Single Window App – Modernize your application

By Thomas Maul, 4D Germany.

Technical Note 24-14

Table of Contents

Abstract

Based on an older 4D example, 4D Invoice, this technical note demonstrates ways to modernize an older application into a modern Single Window app. It covers a dynamic, user-customizable toolbar, a user-customizable list box, and finally, a display and user interaction approach similar to Microsoft Outlook. This allows users to work with data within the same window, avoiding the typical double-click to view detailed information.

Introduction

Classic applications often work by displaying a list of data in an output form, the end user double clicks to open a new window, running in a new process, to get details about a record.

Opening a new process for every new window increases the load in large networks, but especially has a huge impact on performance in slower networks, such as in home office.

To change the user experience to a single window mode, we show several steps, such as adding a kind of ribbon toolbar, customizable by the end user, a list box to replace the output form, and a preview area replacing the double click. Those are just examples for the process to modernize. You can use all or just part of this concepts to enhance your application.

To make it easier to follow we divide the content of this technical note in 3 parts:

- Toolbar
- List box
- Preview area

Minimum requirements

As the whole concept is based on classes, your application must be in project mode. The source and example code used in this tech note take advantages of new language features added with 4D 20 R2 – R5.

For the toolbar example: if you need to use an older 4D version, you can use an older version of the toolbar classes from this repository https://github.com/ThomasMaul/Toolbar. It requires 4D 18 R3 or newer. The main part of the functionality is the same.

About the example application

4D Invoice was published more than 20 years ago, and released in many different versions. As standard example, or to show new features, such as using object fields as custom fields. Latest public release was a total rewrite to show ORDA functionality. The example was also used on several 4D Tour events.

As this technical note shows how to modernize an older application with minimal rewrite, we chose an older version of 4D Invoice, based on 4D 16, as base for this example.

Find the latest version of the example here:
https://github.com/ThomasMaul/EA_Invoices/tree/SingleWindowMode

Part 1 - Toolbar – Ribbon Bar

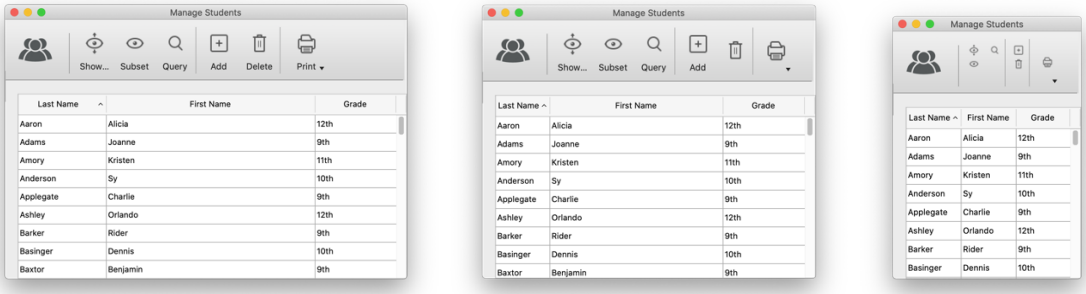
Especially on Windows SDI (Single Document Interface) Applications are not using menu bars but focus on Ribbon or Toolbars.

This part of the technical shows how to implement a dynamic toolbar, responding to window resizes, allowing the end user to position and chose functionality as needed.

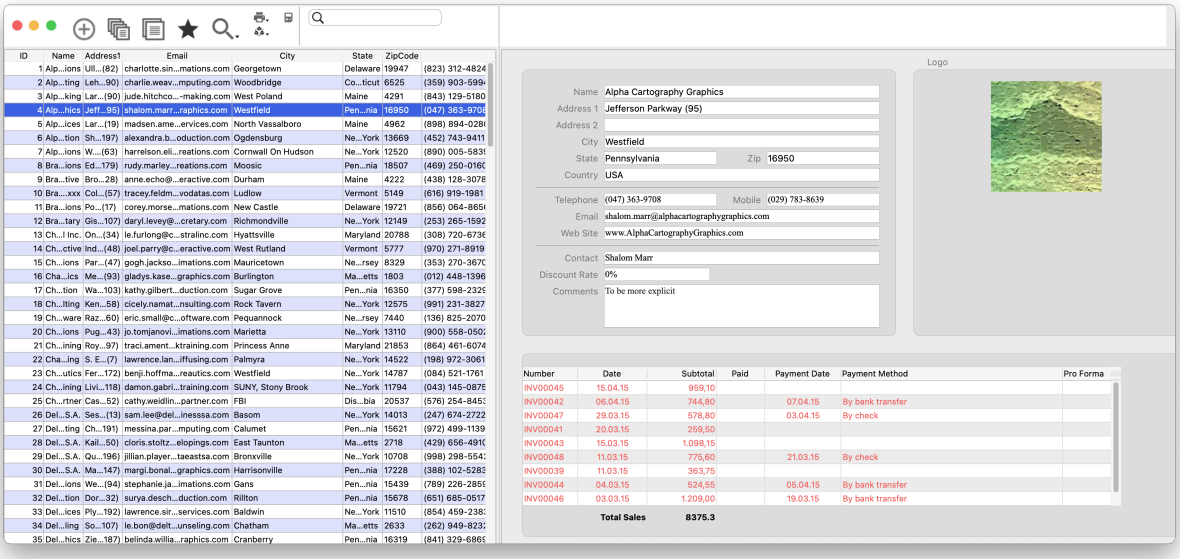
To add a responsive/resizable button toolbar to your application you just need to add two classes, handling all the work. The toolbar resizes depending on the available space, using large, full sized buttons with title if enough room, getting smaller by removed title (then displayed as tip) or even using smaller icons and stapling two buttons on top of each.

The class supports normal buttons or buttons with alternate action (arrow beside the button to open a context menu) and sub forms to display a search picker.

The same toolbar, responding to different window sizes:



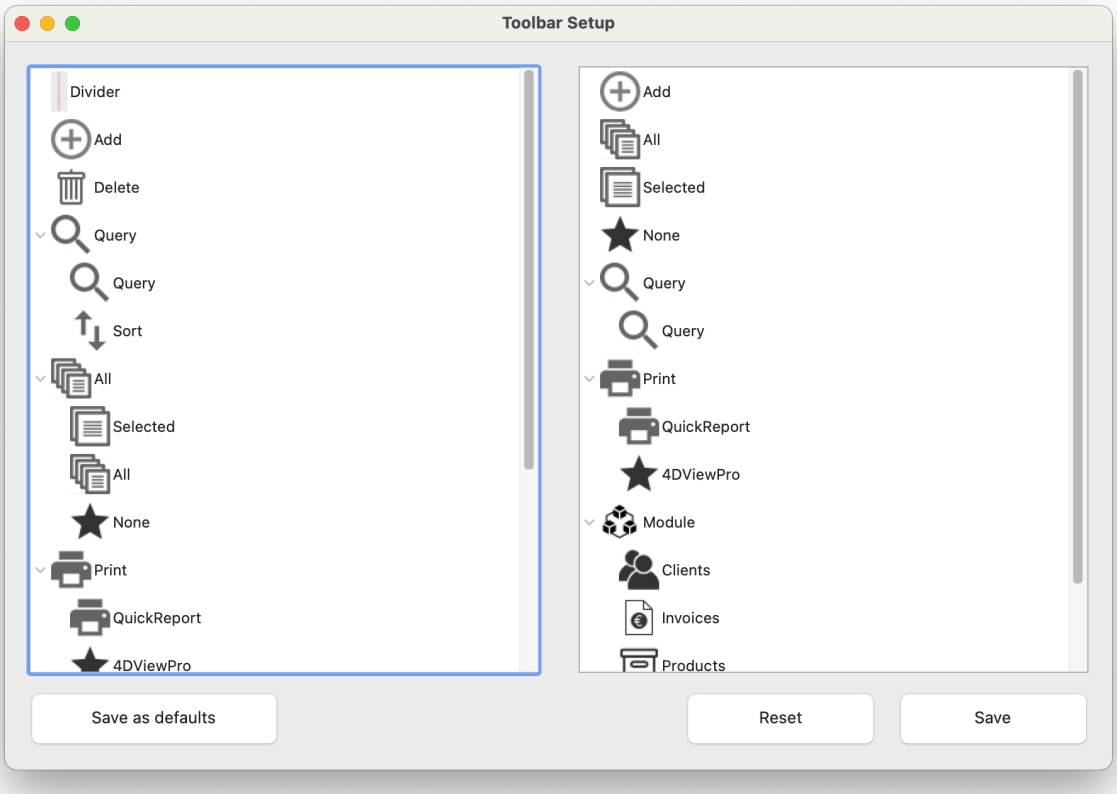
With the enclosed example application, 4D Invoice, move the splitter between left list display and the preview area on the right, to change the width of the toolbar:



Toolbar Setup – End user setting

The Settings dialog allows the end user to select which buttons to be displayed – and allowing to reposition them, or to move buttons from a subgroup to become a main button.

In the example database, click the button “Settings”, then “Toolbar Setup”, to get this dialog:



The left area includes all available buttons or features. The right side the current user selected. Try for yourself, on the right side, drag & drop buttons to another position, right click to delete totally. Any time, click “Reset” to go back to default set.

When you compare left/right, you will notice that the default set includes a group, top button “All”, including sub buttons (context menu) featuring All, Selected and None. The toolbar shows this as a button with an arrow, clicking the arrow opens a sub-menu with the 3 additional buttons. The end user might rearrange them, such as moving sub-buttons to top level, by example for often needed actions.

To reduce end user support calls, the code checks that a button cannot be moved into a wrong group or exists twice.

The Toolbar Setup Dialog is not mandatory for the toolbar functionality. It might help you as developer, your customer’s admin and the end user to customize, but if preferred or needed, all can be handled by code.

Toolbar Setup installation

To install in your application, copy the class “Toolbar_Setup” and the project form “Toolbar_Setup” in addition to the two main classes, into your application.

The form “Toolbar_Setup” uses some localized strings, which are stored in Resources/en.lproj/ToolbarEN.xlf”, copy this file/folder as well. A German translation is provided as well, to add other languages, duplicates the folder, rename names and content.

Toolbar Setup configuration storage

As a developer, you need to deploy two files, one containing a list of all available buttons (useable in Toolbar Setup, displayed on the left) and another to contain a default settings list.

The files are in Resources/Settings/Toolbar/Buttons.json and Default.json

When an end user modifies the list and click save, the settings are stored in Logs/Setup/Toolbar/User.json.

Note: while in Single User mode, the Logs folder is located beside the Data file, in client server mode it is located in the current user folder, location depending of the OS.

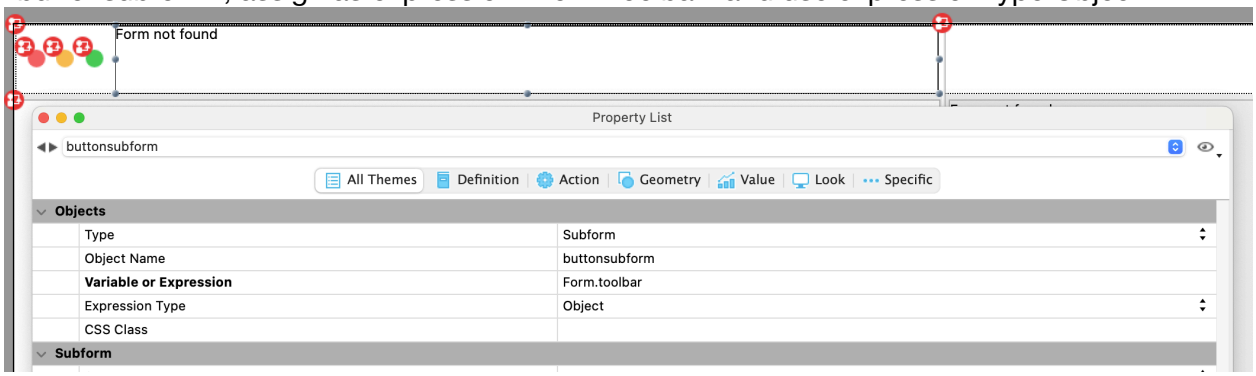
To help the developer to create the Default.json file, just use the dialog, arrange the buttons as wanted, click save and use/rename the User.json as Default.json, both have the same format.

Toolbar Init – Using Setup dialog

When you use the Setup dialog, initializing the toolbar is simple.

```
Form.toolbar:=cs.Toolbar.new()
Form.toolbar.load()
```

In your form, create a sub form, don’t assign an existing form, give it the name “buttonsubform”, assign as expression “Form.toolbar” and use expression type Object:



The names buttonsubform and Form.toolbar are hard coded. The callback method name is hard code as “ORDA_Listbox_Method”.

Toolbar Init – Using code

Doing all by code is more work – but gives more control.

Get a new instance of cs.Toolbar, as above. Then, for each button, get an instance of cs.Toolbar_Button and add it to the toolbar object using Form.toolbar.add($mybutton)

When you create a new instance of Toolbar_Button, you need to (or you can do) specify every single aspect, such as big icon, small icon, font size, style, title, width, tooltip, popup placement and callback method.

See chapter Toolbar Button – Properties for details.

Toolbar Callback – Respond to clicks or events

When creating a button using code, you assign a callback method, called whenever a button is clicked. When you use the Setup Dialog, the name is hard coded as “ORDA_Listbox_Method”.

This is the place to handle a button click event.

As the method is called without parameters (it acts as an object method), the first job is to learn about the button name, to be able to respond to which button is clicked.

Example:

```
var $buttonname : Text:=FORM Event.objectName
Alert($buttonname+" was clicked")
```

Note: FORM Event always return the property code, to learn about the event type, such as On Clicked, and in addition many more properties, depending on the context. objectName is a very useful information in many cases, but if you never looked deeper, especially for list box objects, it make sense to check the documentation (https://developer.4d.com/docs/FormObjects/listboxOverview/#supported-form-events-1)

The example application 4D Invoices uses the callback method ORDA_Listbox_Method for several events. Beside toolbar button events also for resize or double clicks in the list box. You might want to take a look inside, to get some ideas.

For button clicks, it forwards click handling to a function in the list box class:

```
var $buttonname : Text:=FORM Event.objectName
CALL FORM(Current form window; Formula(Form.ORDA_listbox.handleButtonClick($buttonname; $event)))
```

This class function has a case of checking for standard buttons, as All, None, Selected, Query, Sort and Print, and provide generic functionality for these actions. For any other name, which is more application specific, it calls back the ODA_Listbox_method (again), but passing a parameter “customButton” with the button name, so this method can run more customized code.

The generic functionality handles basic job, such as selecting all records and update the windows title (5 of 5 records).

But what if you need a different behavior for a specific table?

Take a look at the code for button = “all”. It calls This.useAll($classname).

```
Function useAll($class : 4D.DataClass)->$all : 4D.EntitySelection
If ($class.useAll#Null)
	$all:=$class.useAll()
Else
	$all:=$class.all()
End if
```

Looks maybe a little bit unusual, but the idea is simple. It checks if the given table/class has a class function named “useAll”. If yes, it calls that method, else it calls the standard .all() function.

So let’s assume we call useAll(ds.CLIENTS)

If there is an existing data class function named “useAll” in CLIENTS, it will execute this, else it just run a ds.CLIENTS.all()

So we have a generic all function, useful for all tables, while we can overwrite it for some tables by simply adding a specific all function in those tables.

And if you look in the CLIENTS class of the example application, you will see:

```
Function useAll()->$all : cs.CLIENTSSelection
$all:=ds.CLIENTS.all().orderBy("Name asc")
```

If the button is clicked while displaying CLIENTS, it runs and automatically sort the result. You could imagine for invoices to show only unpaid invoices, for orders only unshipped orders, etc.

This concept allows to have a general behavior and overwrite it – if needed – on a per table base. Instead of having a long case of, table specific code is directly in the table. This is more a question of how you want to work, having per function, such as all or query, a long case of handling all table exceptions, or having the exceptions directly in the table. We used this concept here to show a way to benefit from the new class functions.

Toolbar Search box

The Toolbar class allows to add a search box, displayed on the right side of the toolbar.

When using the Setup dialog in combination of the list box class, the search box is automatically displayed (or hidden), depending if the current displayed table has a class function named “quickSearch”.

Example for class CLIENTS:

```
Function quickSearch($value : Text)->$all : cs.CLIENTSSelection
If (Length($value)<3)
	$all:=This.useAll()
Else
	$all:=This.query("Name=:1 or Email=:2 or City=:1 order by Name asc"; $value+"@"; "@"+$value+"@")
End if
```

When you display clients in the window, a search box is displayed in the toolbar. Switching to projects, search box disappears (as there is no quickSearch function in class Projects).

Using the Toolbar through the setup dialog assign ORDA_Listbox_Method as callback. The method checks for clicks or On Data Change event:

```
: ($event=On Data Change)
    If (String(FORM Event.objectName)="search")
        CALL FORM(Current form window; Formula(Form.ORDA_listbox.handleSearchbox()))
    End if
```

handleSearchbox() is a function from class ORDA_Listbox and checks if the current table has a function named quickSearch, if yes, it executes it.

If you initialize your toolbar yourself, add a searchbox by code, when you add the buttons.

```
var $searchbox : cs.Toolbar_Button:=cs.Toolbar_Button.new(New object("name"; "search"; "group"; "300";
"mytype"; 1; "prio"; 1000))
$searchbox.width:=205
$searchbox.height:=36
$searchbox.method:="mycallback"
$searchbox.subform:="SearchPicker"
C_TEXT(vSearch)
$searchbox.dataSource:="vSearch"  // needs to be a process text variable, Form.xx will not work
$searchbox.dataSourceTypeHint:="text"
vSearch:=""  // default text empty
Form.toolbar.add($searchbox)
```

The example above uses the widget SearchPicker (https://doc.4d.com/4Dv20R6/4D/20-R6/Overview.300-7185320.en.html) in the toolbox, assigning “mycallback” as object method.

In this method, check for on data change and respond as required, by doing a search or whatever.

Toolbar Button - Properties

Create a new instance for each button you need in your toolbar. In the constructor new.() you can pass almost all properties of a button.

mytype
0 for button, 1 for subform/container. Not a standard 4D object property

title
text for button

name
object name, useable in event method (use OBJECT Get name(Object current))

prio

Priority, which button to reduce first. Smaller numbers reduced last, highest first Not a standard 4D object property

group (text)
When group changes, buttons are separated with a vertical line Not a standard 4D object property

icon
relative 4D path to 32x32 png of type 4 state button, such as /RESOURCES/Images/ButtonNew32.png

icon16
relative 4D path to 16x16 png of type 4 state button, such as /RESOURCES/Images/ButtonNew16.png Not a standard 4D object property

tooltip
text for tip. If not passed it uses button title

width
Width of button/subform. Default 70

height
for button or subform. For button by default 55. Automatically reduced to 30 for small buttons

popupPlacement
empty or "separated". Separated adds an arrow to support right click options. When used buttons usually needs more width, don't forget to add event "onAlternateClick"

events (collection)
events when to call the assigned method. You will need "onClick" in almost all cases, for right click also "onAlternateClick"

style
CSS style name

method
method to call for event. Pass the name of a project method

subform
only when type=1. Name of subform to display, such as "SearchPicker"

dataSource
only when type=1/subform. Assign a data source, for SearchPicker it must be a process text variable, Form.xxx will not work

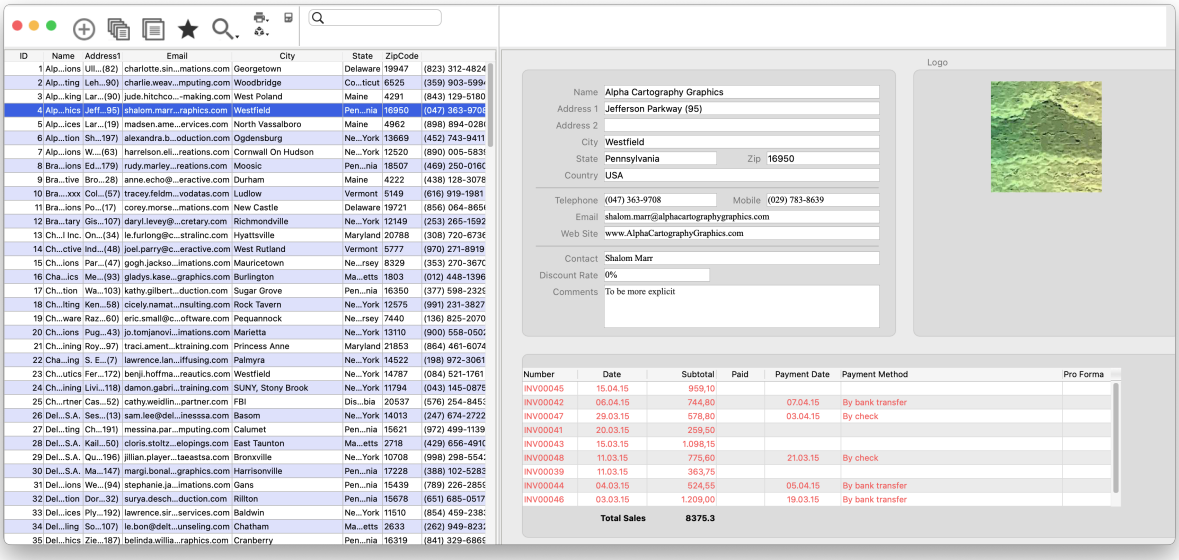
dataSourceTypeHint

type for dataSource. For SearchPicker pass "text"
You should not need to call any of these methods yourself for using the Toolbar. Only use them, if you need to enhance/rewrite functionality.

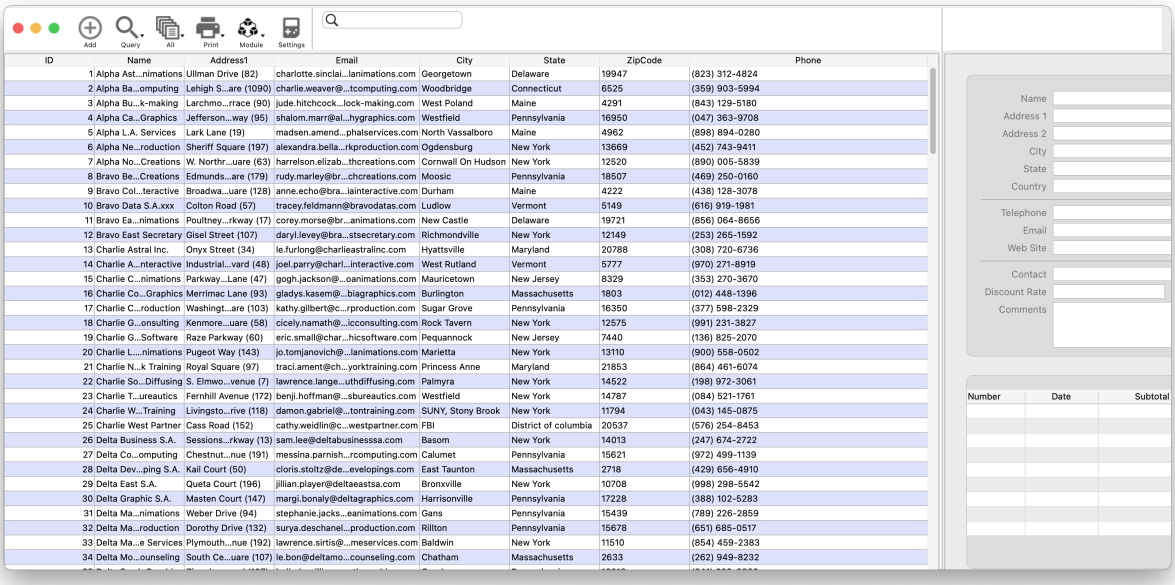
Part 2 – Customizable list box

To add a customizable list box to your application you just need to add one class, doing all the work, and an empty list box (no columns assigned) to your form. The displayed table and the displayed expressions for columns are dynamically assigned at runtime.

With the enclosed example application, 4D Invoice, move the splitter between left list display and the preview area on the right, to change the width of the list box:

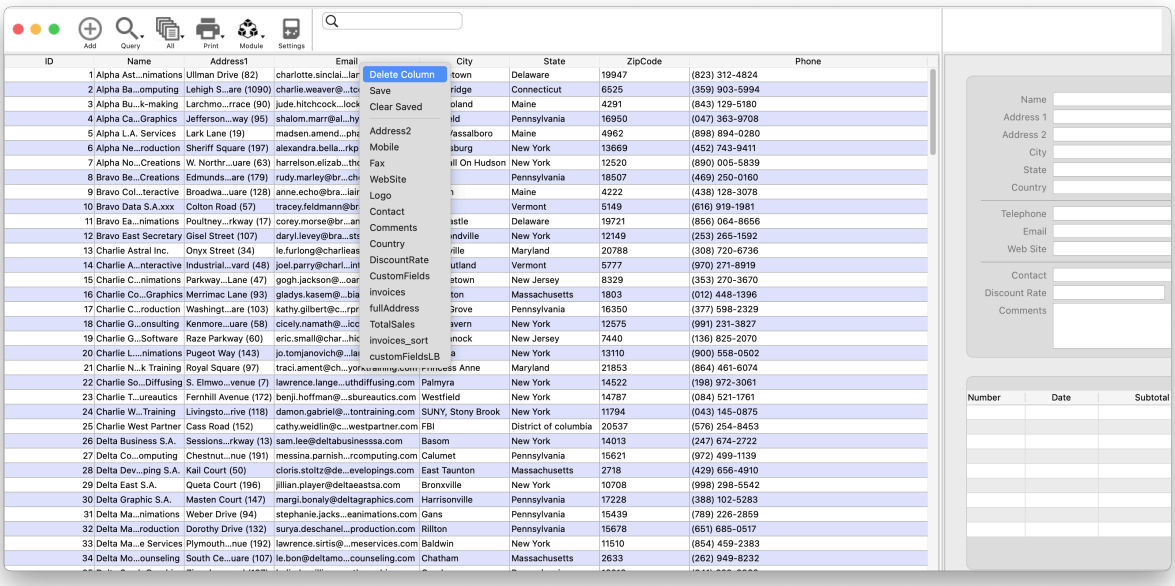


Resize:



As you can see above, each column is resized, not just the one far right.

The end user can right click a column, to add or remove columns:



This list also allows to save the current display, including added or removed columns, resized columns (giving more space and so priority to a column in relation to all other columns), this remembers also rearranged columns, the end user can drag&drop columns in another order.

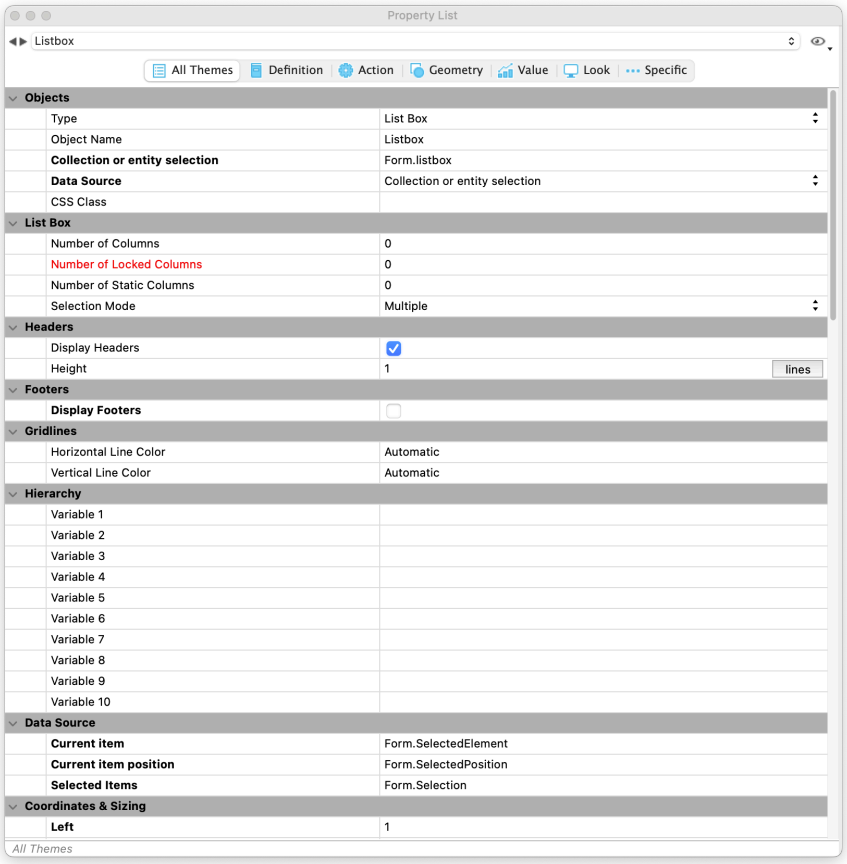
The displayed field list does not only cover “real” fields, it also includes aliases (which are typical fields from related tables) and computed attributes.

Installation

From the invoice example application, copy the class “ORDA_Listbox”. Also copy the project method “ORDA_Listbox_method”, which is used as callback jobs such as double click handling. You can use it as sample to extract code. If you also use the toolbar class, you might want to use most of it as starting base.

Then create a project form and a list box object.

For the list box object set these properties:



Assign Form.listbox as source and set Form.SelectedElement, Form.SelectedPosition and Form.Selection as Data Source. The 3 last one are only needed in combination with the preview area, which will be covered in the 3rd part of this series.

As Form Method add events for On Load:

```
Form.ORDA_listbox:=cs.ORDA_Listbox.new(ds.CLIENTS)  // adapt to your default table to start with
Form.ORDA_listbox.load()
Form.ORDA_listbox.setInputForm()
```

And an event for On Resize:

```
: (Form event code=On Resize)
```

```
Form.ORDA_listbox.resize()
```

You might want to look in the project form ORDAListbox and its form method from the example 4D Invoices as reference.

Finally, you need an object method for the list box, which handles header right click and – optional – double click of a line.

Open the object method from the example 4D Invoice (Form ORDAListbox, object Listbox). You can take it as it is, unchanged.

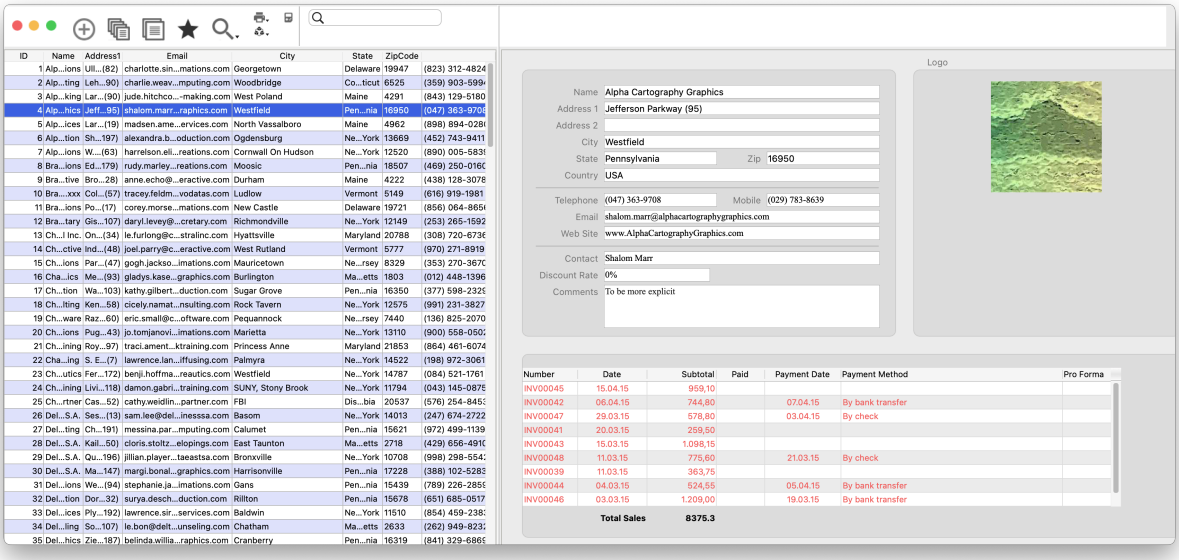
The first part handles header right click, which you properly will use as it is. The part for the event On Selection Change is only required if you use the preview area as well (3rd part of tech note series). The part for "On Double Clicked") can be either used or rewritten, to respond to a double click…

Part 2 - Preview Mode

Many business application shows a list of records and open on double click a new window for edit mode. In a single window application, a simple click on a record displays directly the details in a preview area. This way the end user can quickly browse through records in the list, and receive details with a click. If the user modified something, a click on another record opens an "do you want to save your changes" dialog. As the concept uses optimistic locking it is very efficient.

A double click on a record opens a new window, to allow multi window mode.

Most end users already know this working concept from Microsoft Outlook, making it easy to introduce in business organizations.



The concept is easy to implement in existing applications.

Using the classes from part one and two of the technical notes series it is easy to enhance 4D applications written many years ago, even if they still work in classic language mode. New code is required only for this single window. For the preview area, you need to create a form for each table using ORDA concepts, while keeping all other forms (and code) unchanged. To ease coding usually it makes sense to focus on simple, easy to recode parts, keeping more complex parts in the previous form. The end user can do the usual daily work in the preview area, needing to double click only for rarely used parts.

Installation

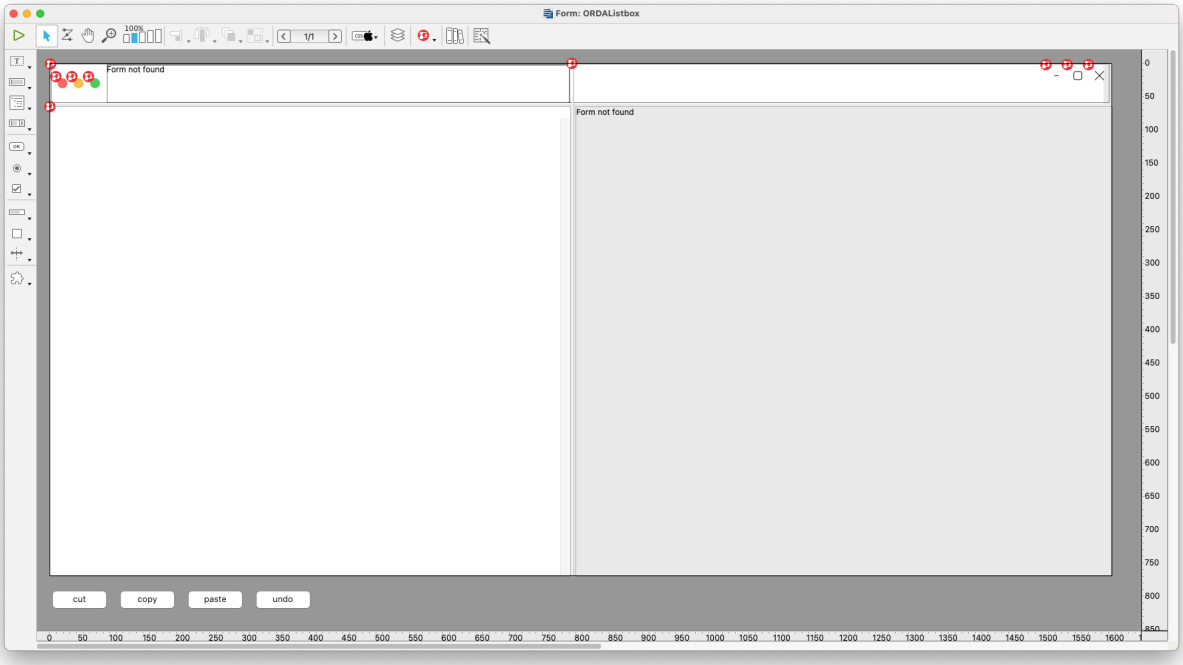
The concept is based on a window, showing a list box and a preview part. This can be done very generic, one list box for all tables and using one form per table, displaying it using a sub form in the preview area.

The list box class of part two of the technical note ease this work a lot, so we advise to use it. But you can also use your own list box code and just follow the concept. Using the button toolbar is optional, if your application already has another approach to avoid or minimize menu bar usage (for Windows SDI mode).

Form content

Open project form ORDAListbox from the example application to have a deeper view.

There are several parts to notice



At the top, very left and very right, are the standard Mac/Windows window title bar buttons, which we handle ourselves, as we hide the standard window title bar. Depending on the current platform, either the Mac or the Windows buttons are displayed.

On upper left is a sub form, which uses a dynamic created form through the toolbar class, to display and handle dynamically the toolbar.

The lower left is a generic list box, filled by code on load and updated when the displayed table changes.

The right part is another sub form, used to display a preview form. Whenever the selected table (also displayed in the list box) changes, another form is applied to the sub form.

Finally there are (hidden) 4 buttons at the bottom, to respond to standard system shortcuts, such as control-c, to respond to edit features, as there is no menu bar displayed.

All communications between these parts goes through the Form object.

As described in the 2nd part of the technical notes series, the list box requires some data sources to be set:

Data Source	
Current item	Form.SelectedElement
Current item position	Form.SelectedPosition
Selected Items	Form.Selection

Those are used in the list box code, to respond to on selection or double click events.

The sub form on the right requires to have an assigned expression of type object:

Objects	
Type	Subform
Object Name	preview
Variable or Expression	Form.preview
Expression Type	Object
CSS Class	

This allows us to pass the current selected entity.

We do not directly use Form.SelectedElement, so we can use additional properties inside the sub form container.

On Selection Change

The event On Selection Change is the main element. It is called when the form is opened (without an element selected), and every time the selection change, either because the user clicks a record, or other records are displayed

The trick here is that we need to care about the previously selected record, not only the current one. When the selection is changed, for whatever reason, we need to check with the previously selected record was modified, to ask the user if changes should be saved.

So when a selection change event happens, we check if there was a previously selected record. If yes, check if it was changed. If yes, ask the user if the changes should be saved. And if yes, try to save it, with auto merge option. If that fails, ask the user what to do, by example to wait. When wait, we need to select the previously selected position – which means we need to store these as well.

In any case we need to remember the current element and position and execute inside the sub form a method to initialize displayed data. In the invoice example, when displaying client data we hide or show custom data list box or for invoices set some fields to enterable or not. Classic applications often calculate data to display in variables, more modern ones are assigning those directly as expression to variables or use computed attributes.

As explained, the code displayed here is more an example or base for your own development.

```
: ($event.code=On Selection Change)
   If (Form.preview.data#Null)
      If (Form.preview.data.touched())
         C_COLLECTION($touchedAttributes)
         $touchedAttributes:=Form.preview.data.touchedAttributes()
         var $check : Boolean:=True

         If ($check)
            CONFIRM(Get localized string("SaveChanges"))
            If (OK=1)
               C_OBJECT($status)
               $status:=Form.preview.data.save(dk auto merge)
               Case of
                  : ($status.success)
                     // nothing all fine
                  : ($status.status=dk status automerge failed)
                     ALERT(Get localized string("SomebodyElseChanged"))
                  : ($status.status=dk status locked)
                        var $user : Text:=$status.lockInfo.user_name+"/"+
                           $status.lockInfo.host_name+"/"+$status.lockInfo.task_name
                        CONFIRM(Get localized string("RecordLockedFrom")+$user;
                           Get localized string("LockedWait");
                           Get localized string("LockedCancel"))
                        If (OK=1)
                           LISTBOX SELECT ROW(*; "Listbox"; Num(Form.preview.Position);
                              lk replace selection)
                           Form.SelectedElement:=Form.preview.data
                        Else
                        End if
                  End case
               End if
            End if
         End if
      End if
      Form.preview.data:=Form.SelectedElement
      Form.preview.Position:=Form.SelectedPosition
      EXECUTE METHOD IN SUBFORM("preview"; Formula(ORDA_Listbox_Method("preview")))
```

Note: it is important to execute the initialization code using EXECUTE METHOD IN SUBFORM, to run in the right form context.

Preview Form - Example

To create the preview form, we duplicated an old input form for a table, just renamed them from Input to “Input_ORDA”.

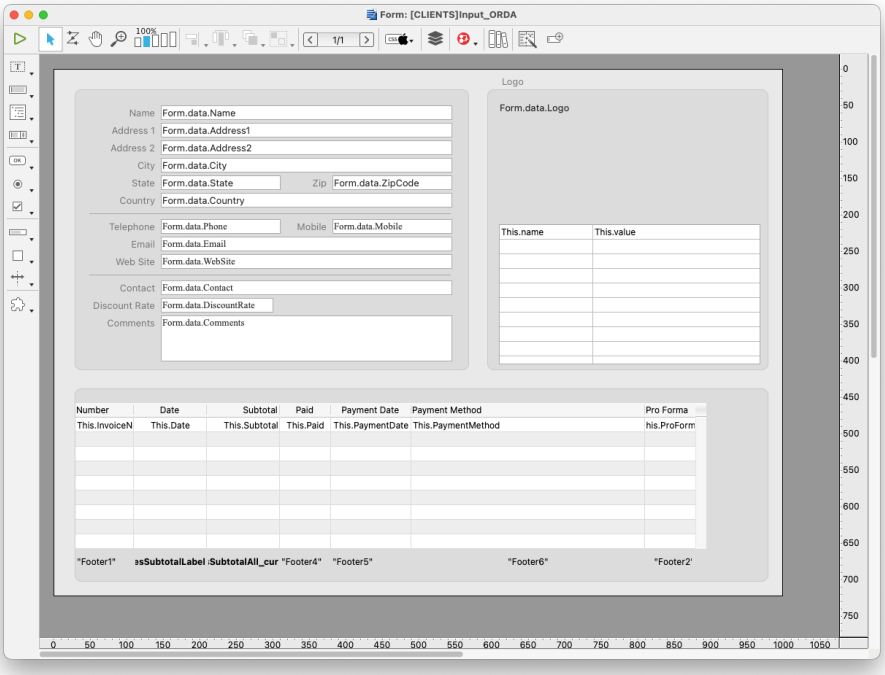
When you compare both, you will notice:

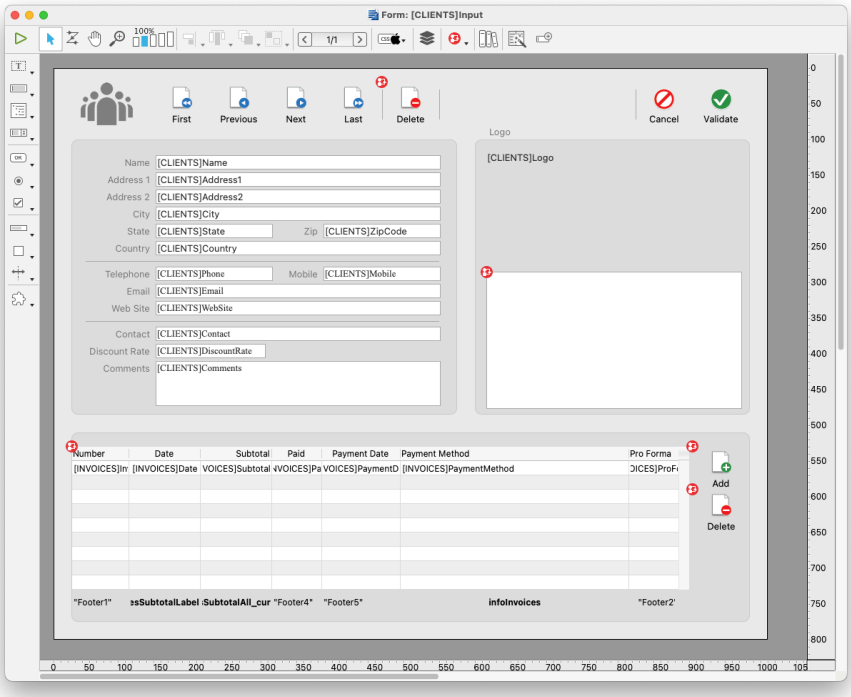
- Data source of each field is changed, like from [CLIENTS]Name to Form.data.Name
- Navigation and OK/Cancel buttons are removed (no need, to navigate or accept just use the list box)

- Buttons to Add/Delete an invoice are removed (example of rarely used, but more complicated features, so avoided to migrate).

How many features you need to migrate from classic to ORDA depends of the workflow of your end users. In our scenario, we expect them to work a lot with clients, need contact data, need details, check for existence, much more often than creating a new invoice through the customer form.

This is just an example, in this case it would not even be difficult to handle these two buttons, we just selected them as show case.





On Double Clicked

The event On Double Clicked in handled in the example application as event to open a new process to modify the record, using Classic Language. This is not to recommend classic, it is as example how to modernize your existing, classic application, with ORDA, without rewriting all. The main window handles usually most of end users work, while requiring a small amount of work to recode. For new development or refactoring of course we recommend to rewrite input forms to ORDA as well.

User benefit of rewrite

Rewriting an application is a lot of work, so the main question – why? What's the benefit?

The most obvious reason is to modernize the interface. SDI application, single window mode, Outlook experience – or simply creating an interface as customers are used from web applications, where double clicking a record is strange, while they are used to just click a record to modify it.

This leads to user experience. Often users are searching visually for a record. Using a search feature to reduce the amount of records, but then click through, as they are not sure about the correct user or invoice, so they look for details to identify them.

The classic mode, with a double click on each, check, close the window and double click the next one, is neither fast for the user – nor for the system (it produces high traffic and server load just to start/close new processes). Doing this job in a single window, just browsing through the list, speeds up the job drastically.

And so we come to a 3rd reason, reducing network traffic and server load. Of course this has high impact in larger companies with many users, reducing network traffic is a key feature in today's life, users working remotely, such as in home office. While internet connections become faster every year, latency times, the time a network packet needs to be send to the server and answered, is often one hundred times slower compared to direct Ethernet connections.

To compare the two ways to work, let's open once the old way, by executing method Clients_Manage, and record all network operations and once the new way, by executing Start_ORDAWindow.

To do so, we start 4D Server and connect 4D as remote running on the same computer. On Server, in maintenance window, click Start request log before executing the method.

Then we just need to sum bytes in/out and count number of packages to directly compare both ways.

Both display a list of clients, in classic mode double click 3 records (open, close, open, close and so on), in ORDA mode just click once to display in preview.

The logfile 4DRequestsLogServer_1.txt lists every transferred packet, with bytes in and out.

In classic mode this transfers 11 kbyte out, 14 kbytes in, using 124 packets.

In ORDA mode, it transfers 92 kbyte out, 6 kbyte in, using 50 packets.

This raises the question, why does ORDA transfer so much more bytes, shouldn't it be less? ORDA tries to group data, to send more in a single packet, to speed up slow networks. Even through the internet, bandwidth is usually high, allowing to send many data, while sending a single packet is slow.

So when the first set of records are requested, ORDA sends a larger chunk upfront, more than displayed and handles that as local cache. This drastically speed up network operations, especially with slower networks.

You can easily double check this. In the same log folder is a file named ORDARequests_1.json, listing every single ORDA packet, including the content. So you can easily see, which requests produce which answer.

Thanks to the grouping, there are less packets exchanged, and as more clicks/double clicks we would do, as bigger the difference, as better the result for ORDA.

The ORDARequests.json log file allows us to optimize the transfer even more.

Here an example:


```
{
    "url": "rest/CLIENTS/calcWindowTitle?$method='entityset'&$methodKind='dataclass'",
    "body": [{
        "__ENTITIES": true,
        "__DATASET": "109E4099080F438997081ED4B044F8B8"
    }],
    "systemUserName": "Thomas",
    "userName": "Designer",
    "machineName": "Thomas Air ARM",
    "taskID": 5,
    "taskName": "P_2",
    "startTime": "2024-09-30T14:04:30.518",
    "response": {
```

```
        "status": 200,
        "body": {
            "result": "200 Clients of 200"
        }
    },
    "ipAddress": "::ffff:192.168.0.80",
    "sequenceNumber": 2165,
    "duration": 151
}
```


We could question ourself if we need to contact the server to calculate this string. As we have all informations locally, we could run the code locally. When creating ORDA class functions, we simply need to set the keyword “local” before “Function” in the function definition – and we saved another packet exchange with the server.

This is just an example how logs can help to optimize the code to speedup usage in slow networks.

To learn more about ORDA logging and performance optimizations

https://blog.4d.com/logging-orda-calls-on-the-server/

https://blog.4d.com/orda-optimize-performance-with-full-control-over-rest-requests/

https://blog.4d.com/orda-rest-request-optimization-step-by-step-a-demo/

Conclusion

For a modern user interface a button toolbar is a key element. Adapting that to different screen widths automatically, to support different monitors or allow users to resize windows, improves user experience. Allowing the end user to adapt the button toolbar to their personal workflow increase the acceptance and user satisfaction. The here introduced classes drastically help to implement such a toolbar in your application, while keeping code control and not becoming depending of a complex framework.

While many applications are using a list box as central user element, most miss to offer to customize which fields to display, to change the order or to change the default width of a column. Using a generic class automate all this needs, making it very simply to offer this functionality as standard feature in an application.

Switching from the Output/Input window approach to a single window helps to speed up many end user work flows, it reduces server load and requires less network resources or exchanges, in total, a good choice for a modernized user interface.