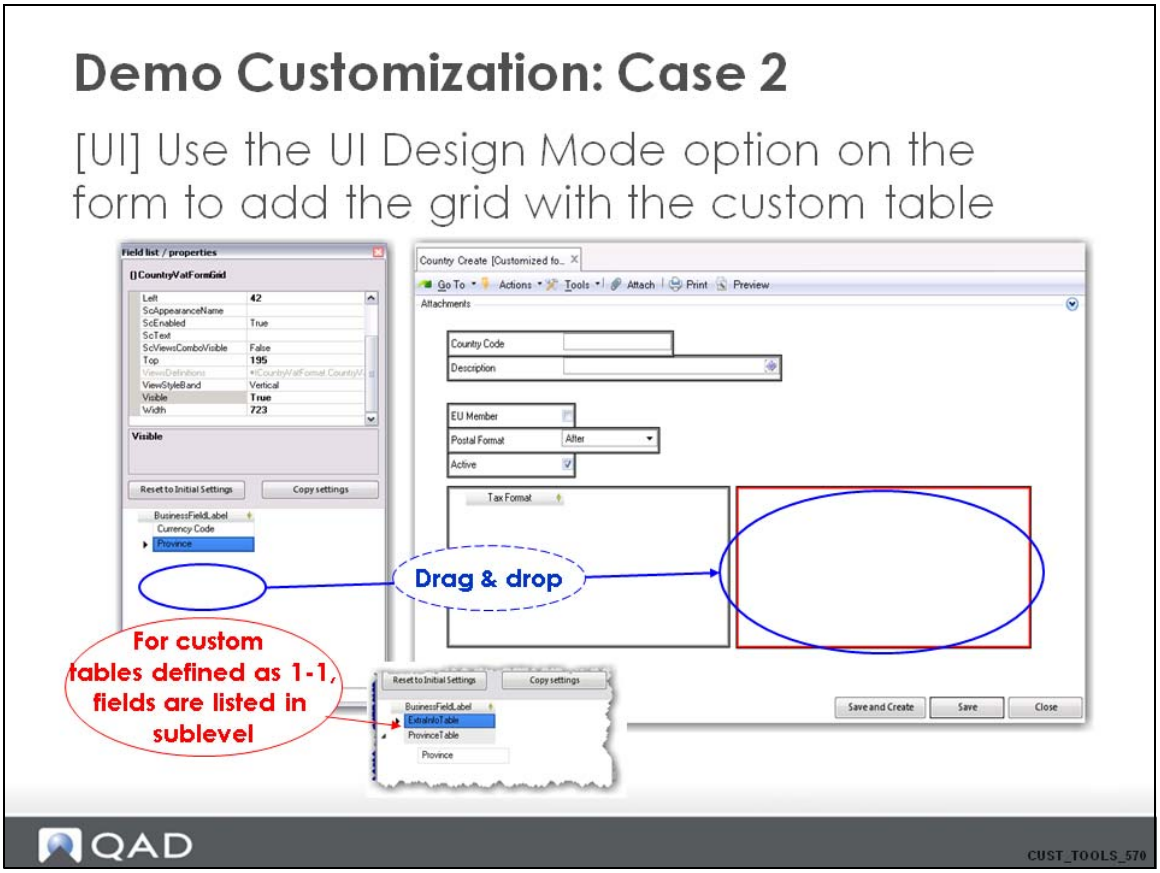
- Compile, deploy Customization, and trim the AppServer
- [UI] Define a user-defined field for custom tables



QAD

CUST_TOOLS_560

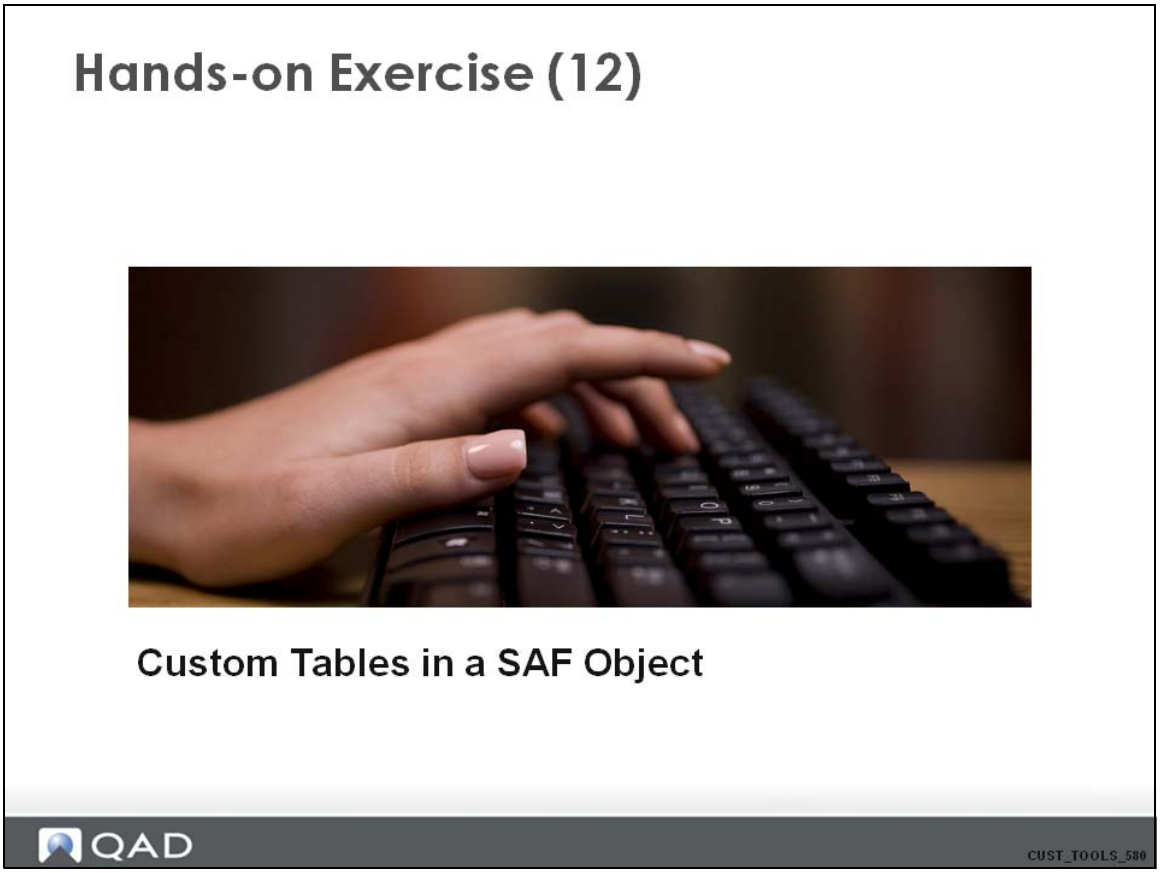
Demo Customization: Case 2 - Add Grid



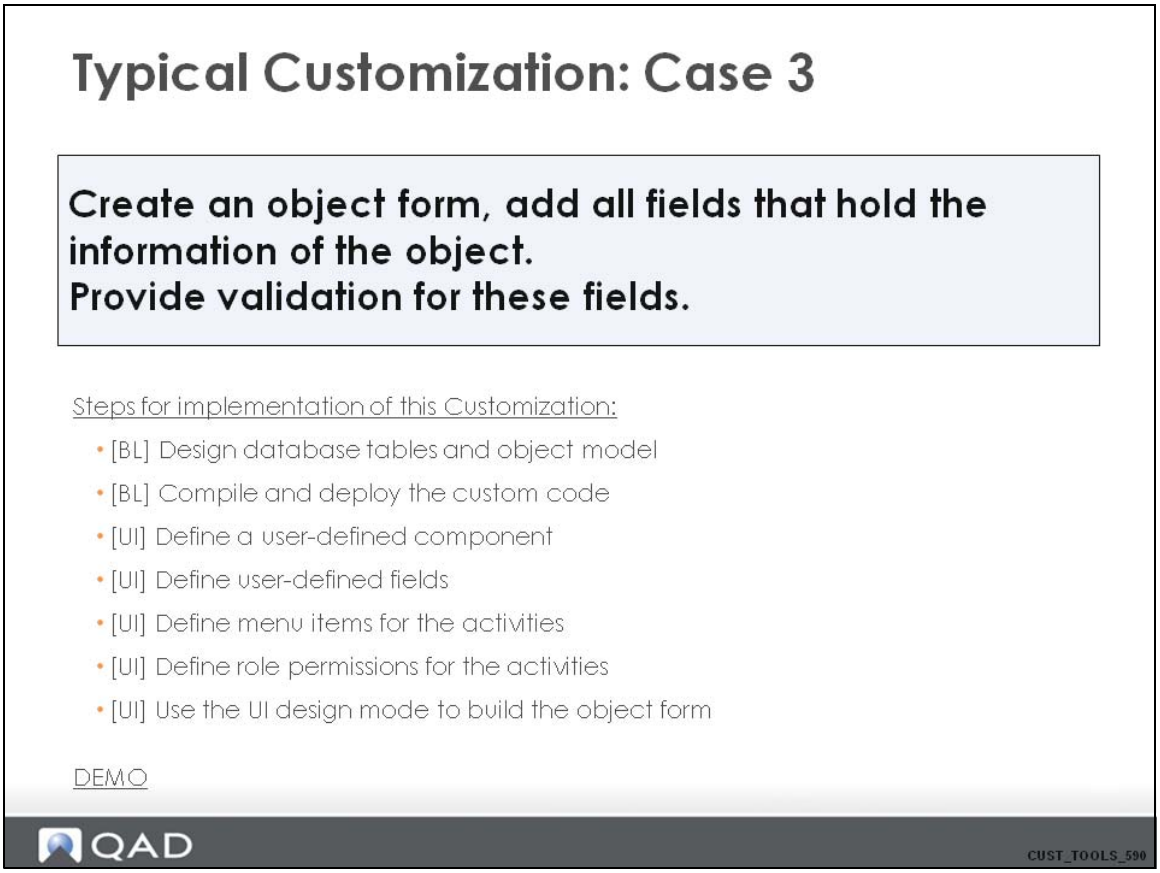
Set the property “AllowAddNew” to true in the properties panel.

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

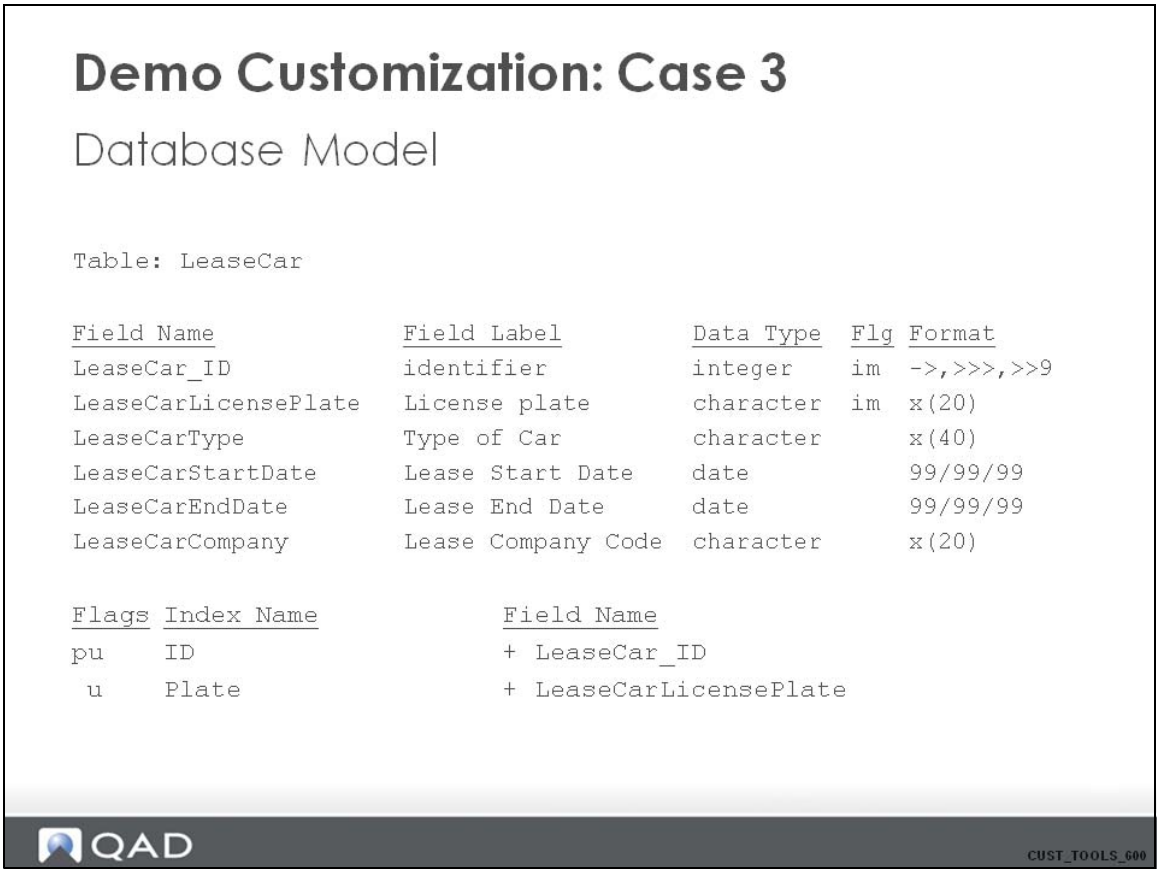
Hands-on Exercise (12)



Typical Customization: Case 3

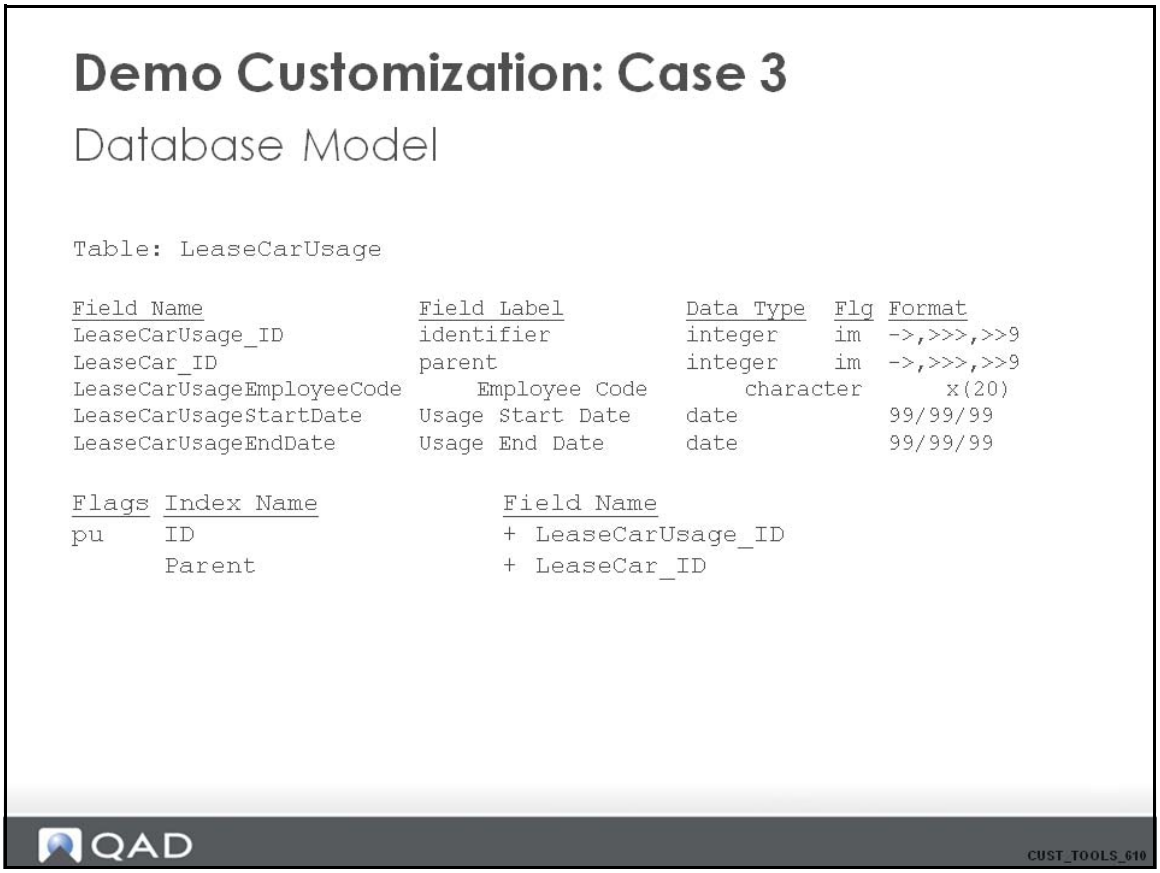


Demo Customization: Case 3 - Database Model

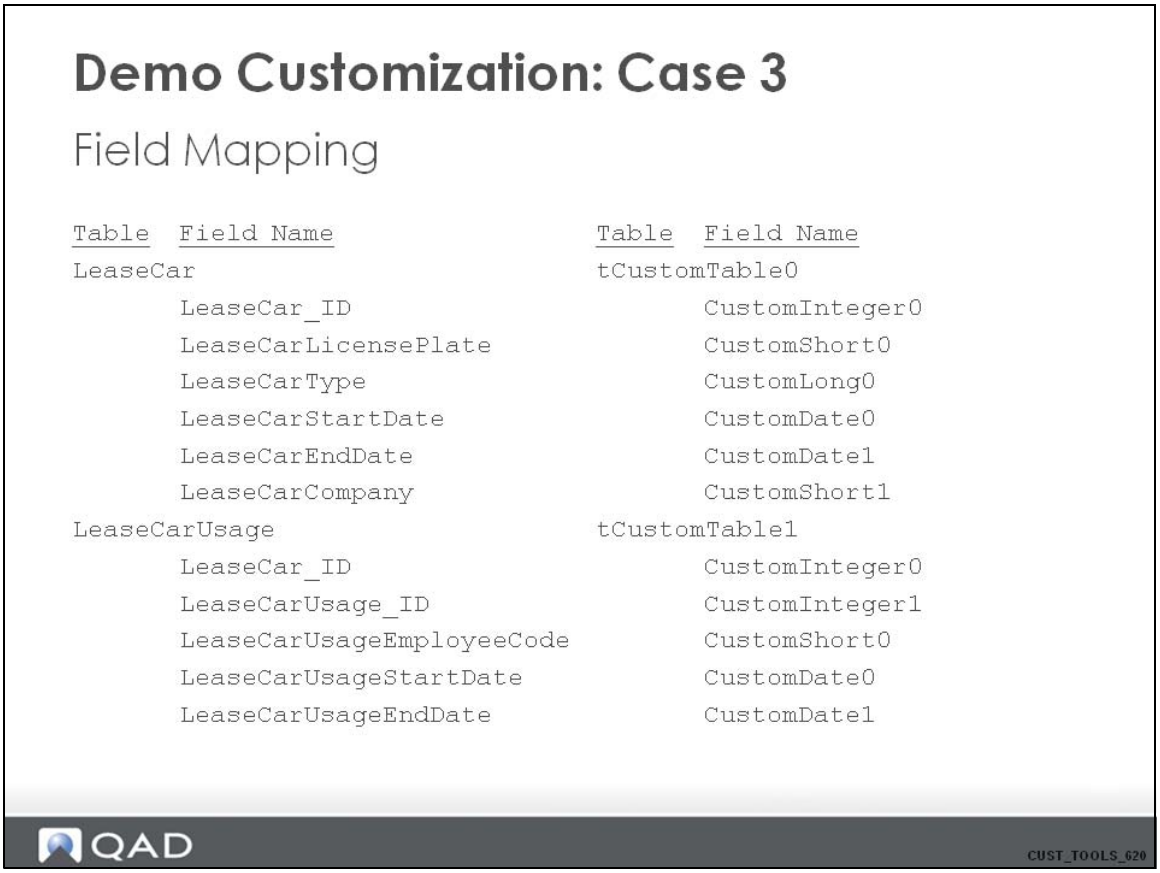


This customization case demonstrates the maintenance of a fleet of leased cars and vans and to which employee a vehicle is assigned.

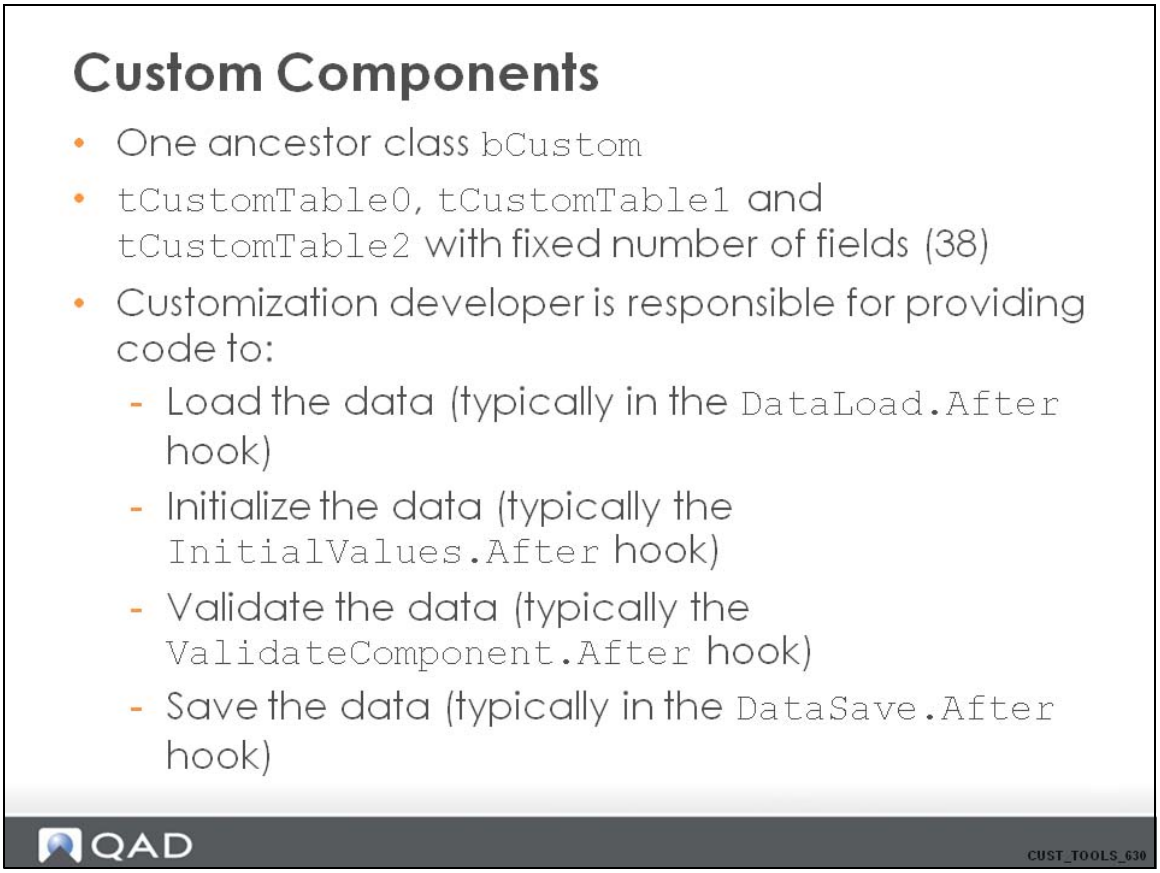
Demo Customization: Case 3 - Database Model



Demo Customization: Case 3 - Field Mapping

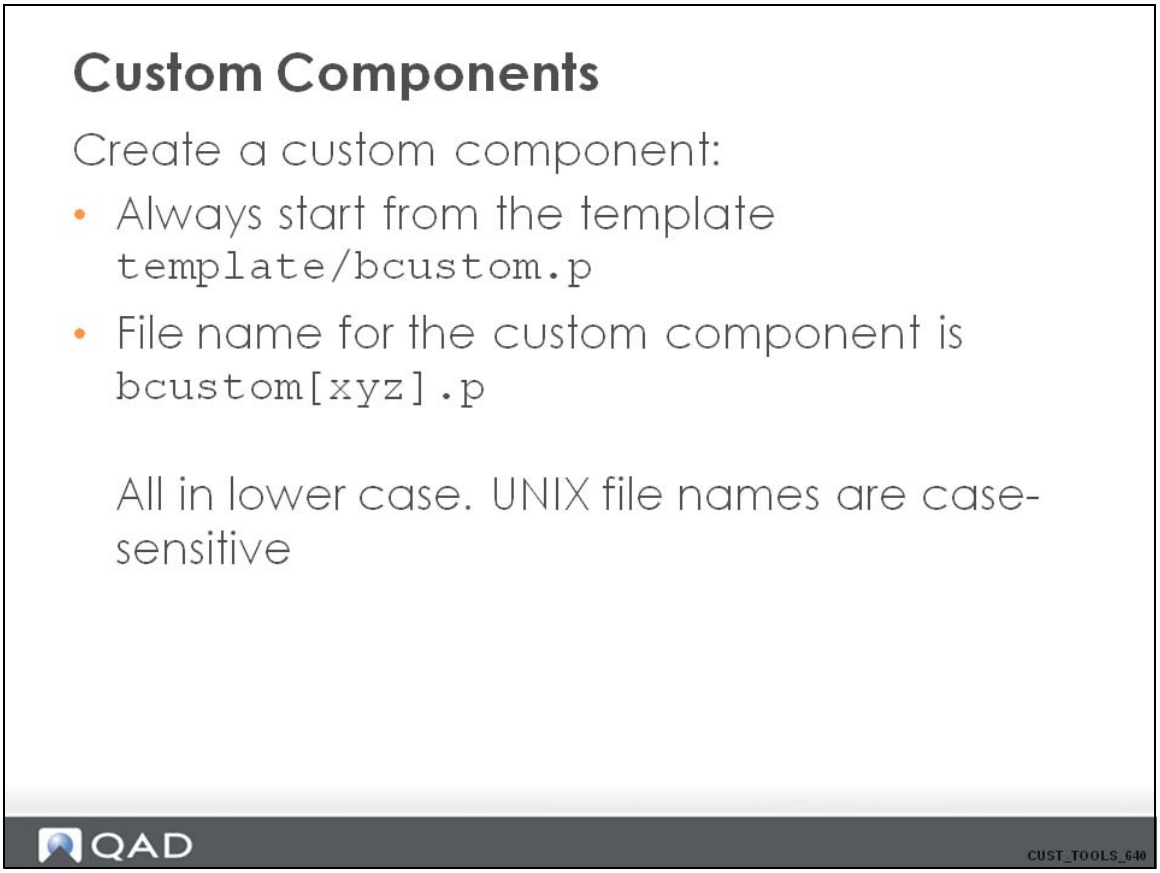


Custom Components



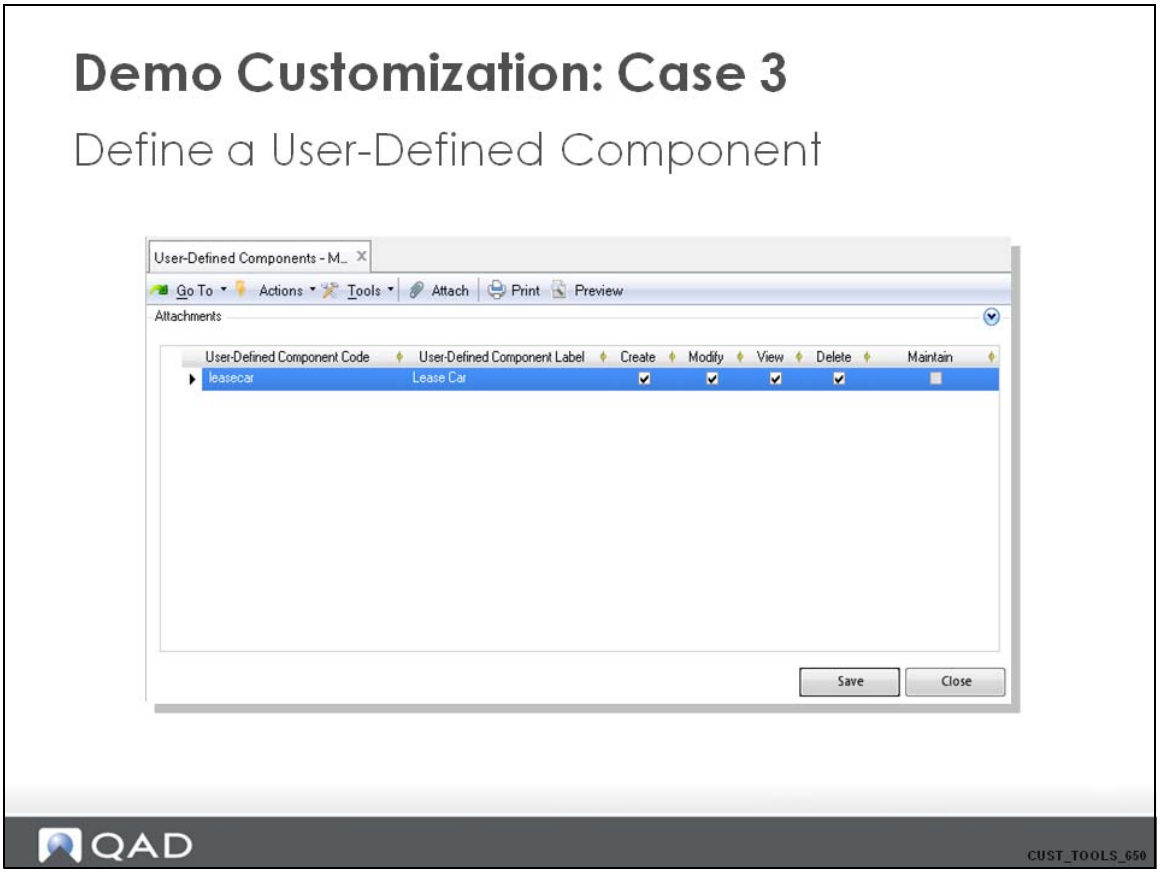
Every custom business component is inherited from component “bcustom”. Every custom component has access to three custom tables only.

Custom Components



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

Demo Customization: Case 3 - Define User-Defined Component



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)

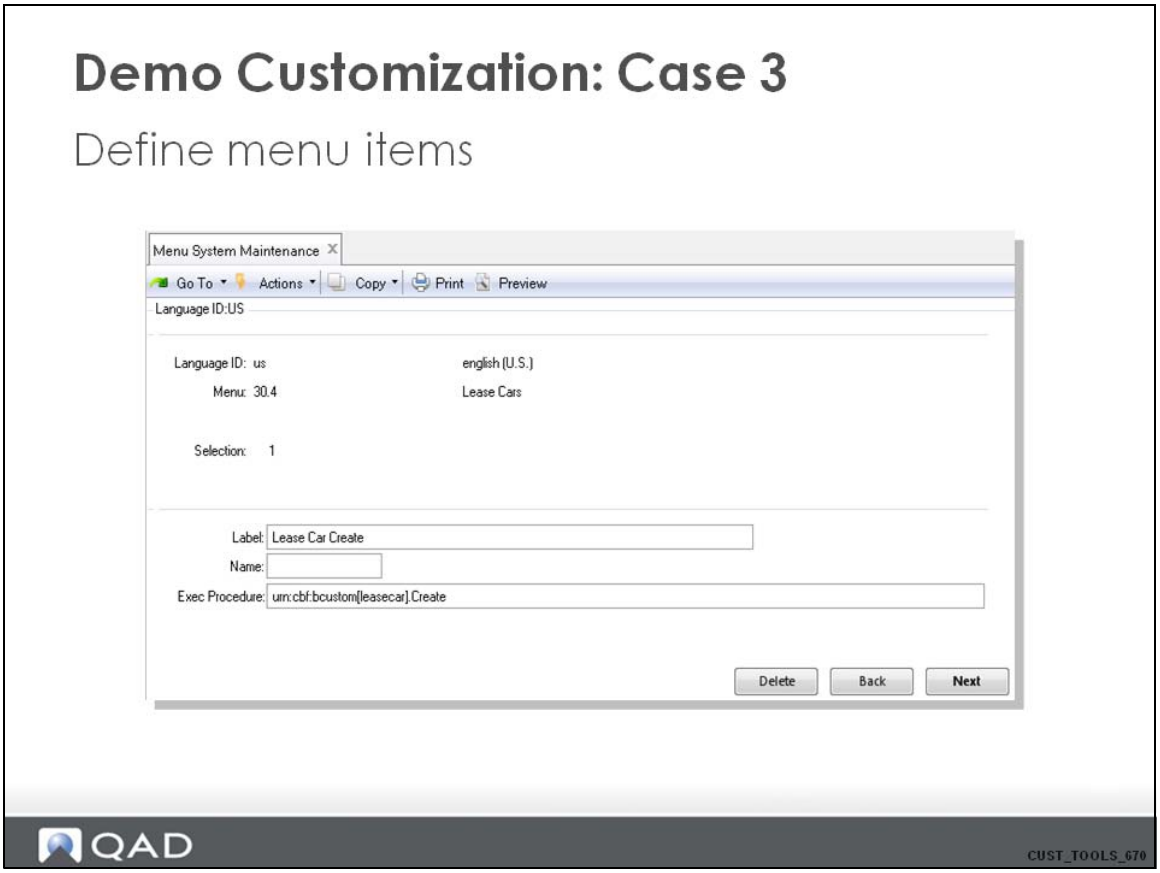
tCustomTable1.CustomShort0 (LeaseCarUsageEmployeeCode)

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”.

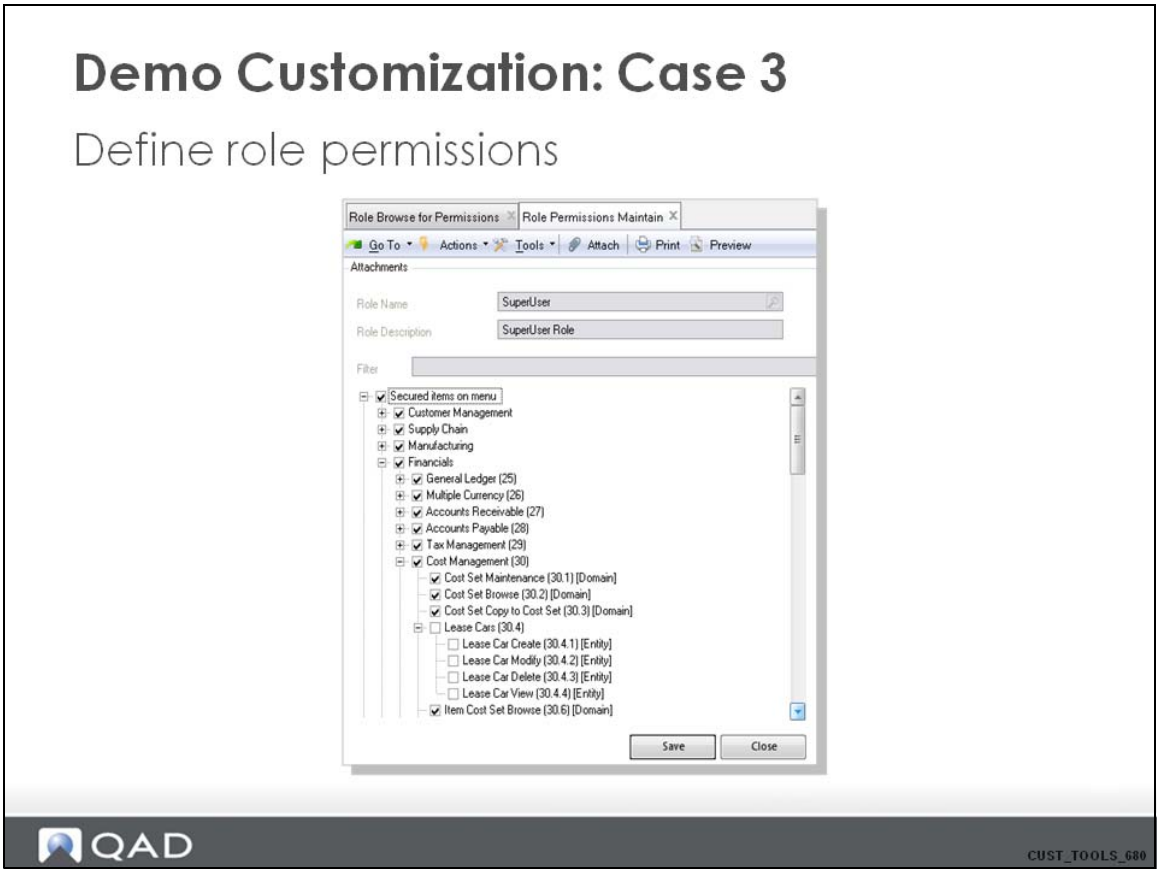
Demo Customization: Case 3 - Define Menu Items



Define these menu items:

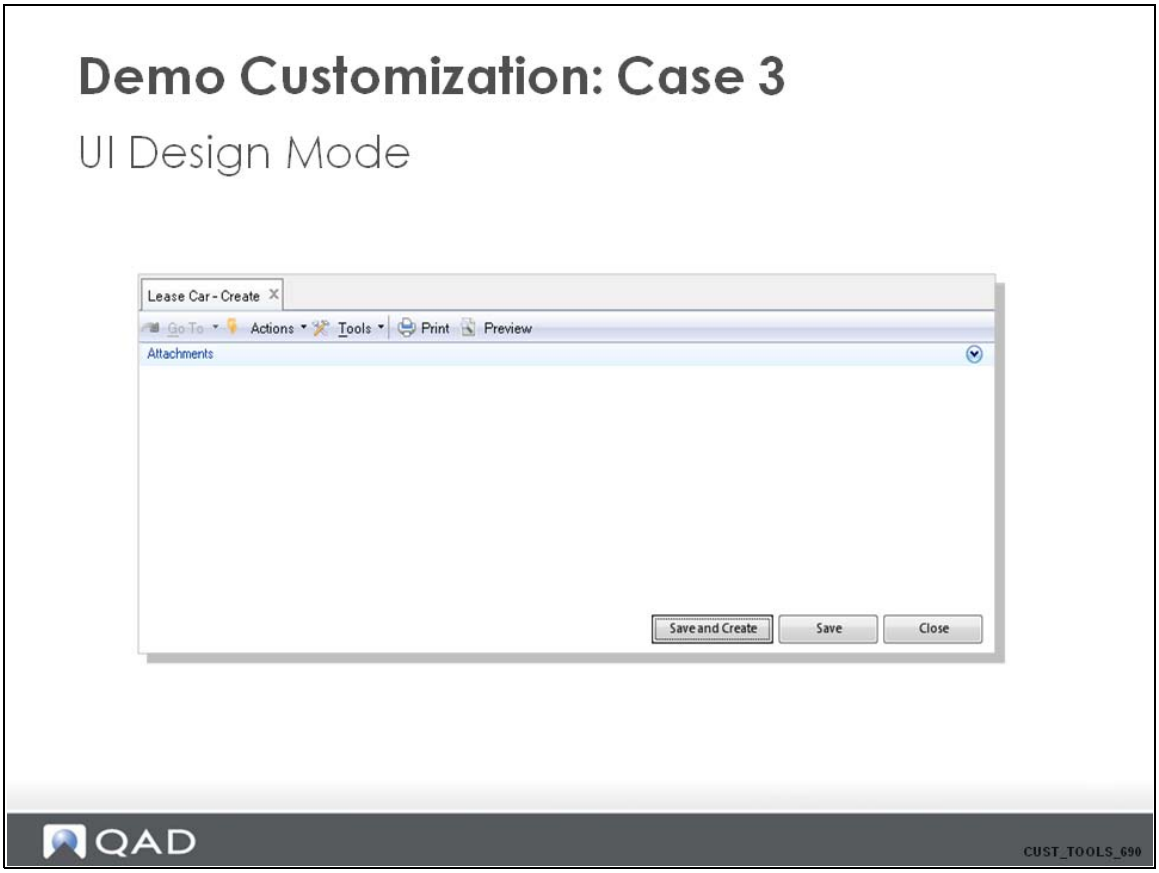
Nr	Label	Procedure
30.4	Lease Cars	30.4
30.4.1	Lease Car Create	urn:cbf:bcustom[leasecar].Create
30.4.2	Lease Car Modify	urn:cbf:bcustom[leasecar].Modify
30.4.3	Lease Car Delete	urn:cbf:bcustom[leasecar].Delete
30.4.4	Lease Car View	urn:cbf:bcustom[leasecar].View

Demo Customization: Case 3 - Define Role Permissions



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

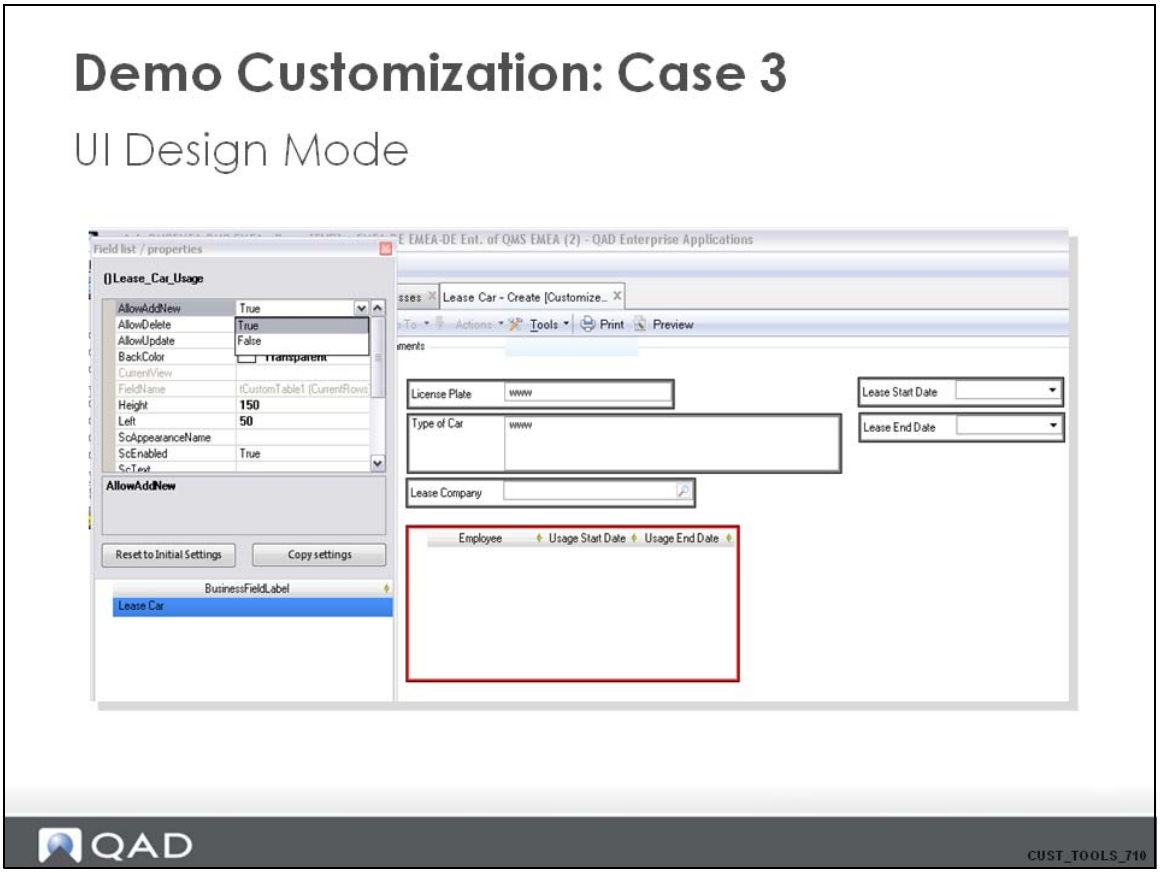
Expand the first tab (Lease Car), then drag and drop all fields under this tab to the maintenance form.

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

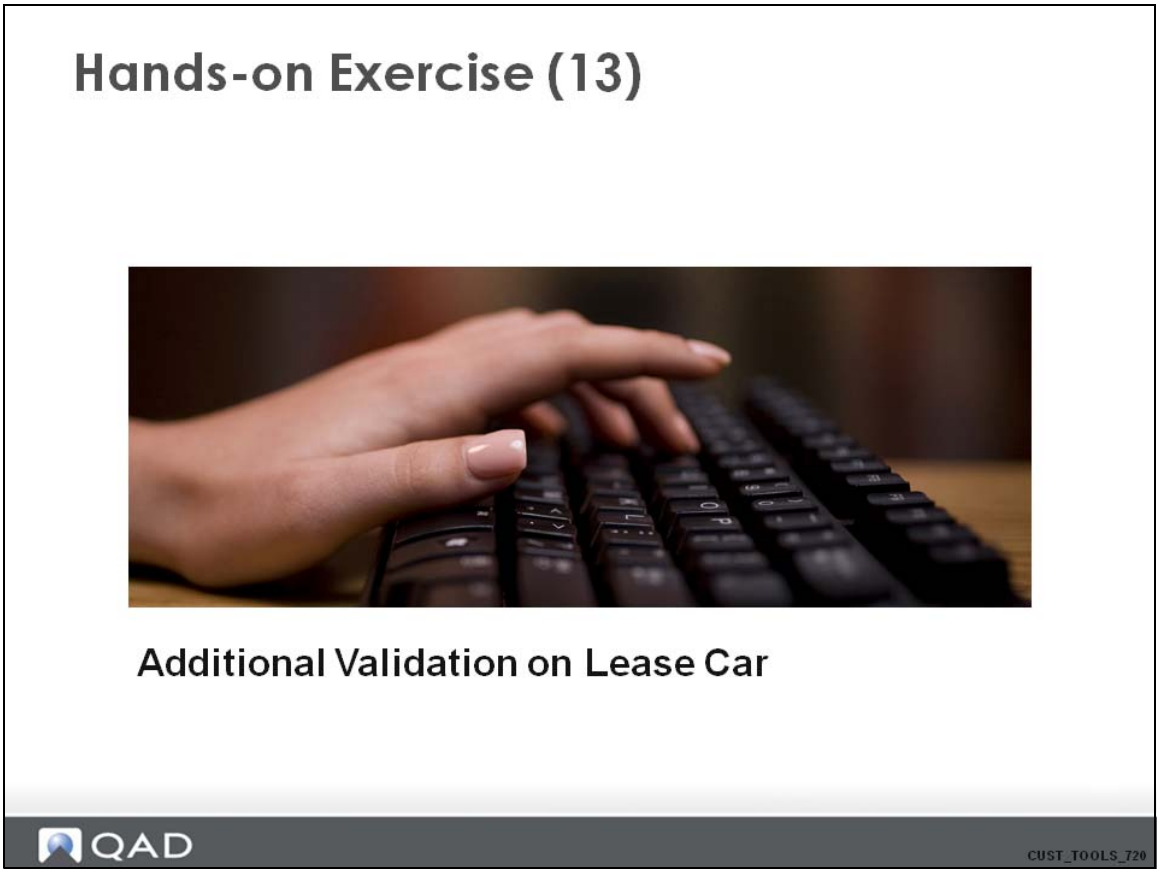


By default, you cannot insert new lines in your grid.

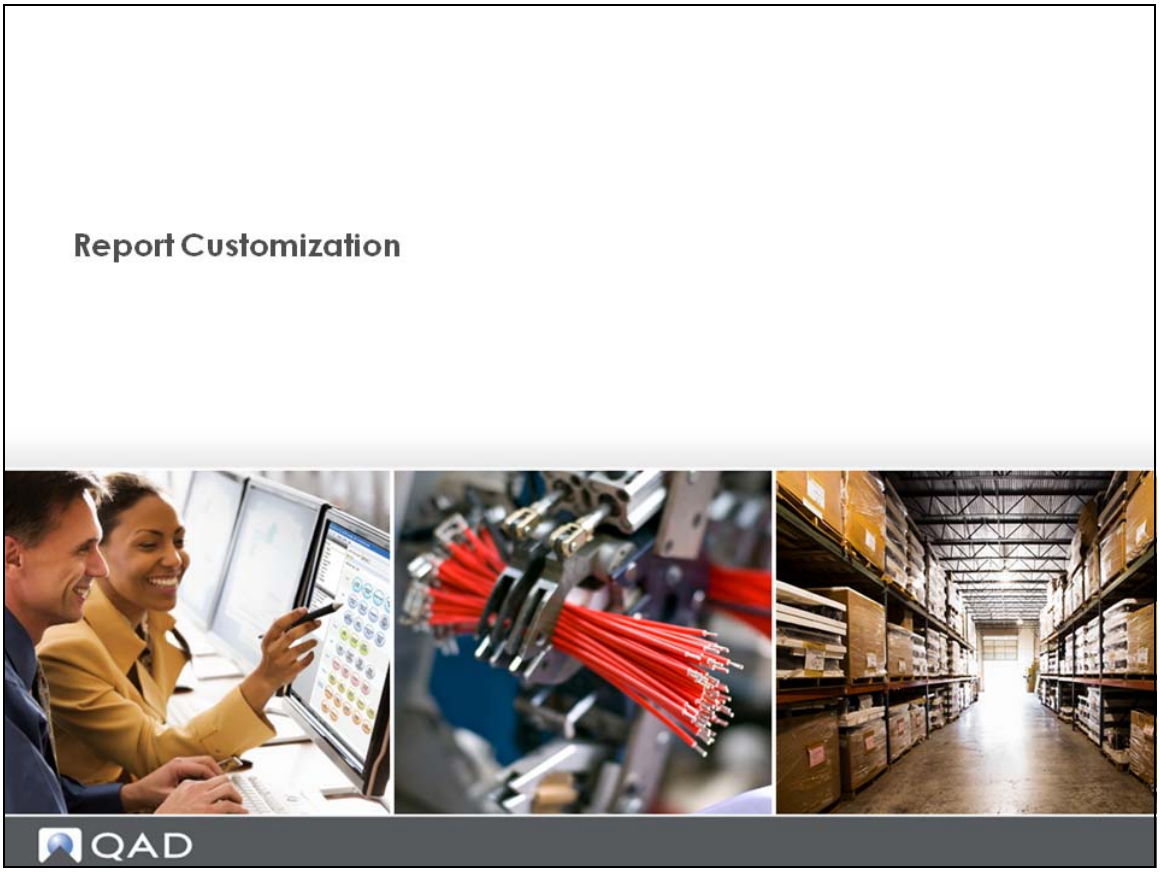
Do not forget to set the “AllowAddNew” property of your grid to “True”.

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)



Report Customization



Report Framework for Financials Reports

Report Framework for Financials Reports

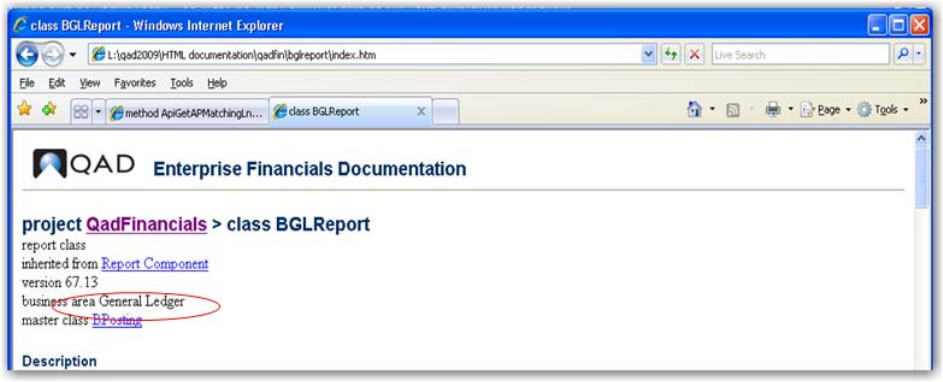
Report Framework for Financials Reports

Report Framework for Financials Reports

Report business component development

Inheritance from abstract foundation component

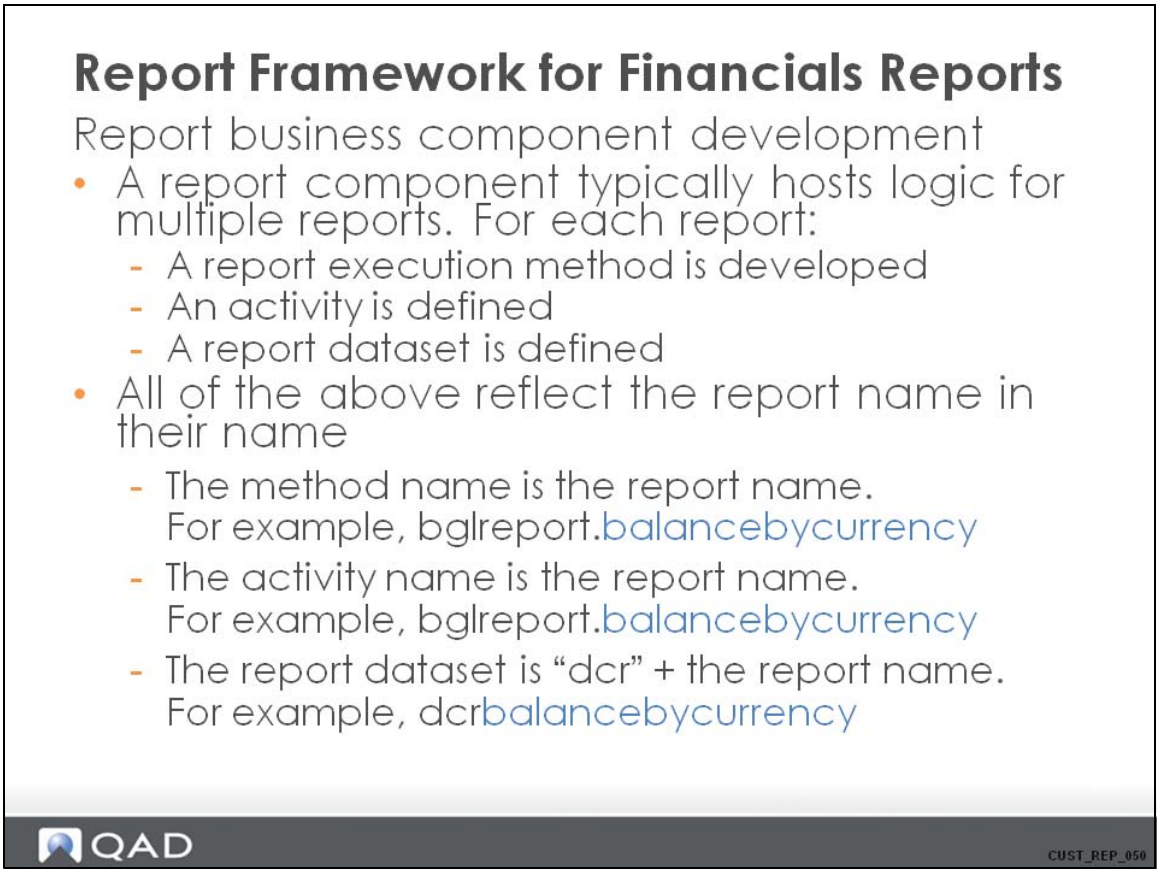
Report Component



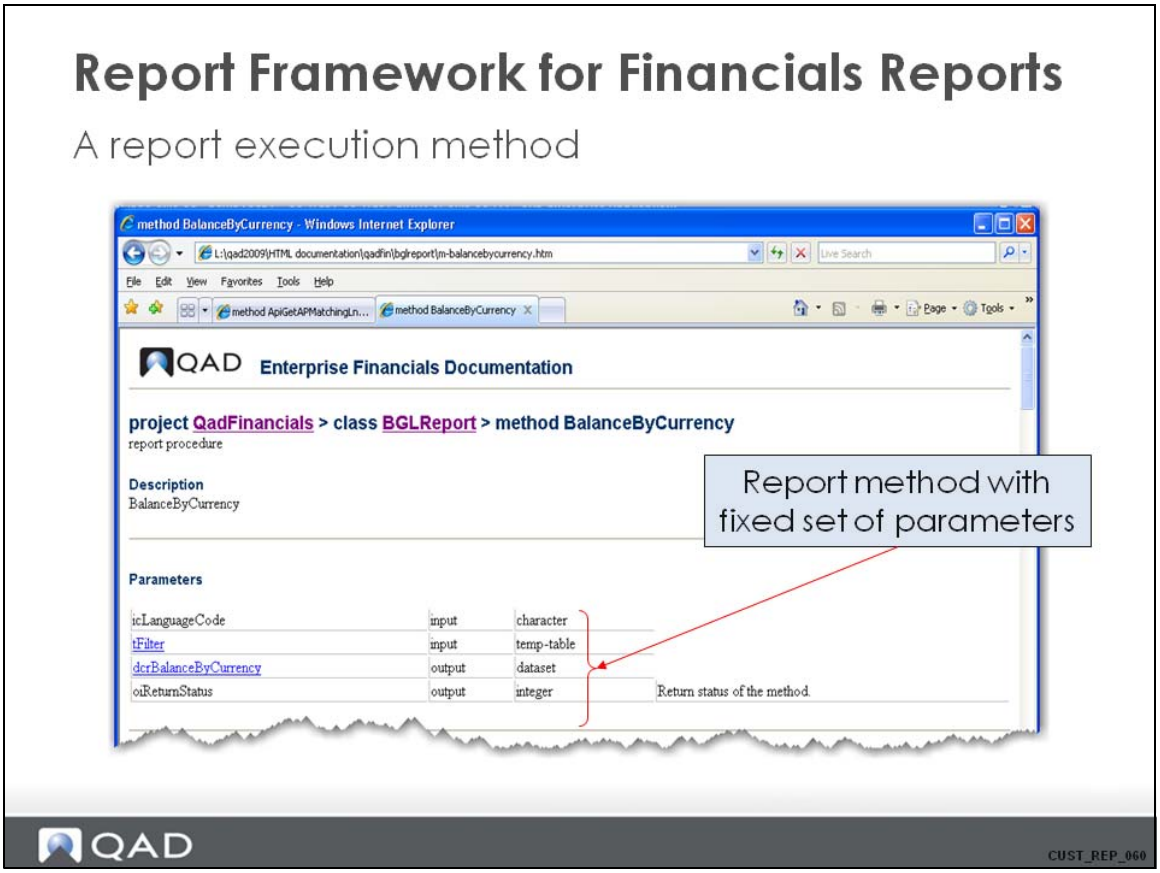
QAD

CUST_REP_040

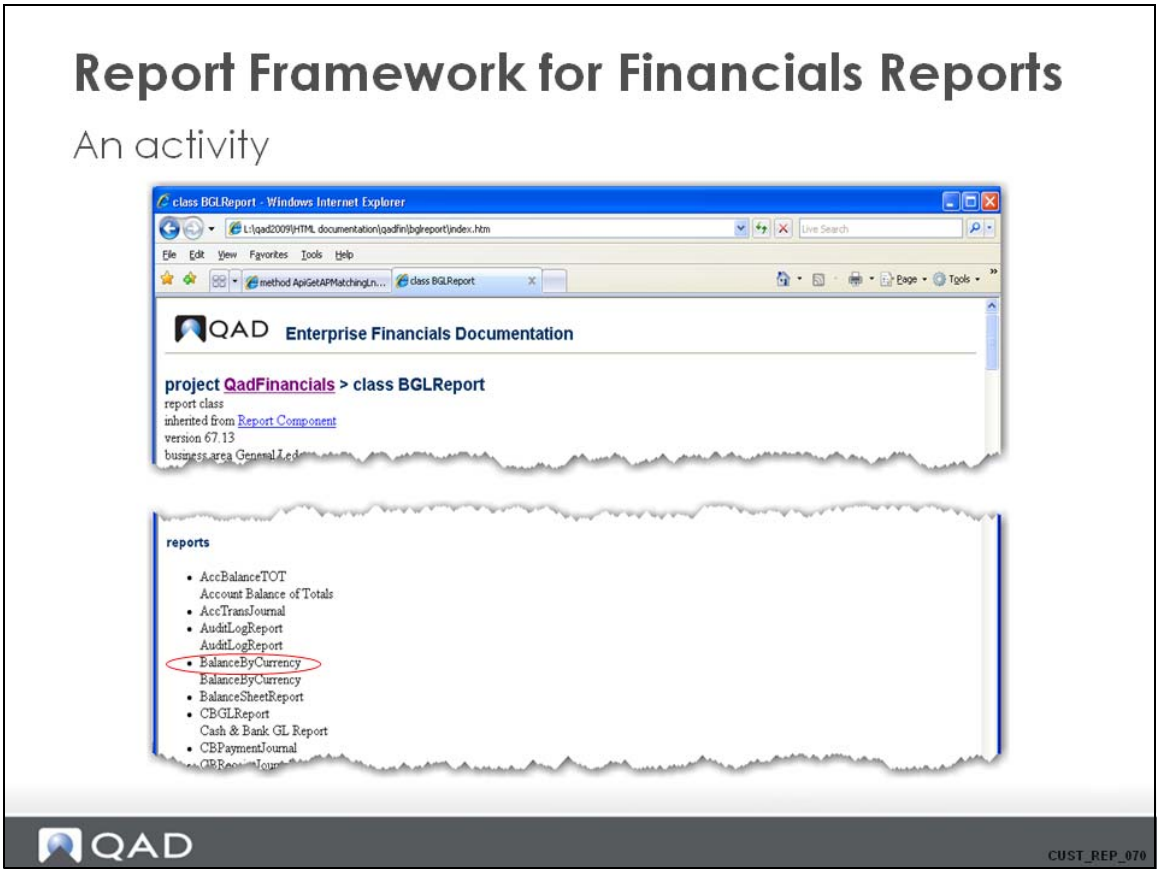
Report Framework for Financials Reports



Report Framework for Financials Reports

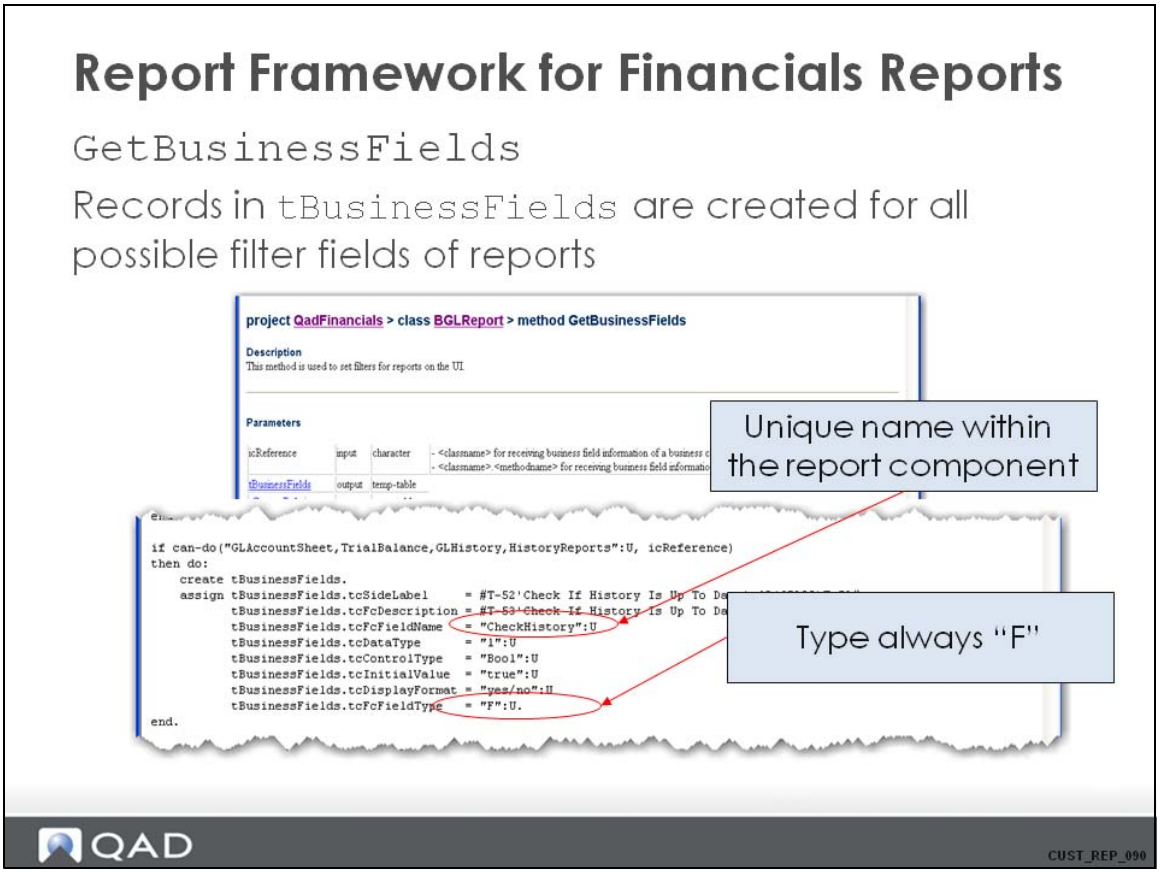


Report Framework for Financials Reports



Report Framework for Financials Reports

Report Framework for Financials Reports



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

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

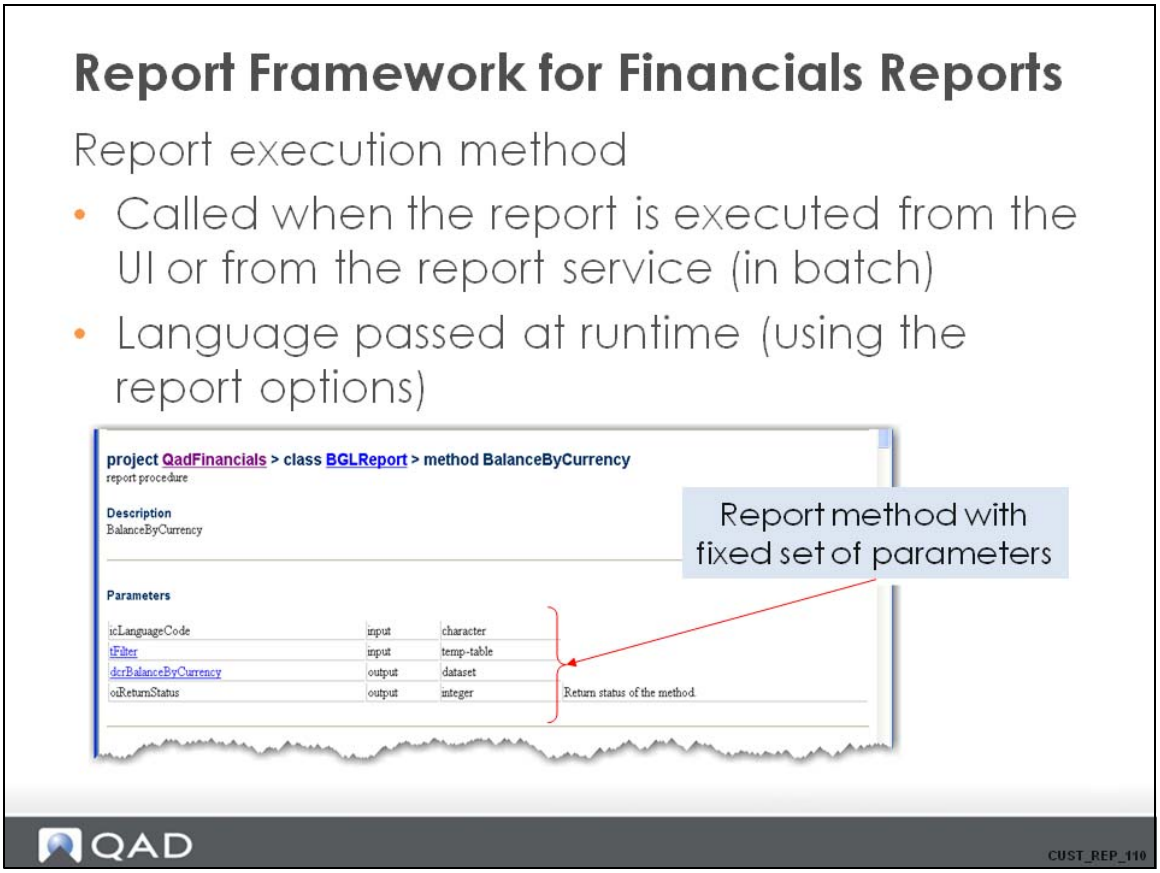
Description

```
for each tFilter:
    case tFilter.tcBusinessFieldName:
        when "SortBy":U
        then do:
            if tFilter.tcParameterValue = "Posting Date":U
            then assign vcSortByFilter = "1" no-error.

            if tFilter.tcParameterValue = "Sequence Number":U
            then assign vcSortByFilter = "2" no-error.
        end.

        when "AccYear":U
        then assign viFromAccYearFilter = int(tFilter.tcParameterValue) no-error.

        when "AccPeriod":U
        then assign viFromAccPeriodFilter = int(tFilter.tcParameterValue) no-error.
```

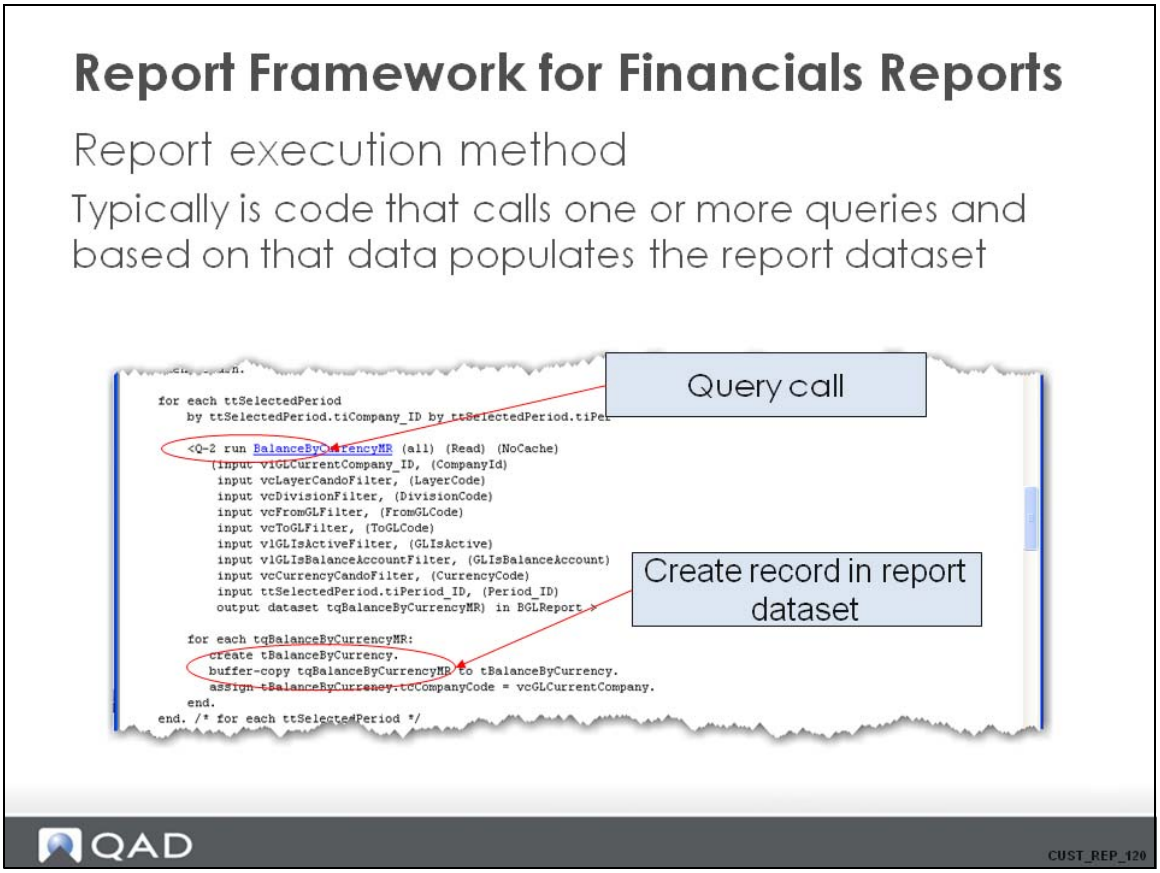
QAD

CUST_REP_100

Report Framework for Financials Reports



Report Framework for Financials Reports



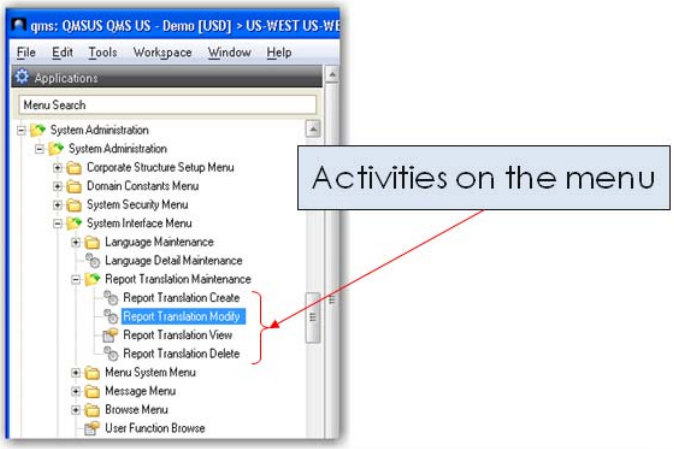
Report Framework for Financials Reports

Report Framework for Financials Reports

Languages and Translations

`BReportTranslation` component

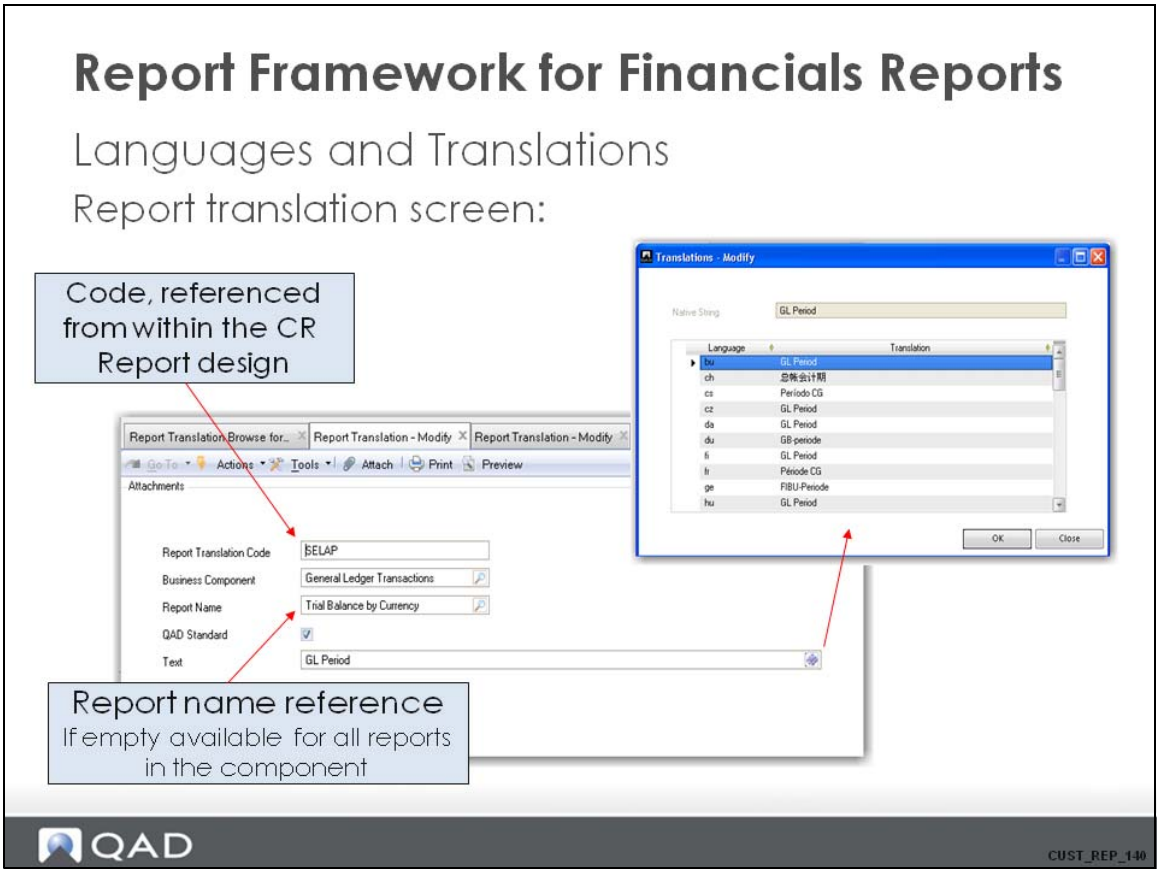
- This component is used to maintain the labels and translations on specific reports
- Menu items:



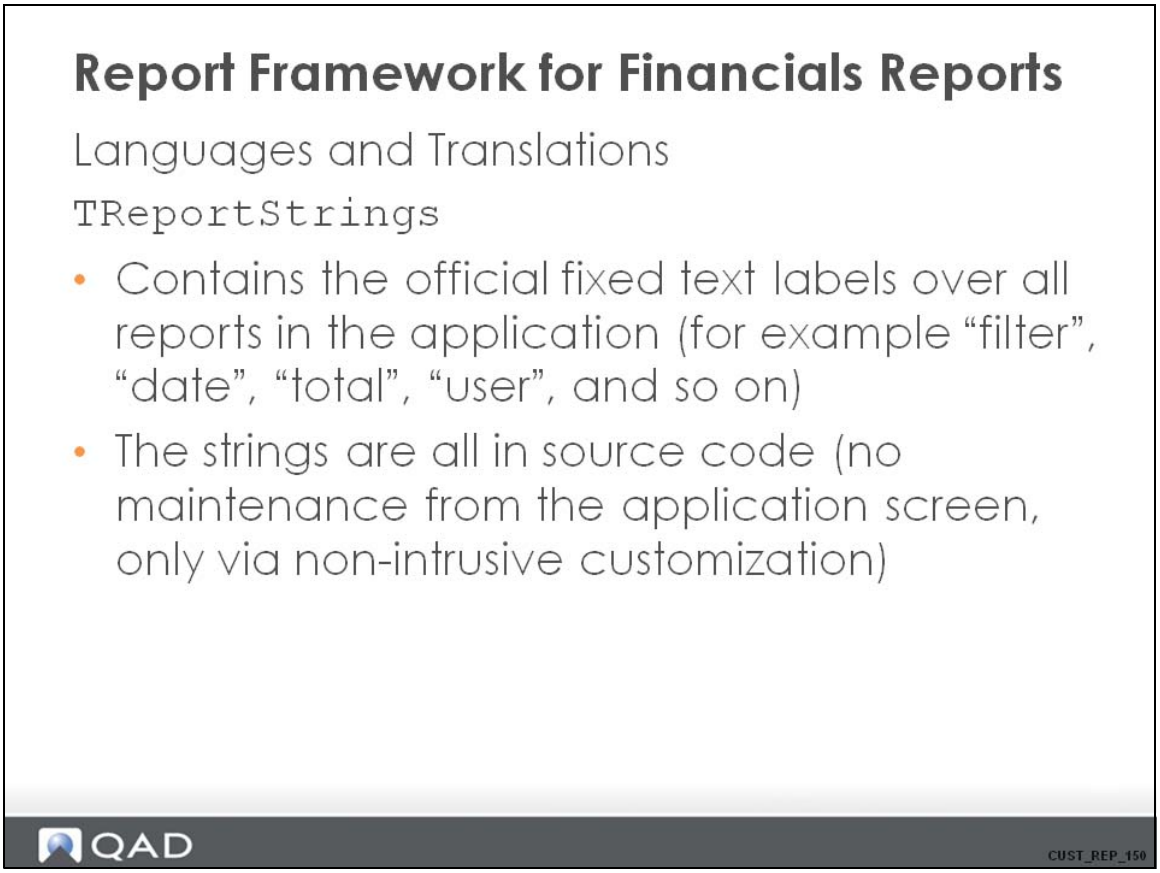
QAD

CUST_REP_130

Report Framework for Financials Reports



Report Framework for Financials Reports



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 = "O-DATE":U.

create tReportStrings.
assign tReportStrings.tcStringText = #T-4'DB':2(359)t-4#
       tReportStrings.tcStringCode = "O-DB":U.

create tReportStrings.
assign tReportStrings.tcStringText = #T-5'End of Report':20(360)t-5#
       tReportStrings.tcStringCode = "O-EOR":U.

create tReportStrings.
assign tReportStrings.tcStringText = #T-6'Filter Values for this Report':100(361)T-6#
       tReportStrings.tcStringCode = "O-FILTER":U.

create tReportStrings.
assign tReportStrings.tcStringText = #T-7'No Data':20(362)t-7#
       tReportStrings.tcStringCode = "O-ND":U.

create tReportStrings.
assign tReportStrings.tcStringText = #T-8'Page':10(363)t-8#
```

QAD

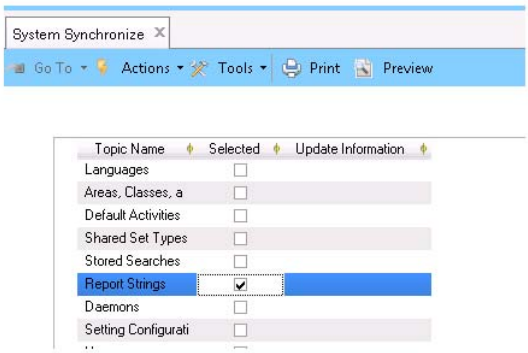
CUST_REP_160

Report Framework for Financials Reports

Report Framework for Financials Reports

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

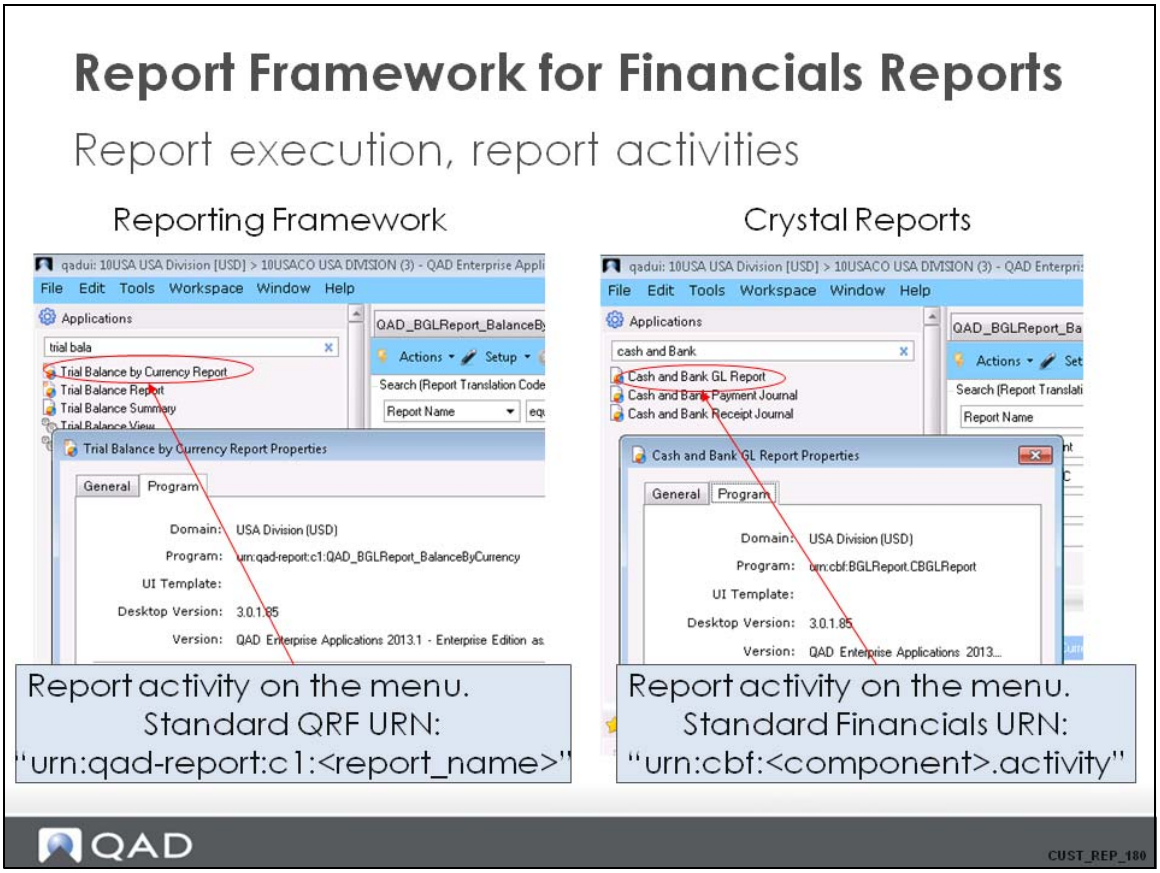
1. System Synchronize (36.24.3.2) for Report Strings
2. Log out from the application and log in again



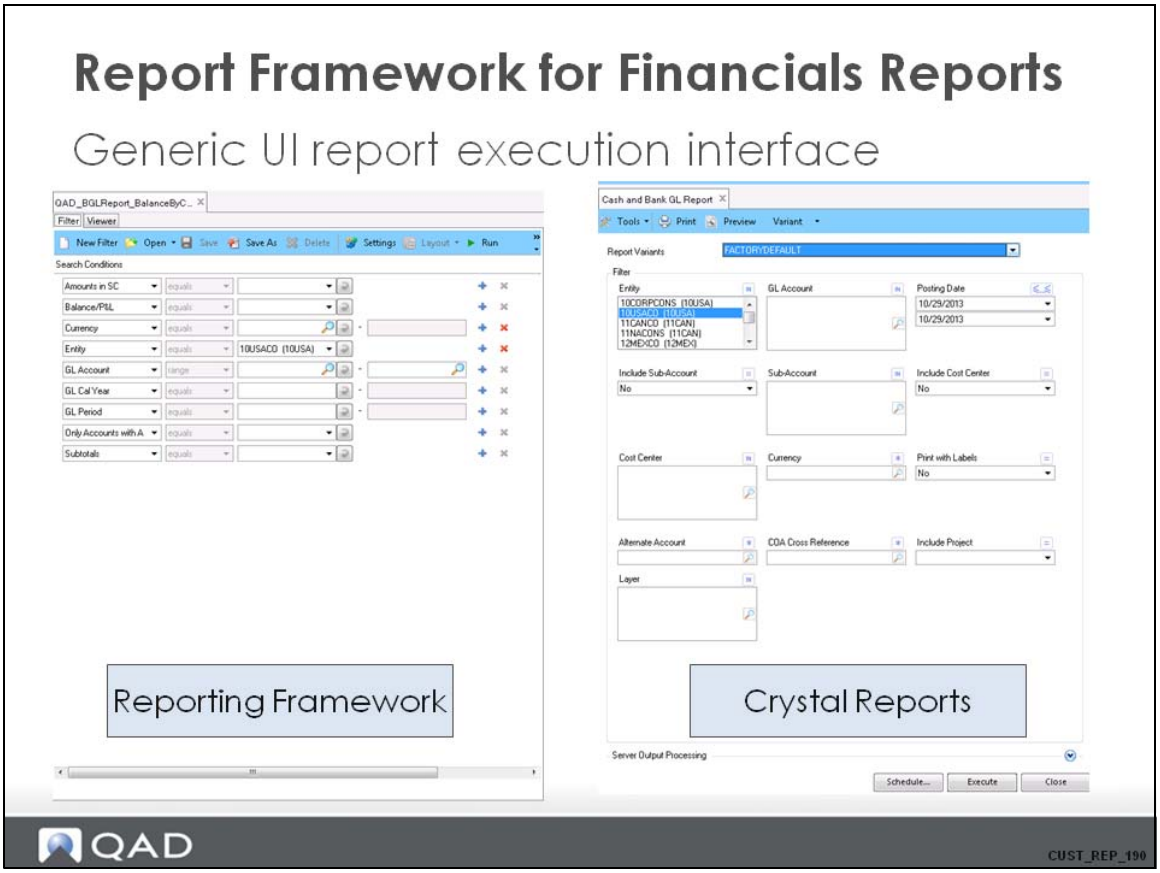
QAD

CUST_REP_170

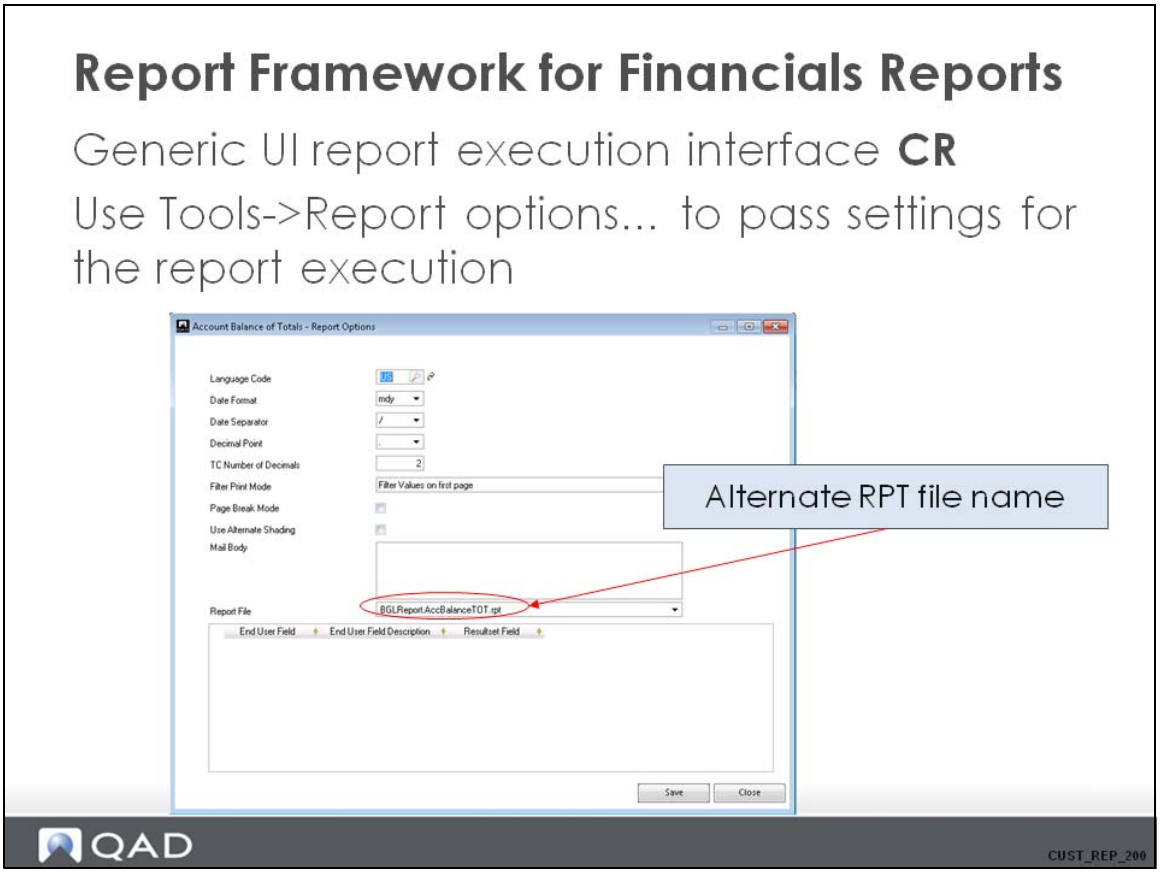
Report Framework for Financials Reports



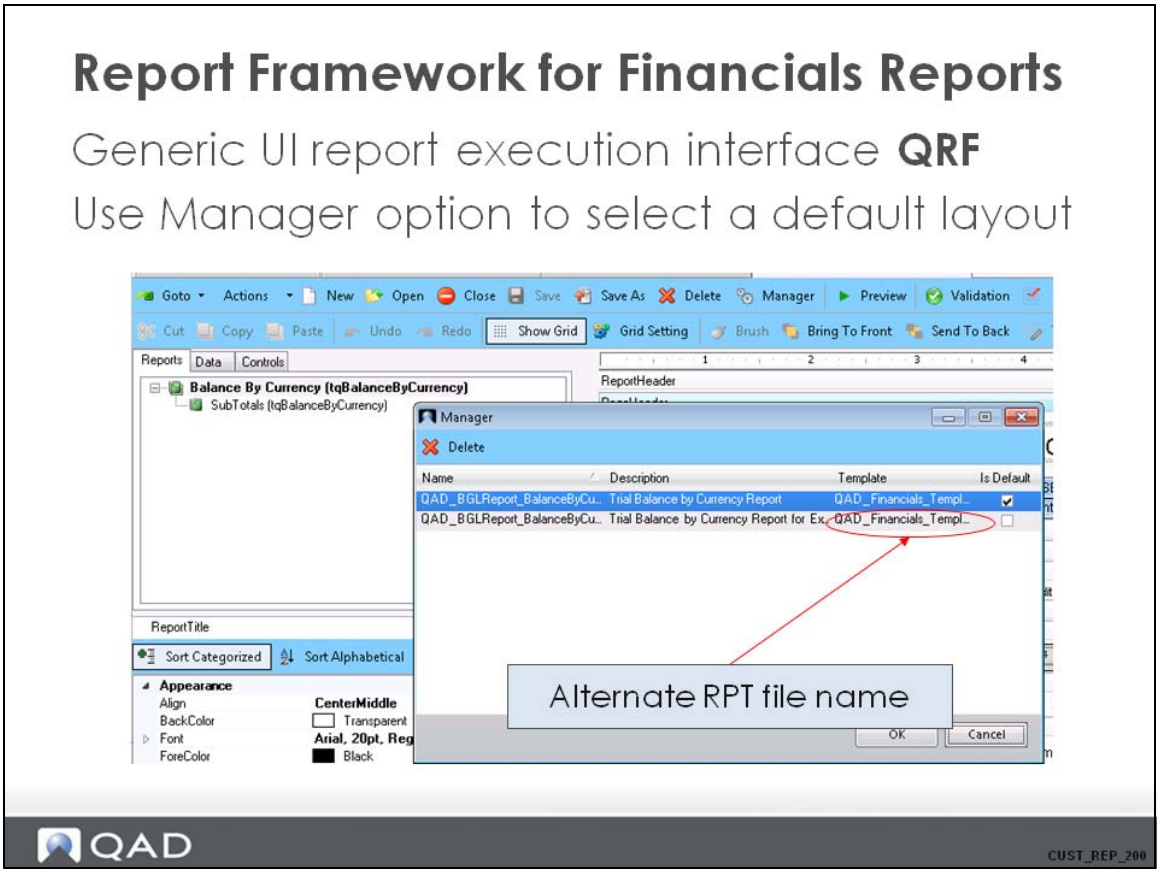
Report Framework for Financials Reports



Report Framework for Financials Reports



Report Framework for Financials Reports

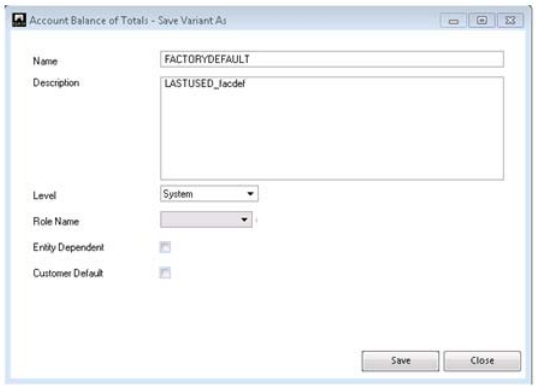


Report Framework for Financials Reports

Report Framework for Financials Reports

Generic UI report execution interface

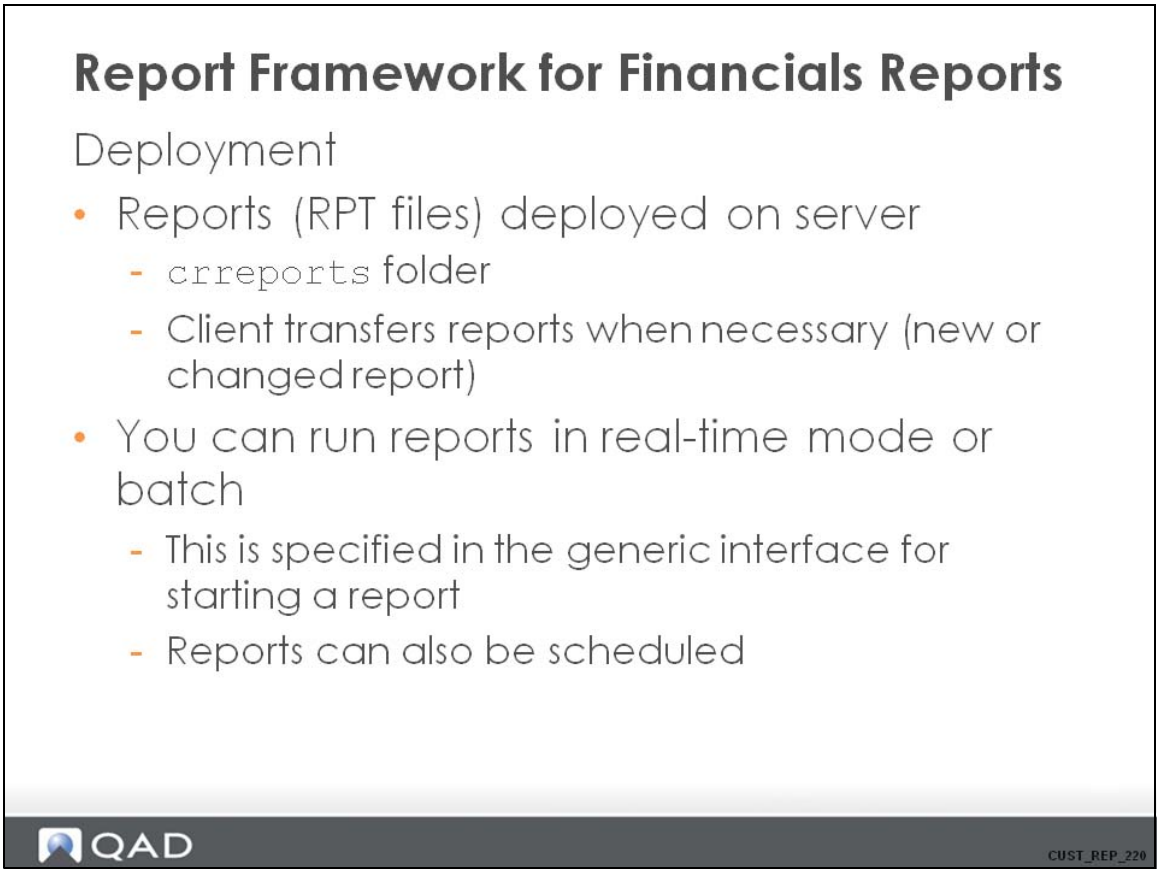
- Settings can be saved as variants.
  (Variant->Save as…)
  Saved data: Report options and filter field settings
- System comes with factory default variant for each report



QAD

CUST_REP_210

Report Framework for Financials Reports



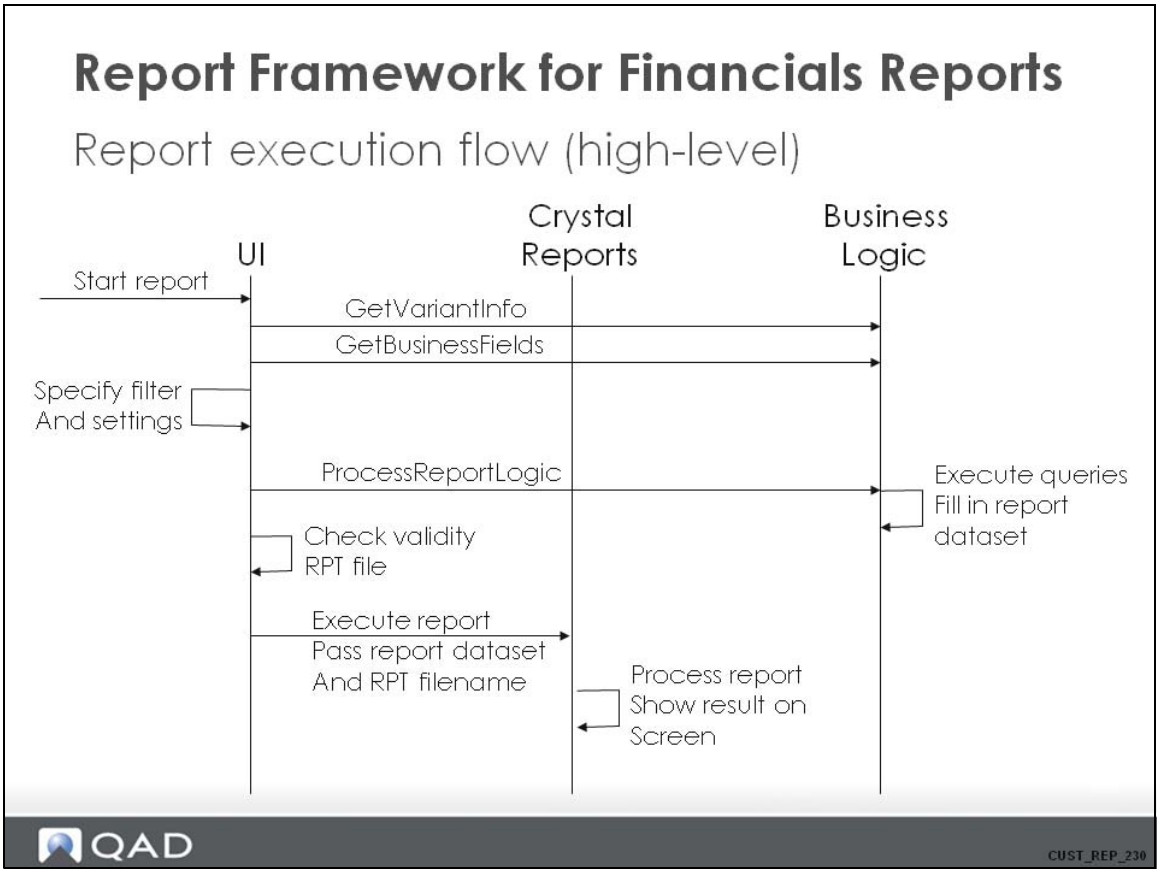
Locate the following folder on the server in the training environment: "`/dr01/qad2009/qms/fin/crreports`".

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.

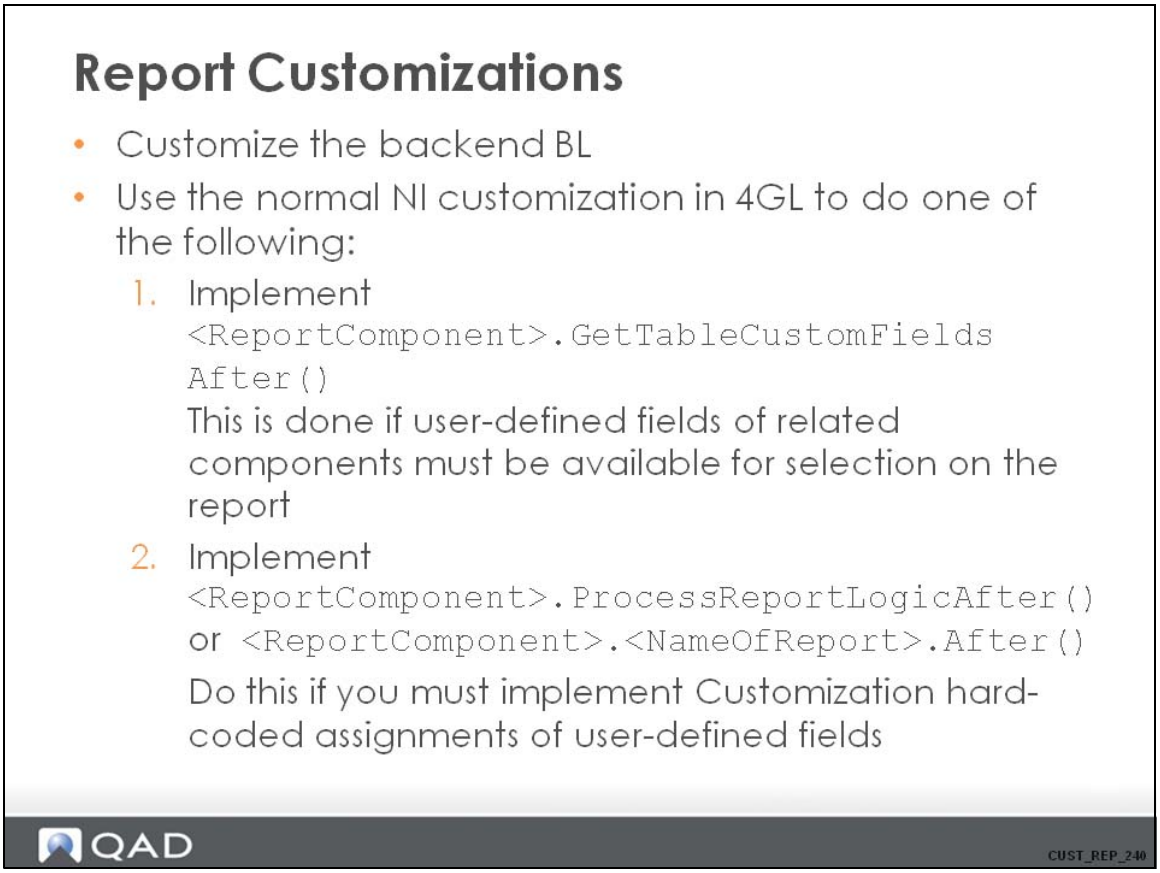
The client transfers the RPT files only when they are to be executed. These files are cached in a client-side folder (`plugins\qad.plugin.Financials\Reports`).

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



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.

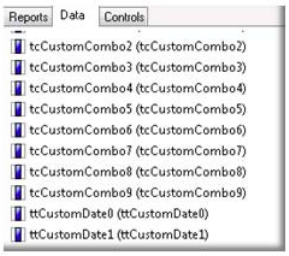
Report Customization - User-Defined Fields

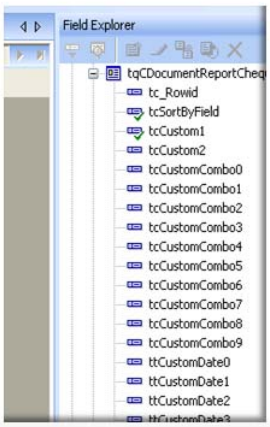
Reports Customization

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



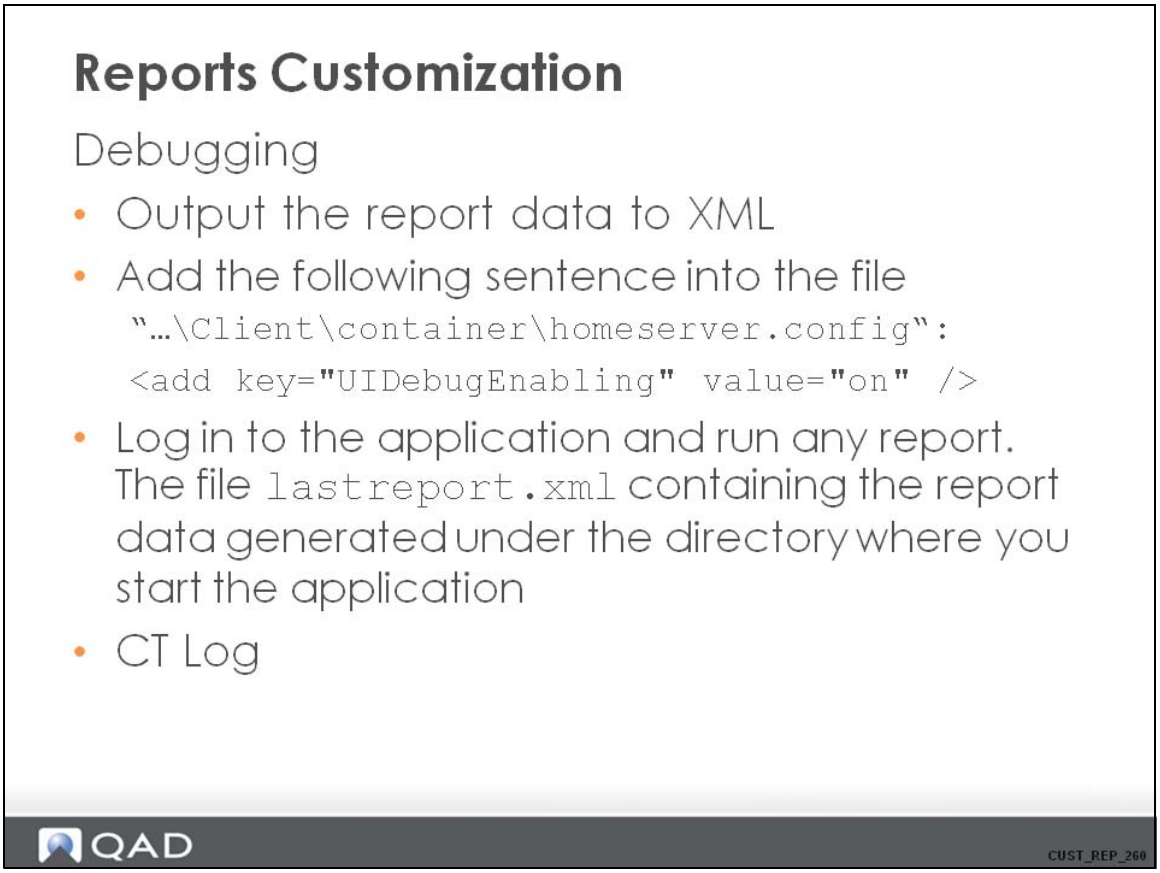


Crystal Reports

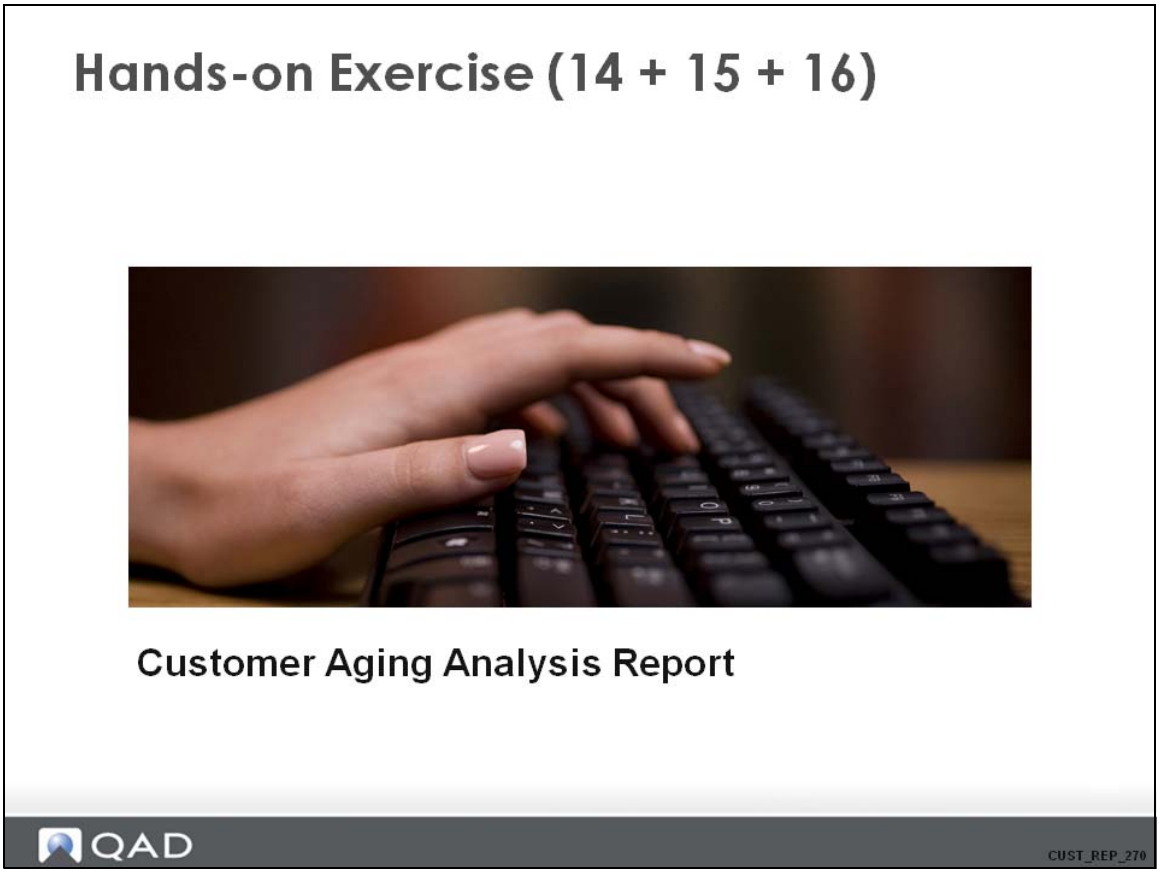
QAD

CUST_REP_250

Report Customization - Debugging



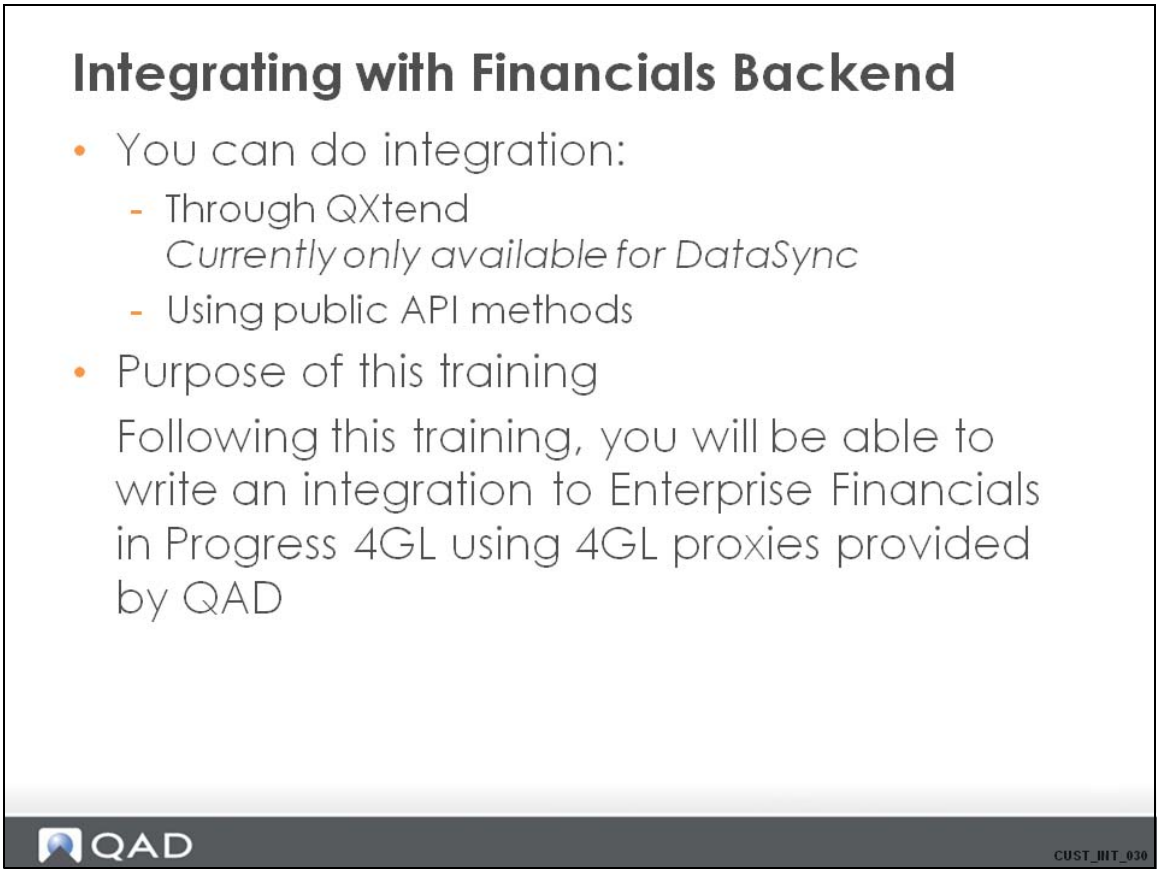
Hands-on Exercise (14 + 15 + 16)



Chapter 4

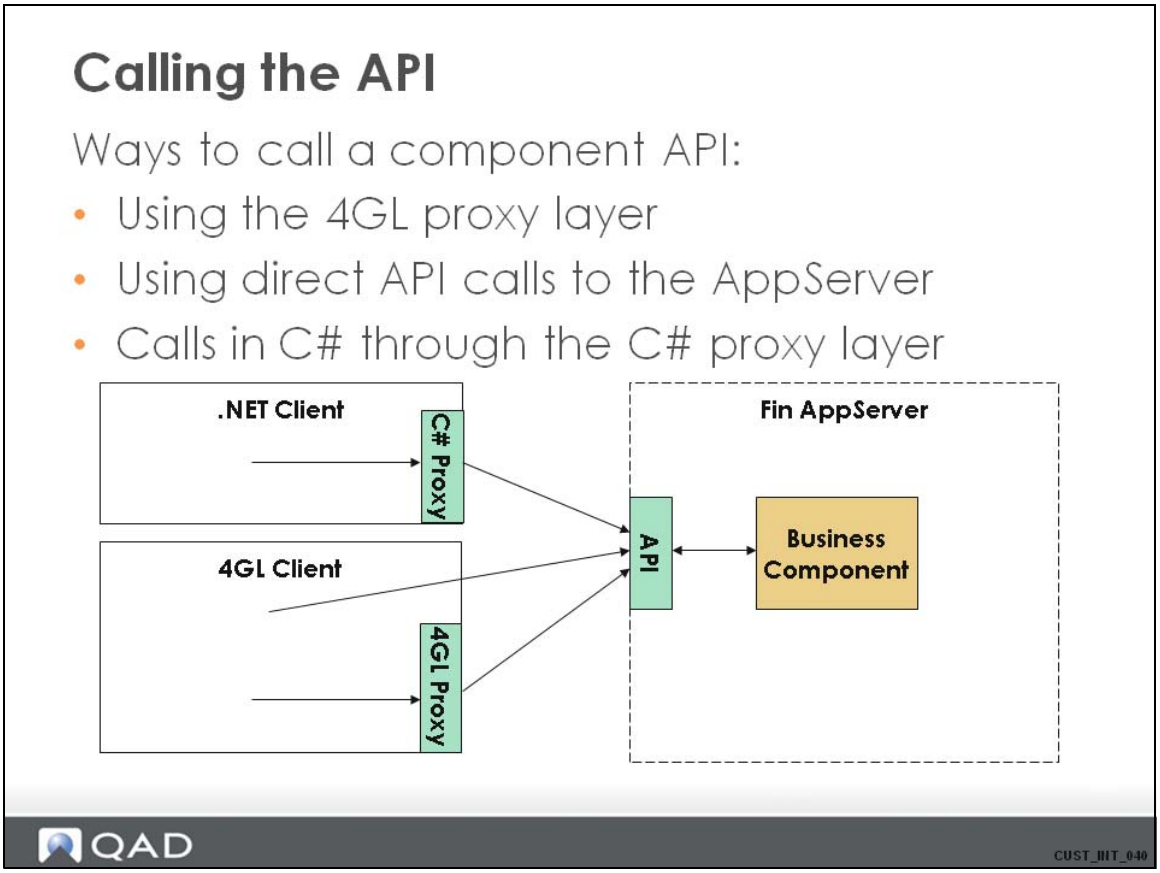
Integration with Enterprise Financials: Backend

Integrating with Financials Backend



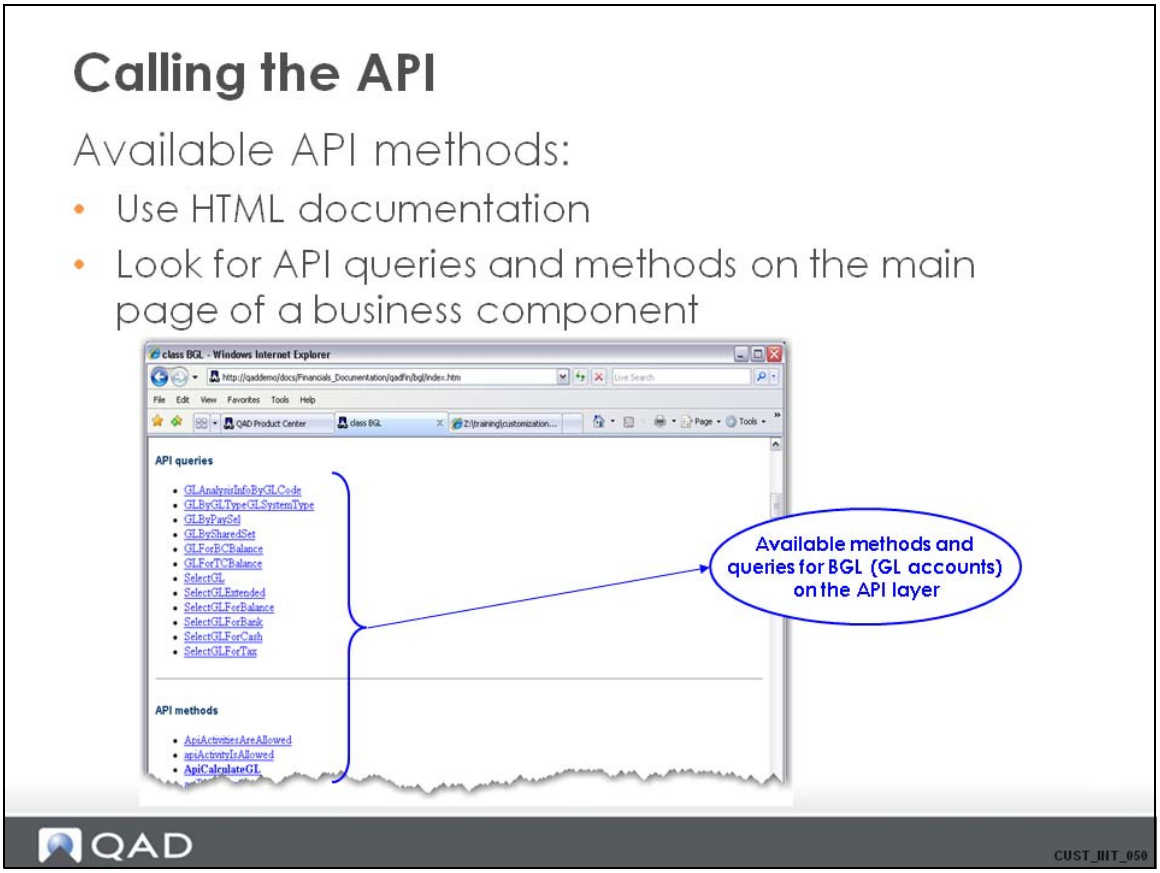
Refer to the QXtend training for information on how to set up and configure data sync for Financials data using QXtend.

Calling the API

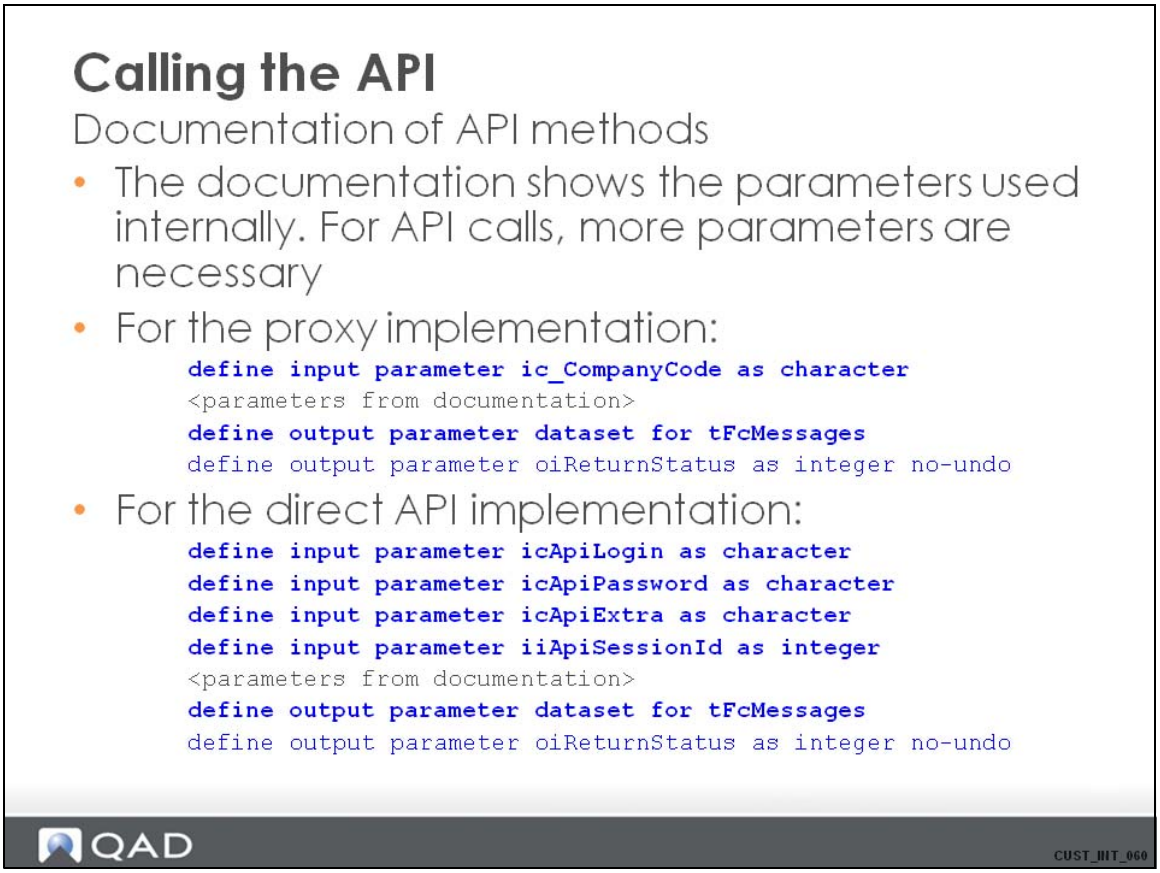


Refer to *Calling API Documentation* for more detailed information and sample code.

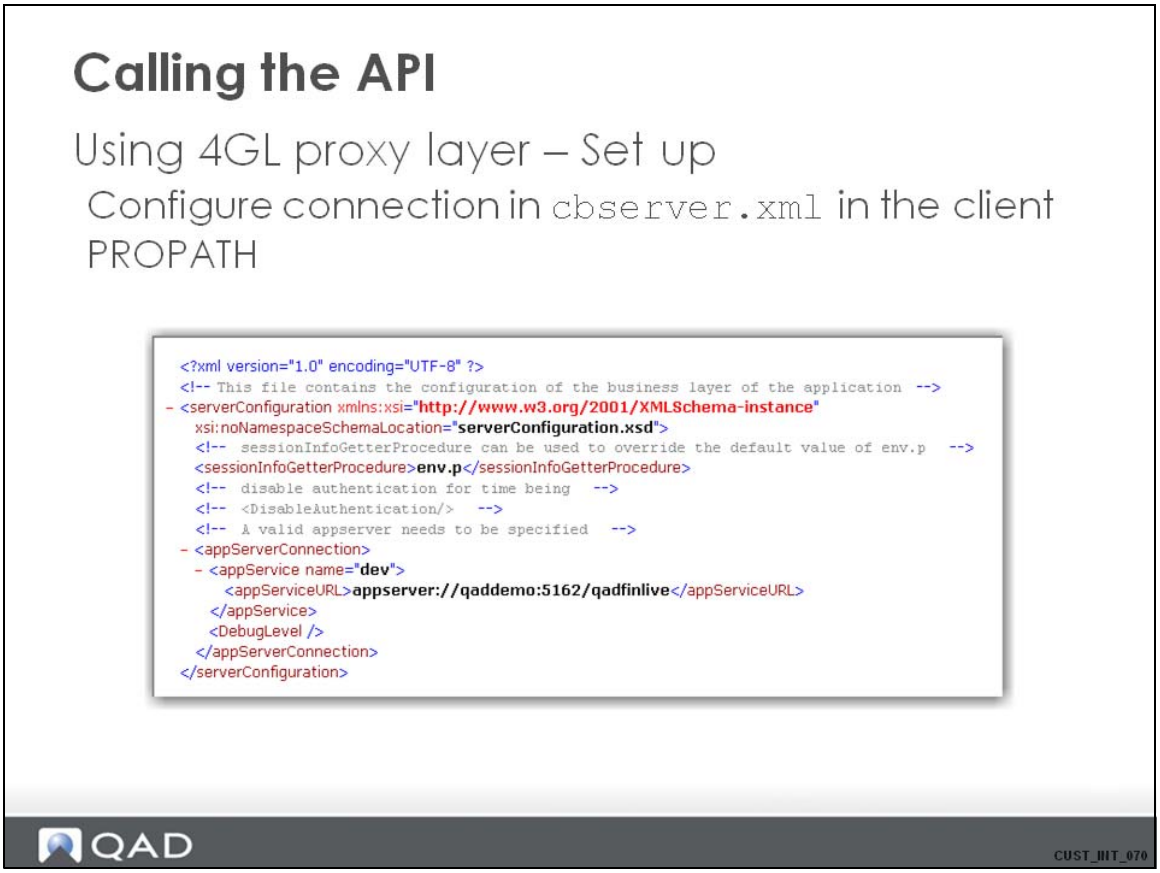
Calling the API



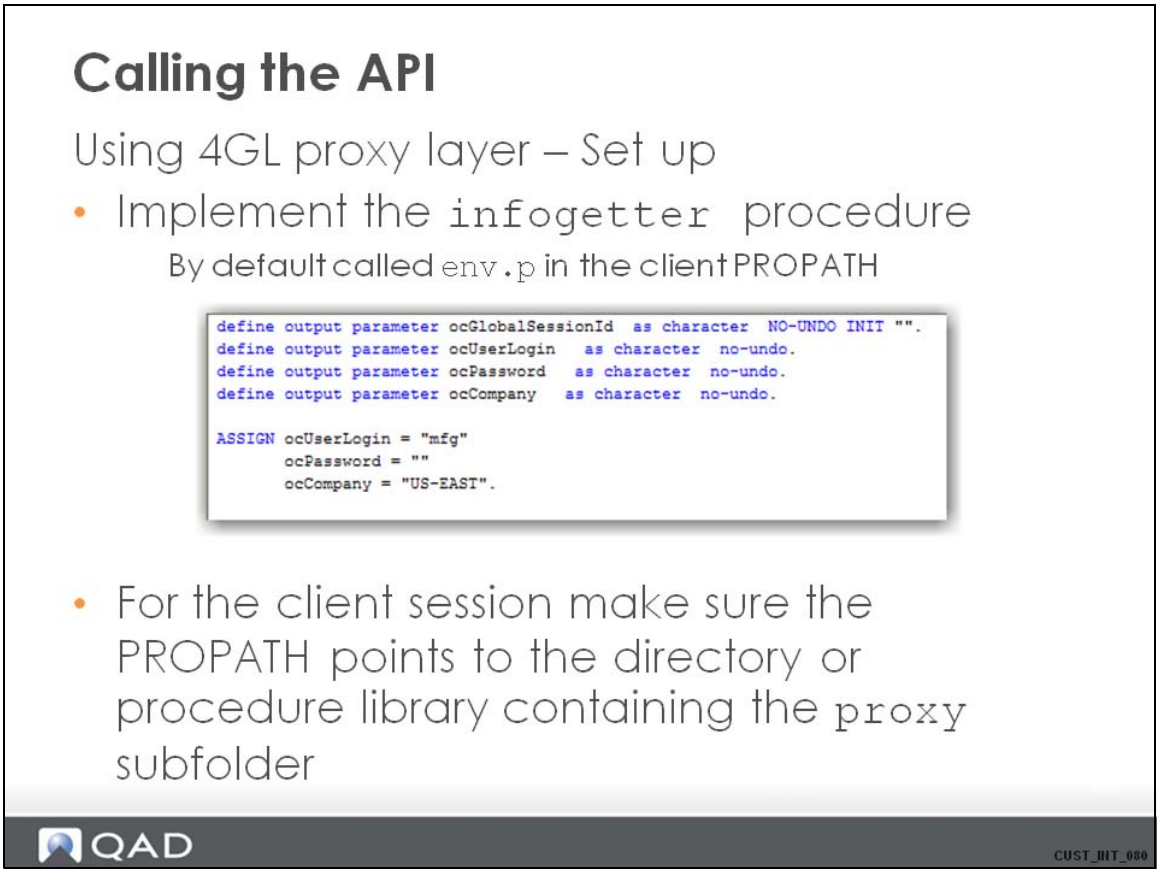
Calling the API



Calling the API

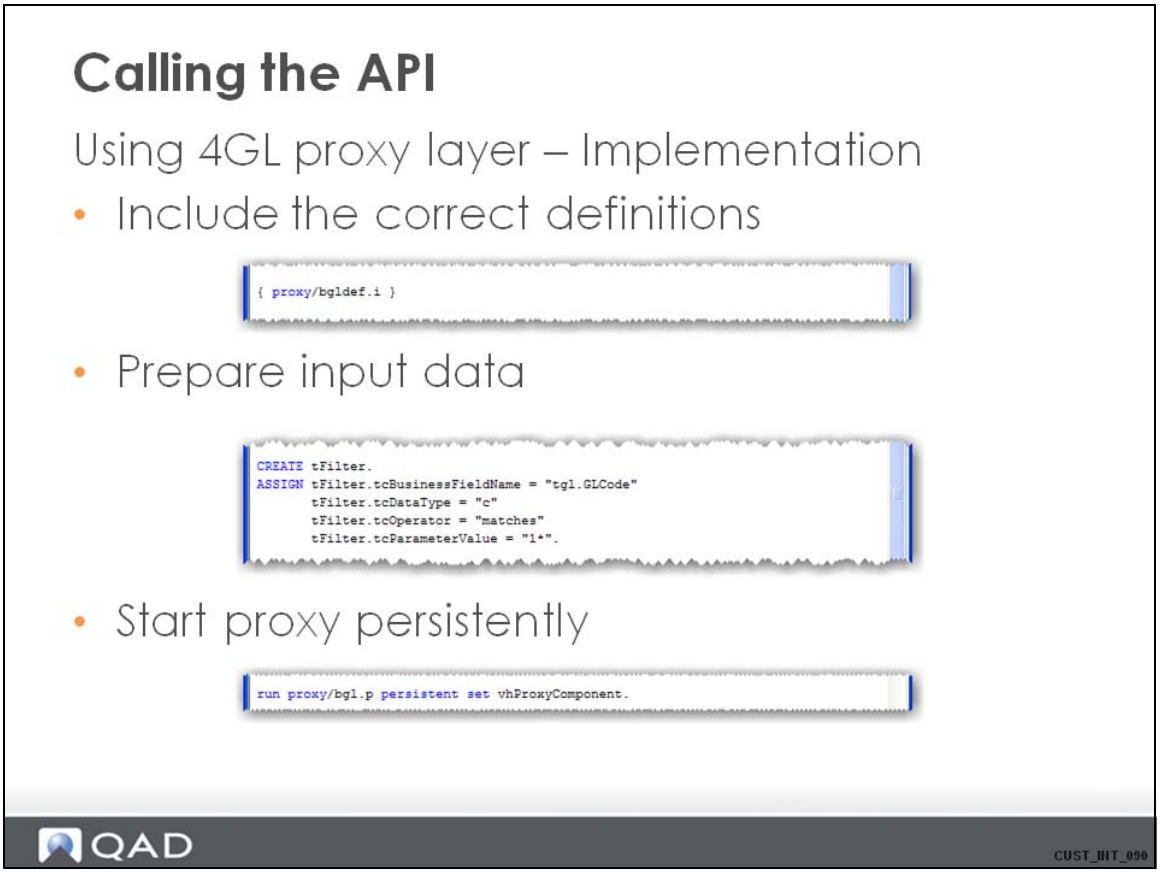


Calling the API

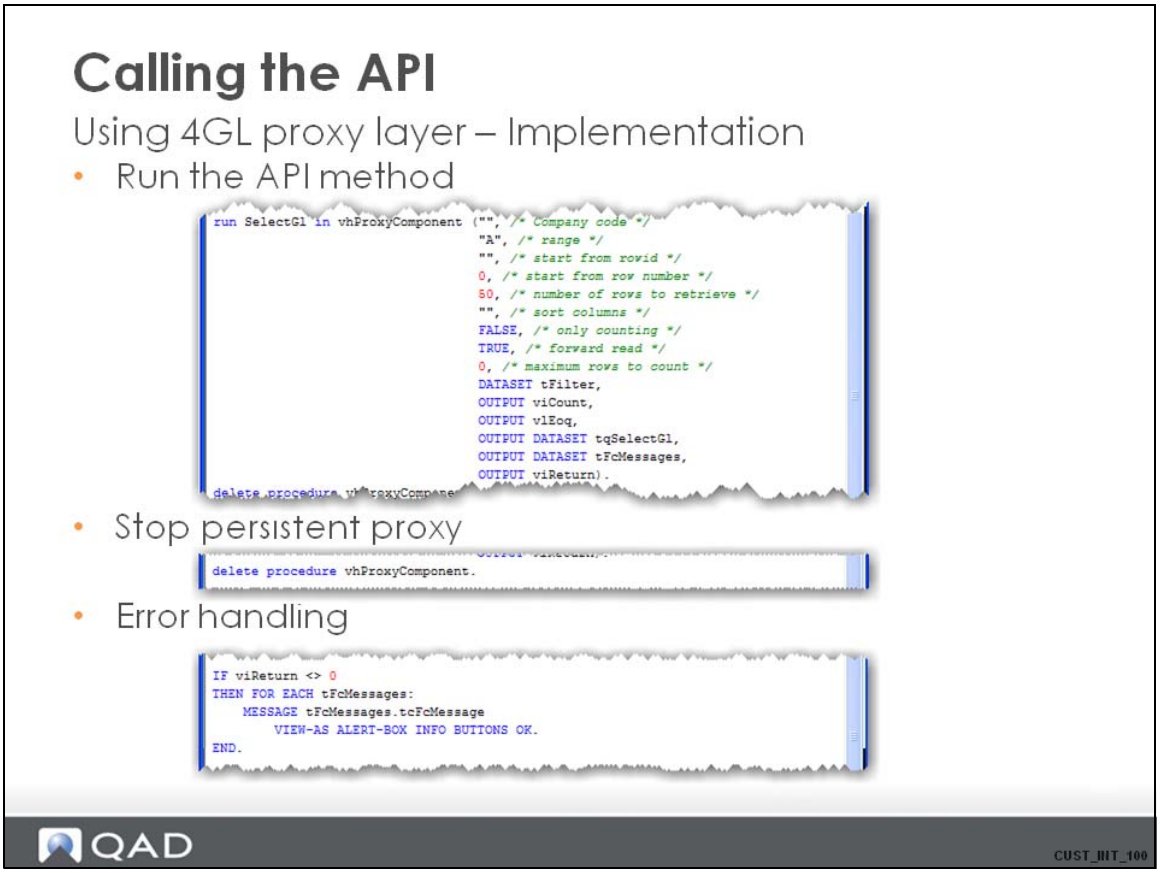


In the training environment, the `PROPATH` should point to "`/dr01/qadapps/qea/fin/proxysrc`".

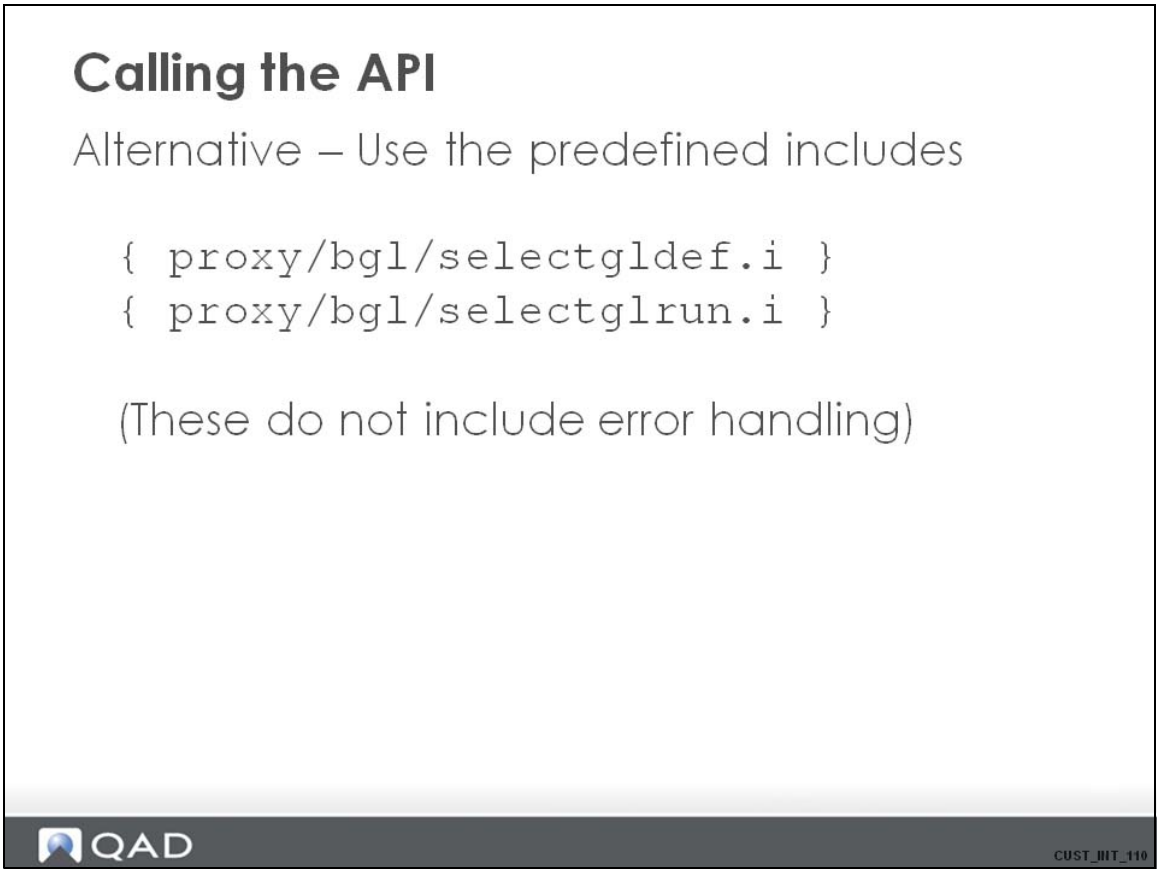
Calling the API



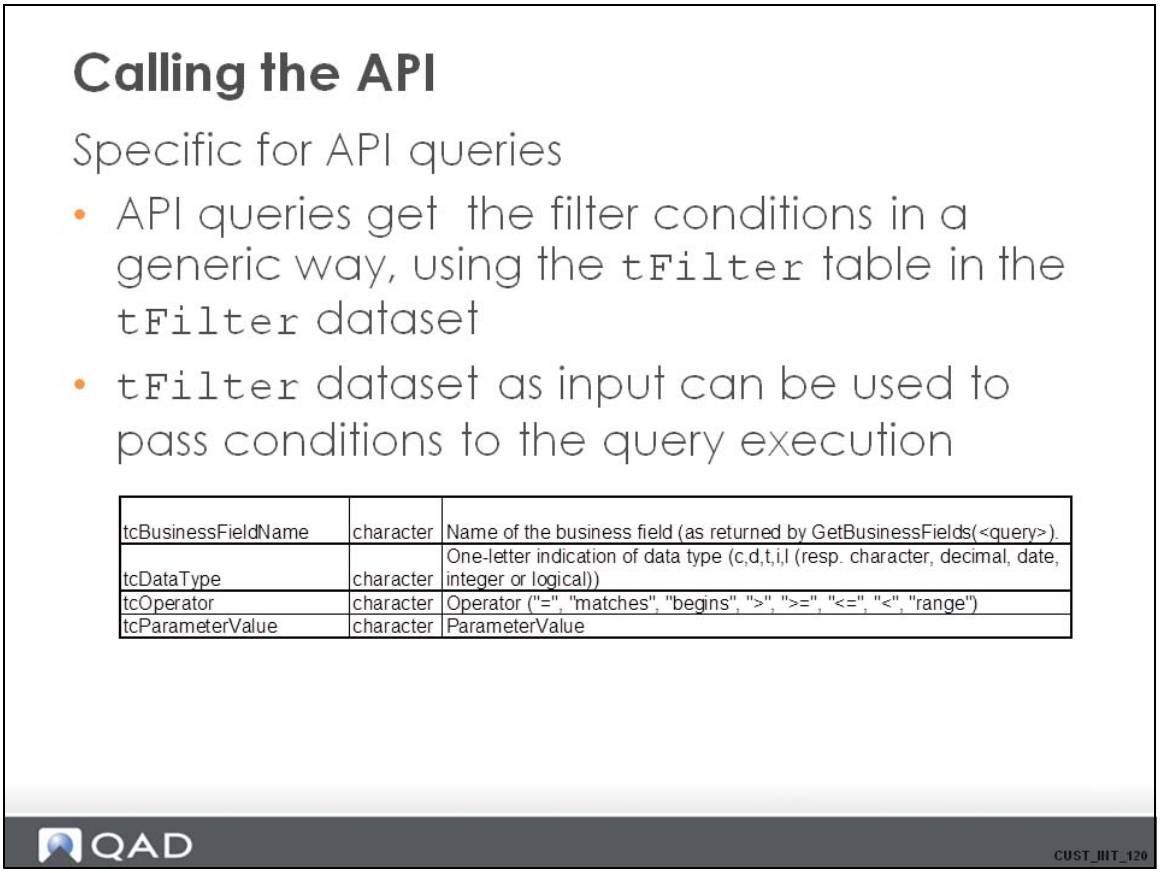
Calling the API



Calling the API



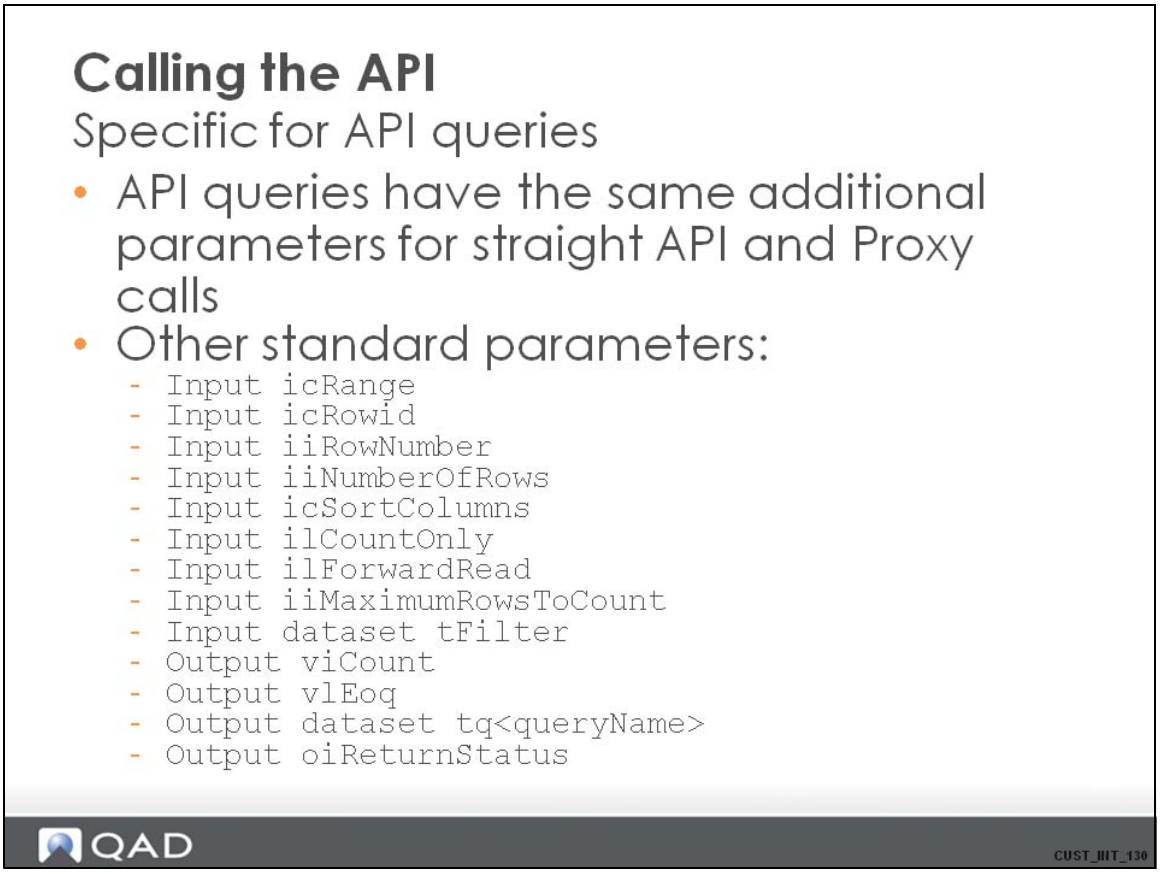
Calling the API



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



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 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 vlEoq,
                  OUTPUT DATASET tqSelectGl,
                  OUTPUT DATASET tFcMessages,
                  OUTPUT viReturn).
```

Calling the API

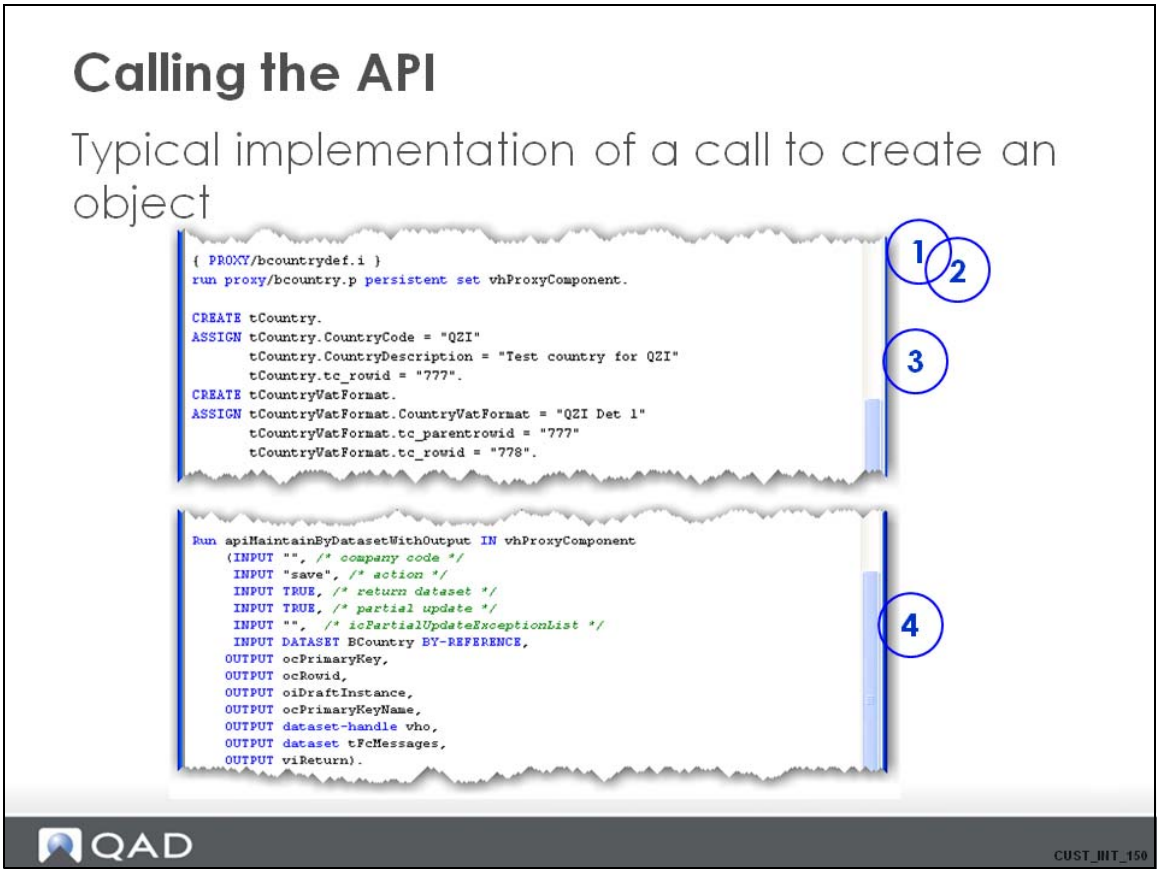
Use the HTML documentation for the classes to find the right method or query.

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



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:

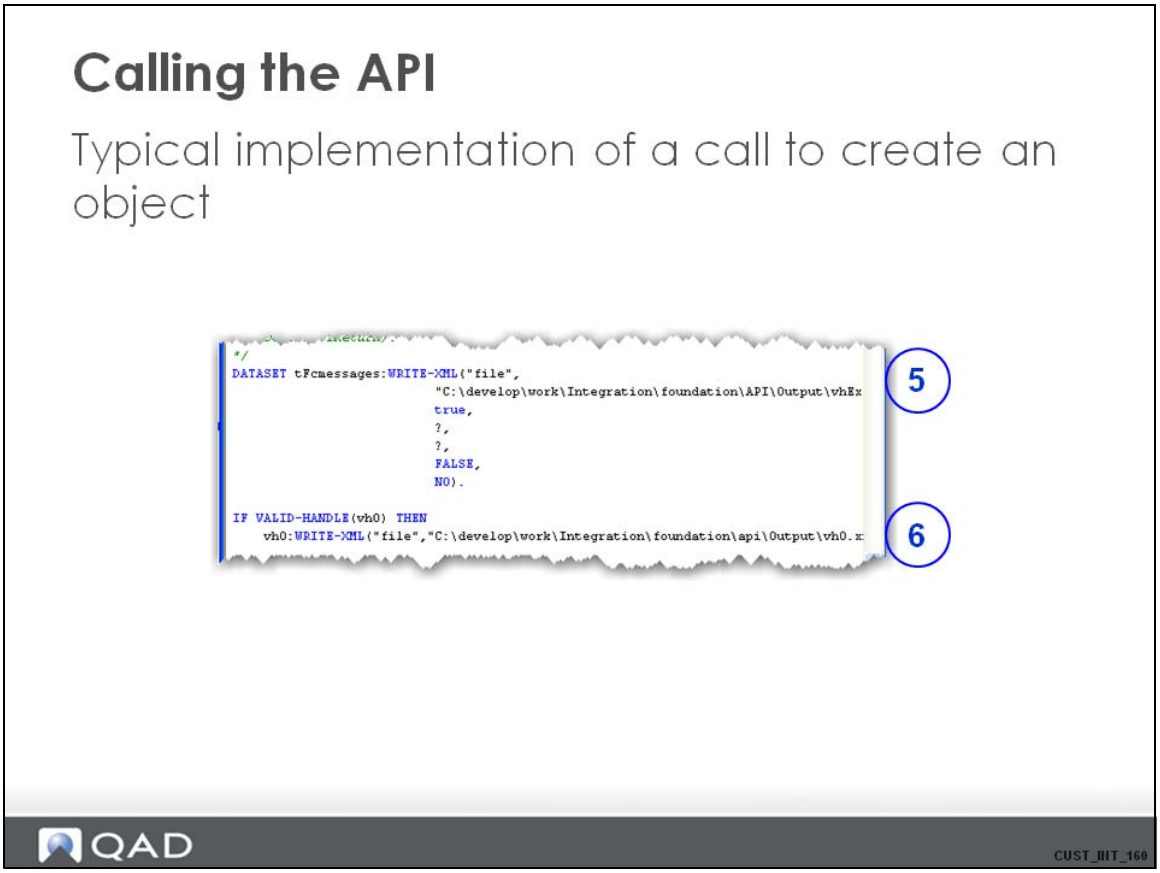
1. Include the definitions for everything required on the client side proxy call.

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.

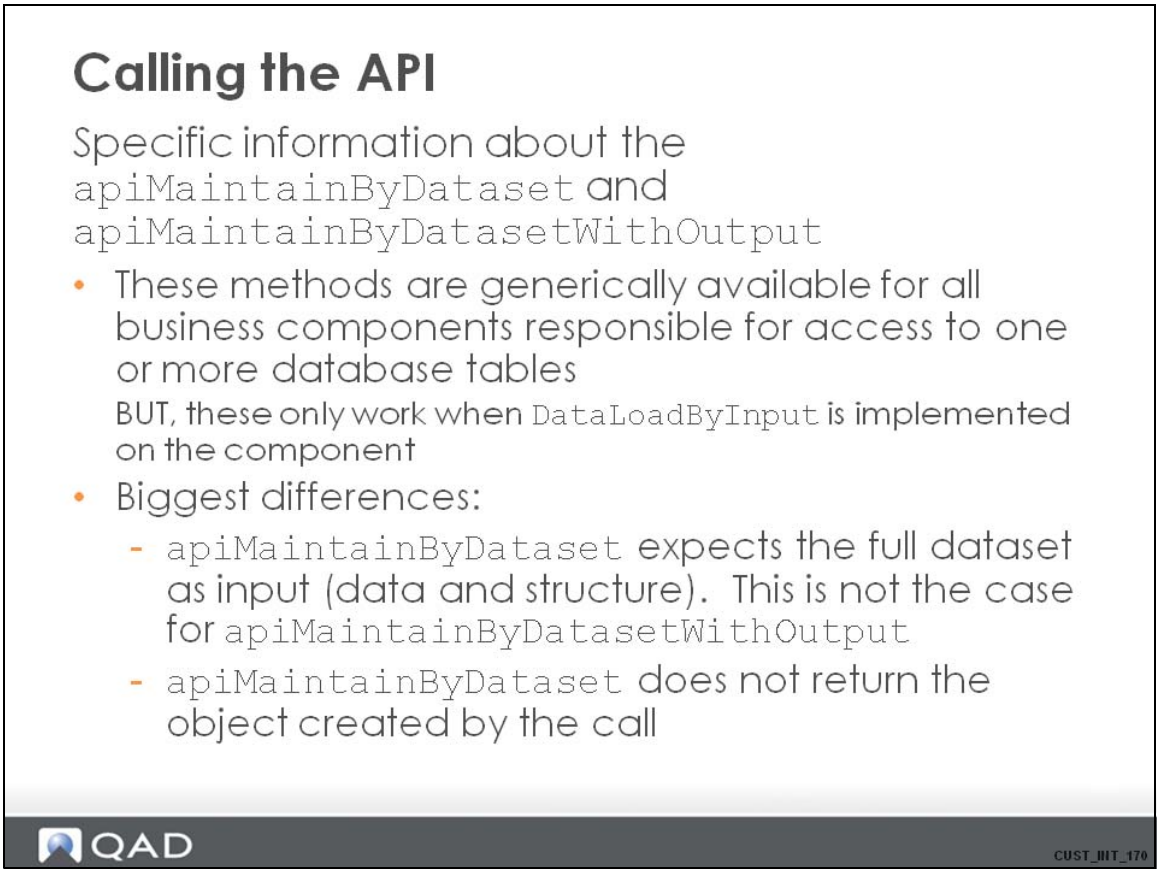
4. Run the ApiMaintainByDatasetWithOutput method.

Calling the API



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



apiMaintainByDataset:

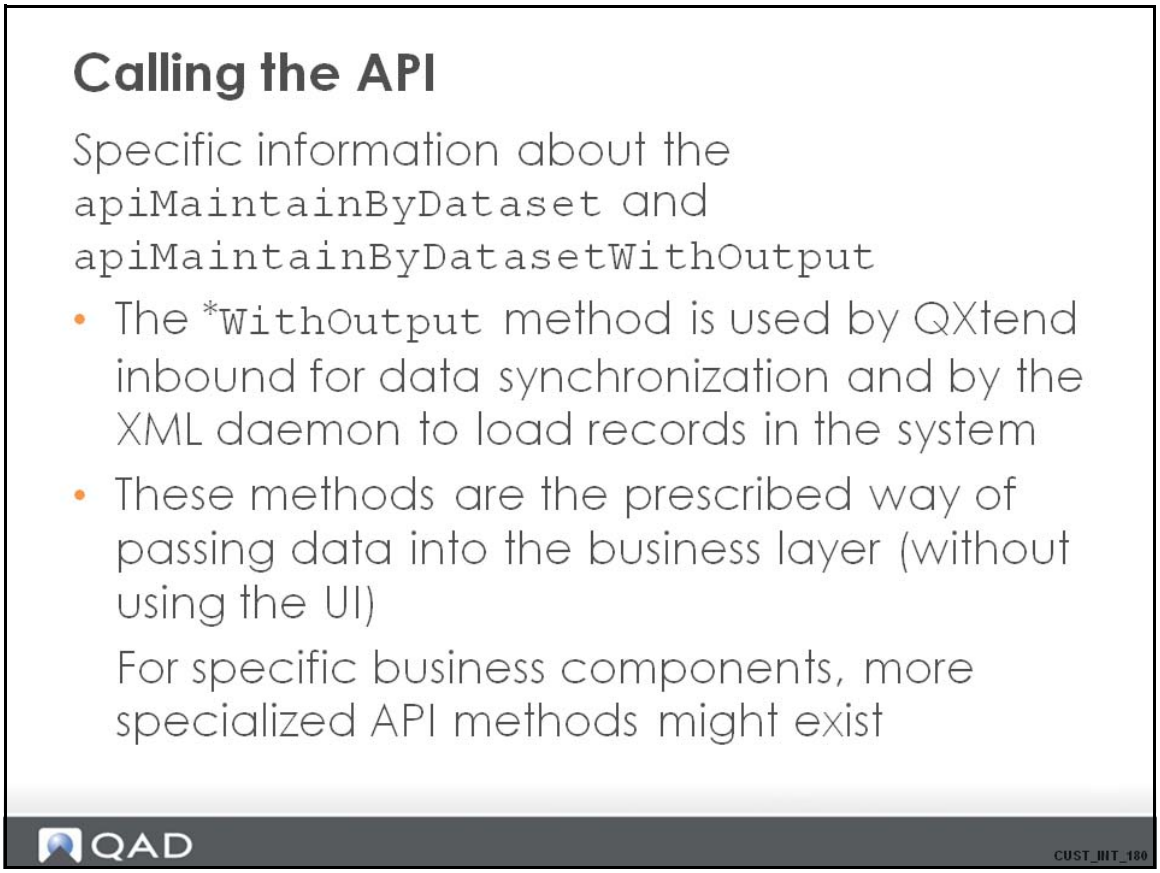
- Expects an input dataset that exactly matches the structure of the object dataset.
- Automatically detects if the object needs to be created or modified based on the alternate key values.
- Expects the full object data. Automatically deletes child records that exist in the database, but do not exist 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:

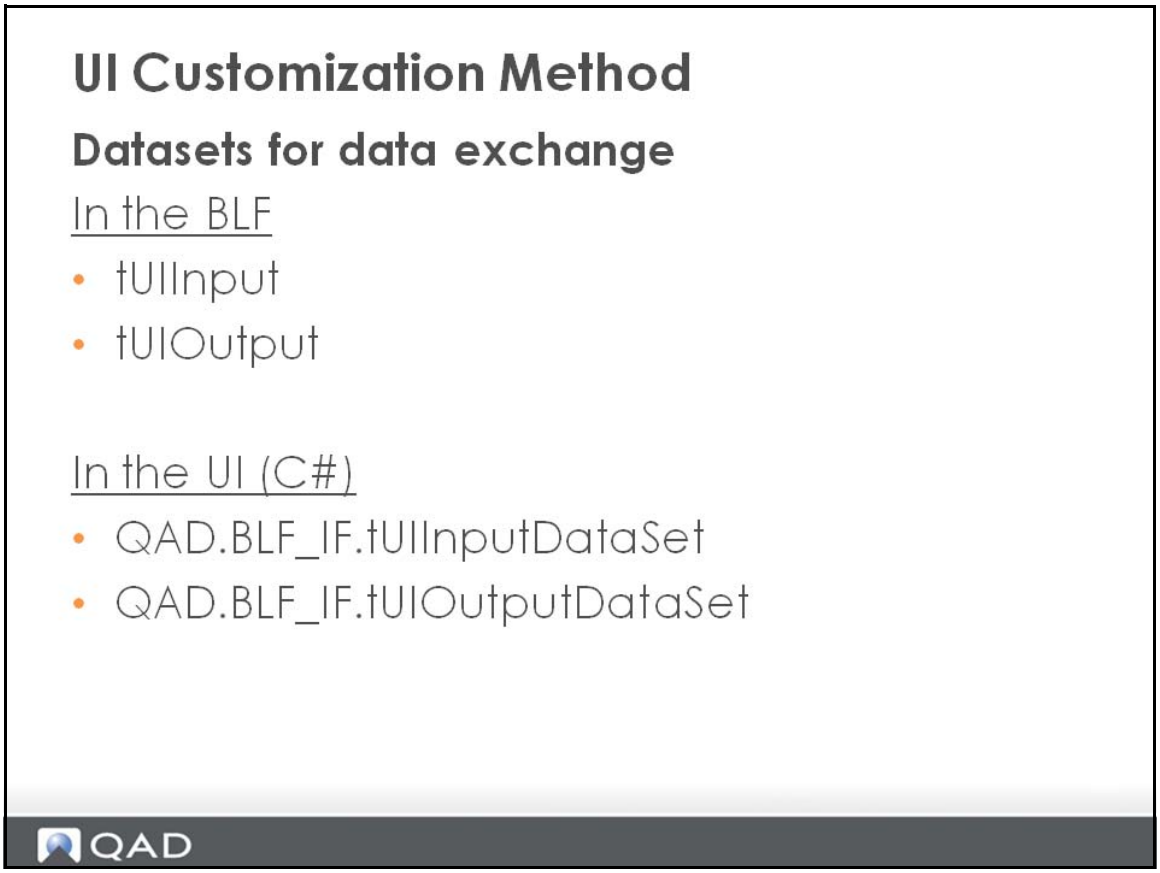
- If ilPartialUpdate is true:
  - Expects an input dataset that can be a subset of the object dataset what concerns the structure.
  - For modification of objects, the data passed to the method does not need to be complete. The bare minimum is the alternate key fields of the main table in the object.
  - 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

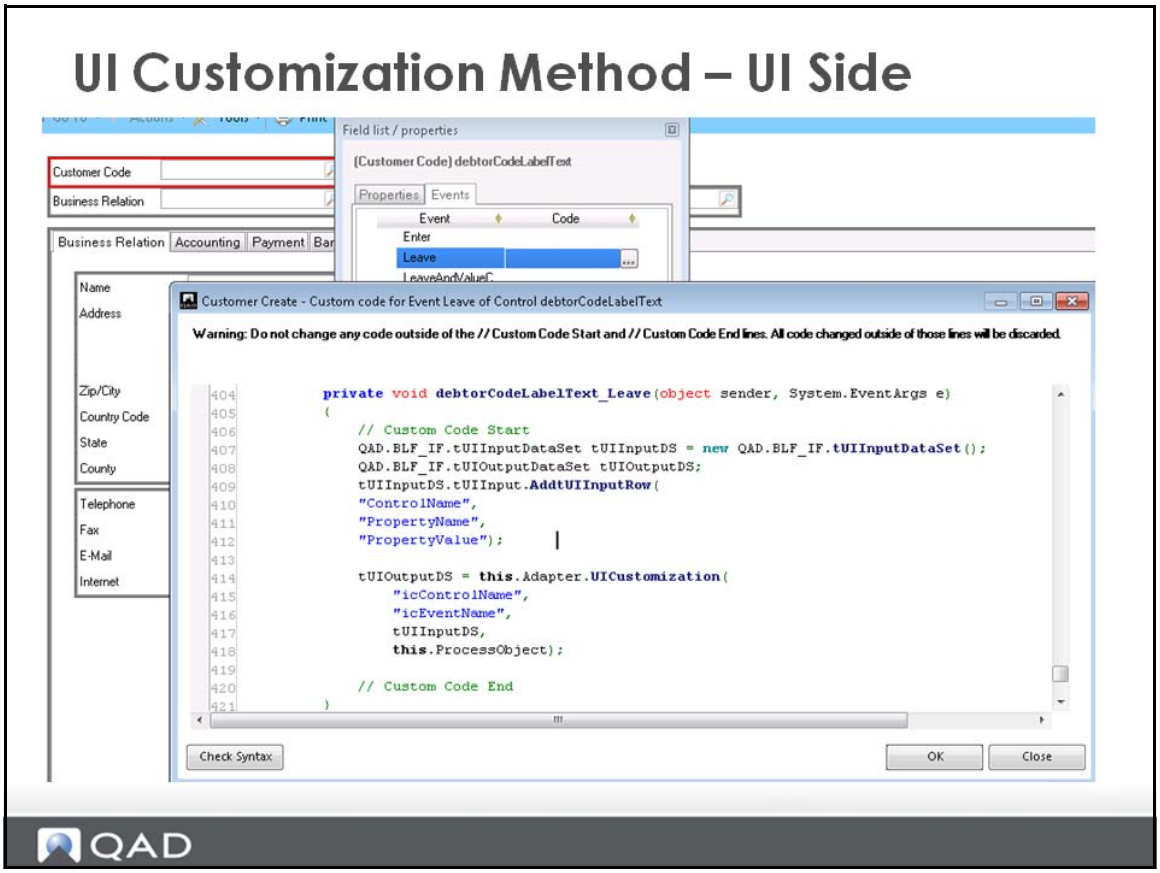


UI Customization Method



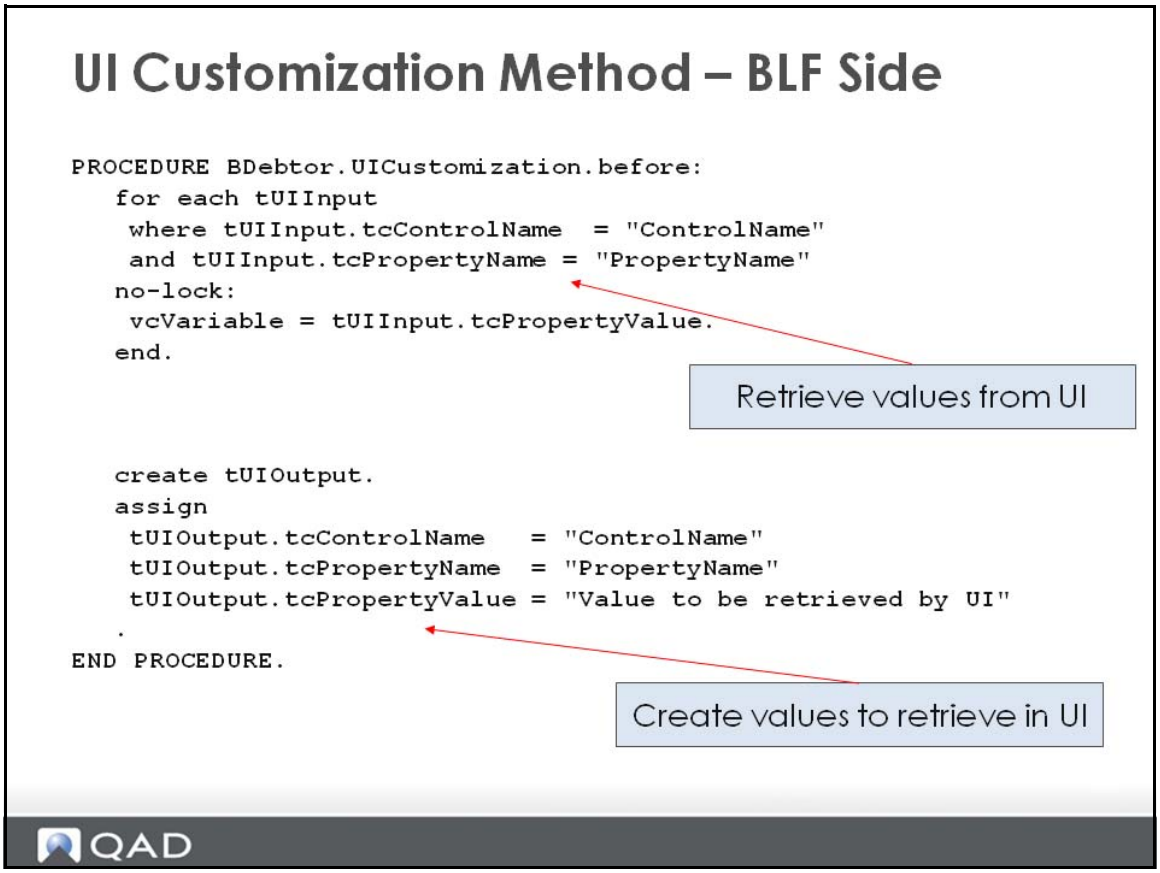
You can use the UI Customization method to transfer data between the user interface and business logic layer.

UI Customization - UI Side



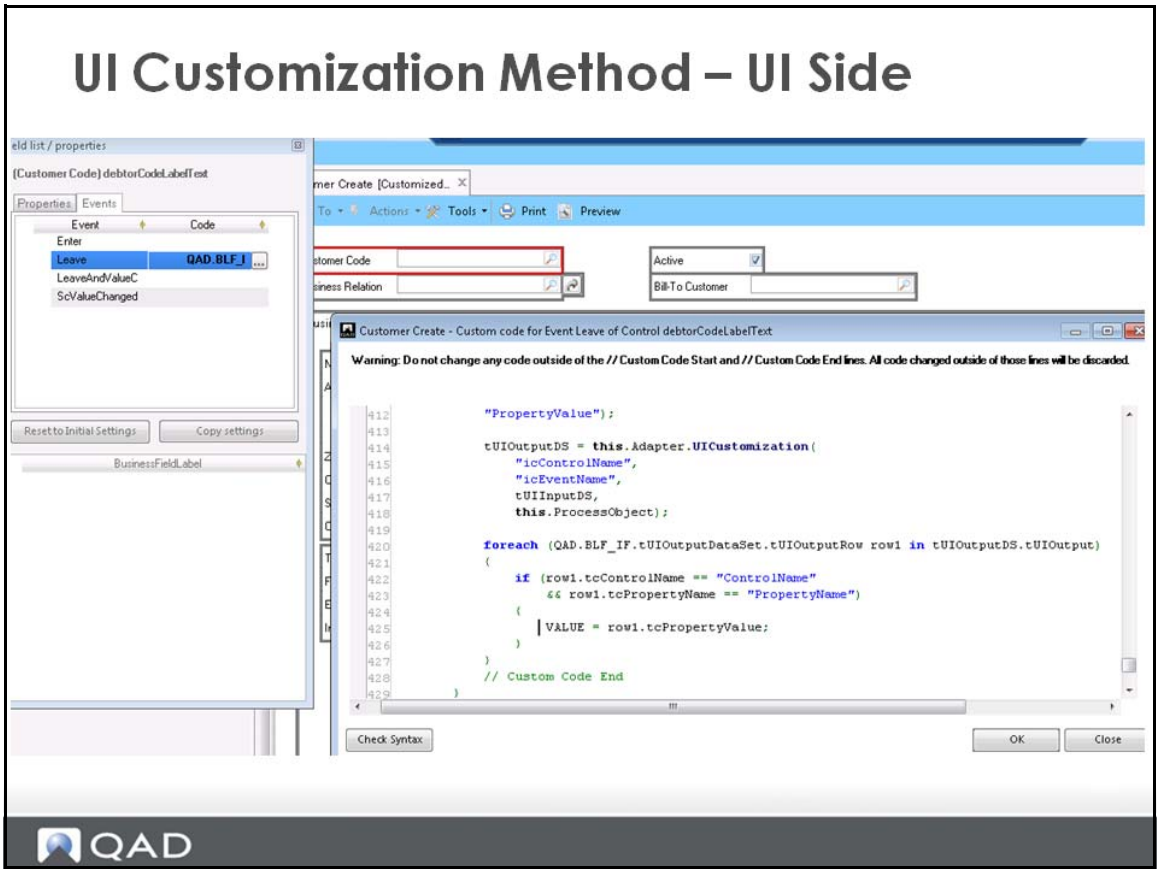
Create data in the UI and use it in the BLF.

UI Customization - BLF Side



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

UI Customization - UI Side



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

QAD Knowledgebase (knowledgebase.qad.com)*
Search for answers, tips, or solutions related to any QAD product or topic.

QAD Document Library (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)*
Visit QAD's one-stop destination for all courses and training materials.

*Log-in required