Designing and Developing a XAML Navigation Shell for Line of Business Applications

A white paper by Billy Hollis, 26 Feb 2017

Modern native applications (for desktop or tablet) for line of business scenarios normally require an "application shell" or "navigation shell", which serves as the entry point to the application and the manager of the application's functionality. Some might refer to it as the "portal" for the application. This is a standard design pattern, but needs for a shell vary considerably among organizations. For example, some organizations need the ability to dynamically load new modules, while other organizations have a sufficiently simple and static application set that no dynamic loading is necessary.

Next Version Systems has developed or assisted in the development of over a dozen navigation shells for various clients in businesses as varied as healthcare, transportation, manufacturing, staffing, cattle management, and fuel management. Based on that experience, this white paper summarizes some of our lessons learned, including what capabilities a navigation shell needs for various circumstances. Hopefully, this paper can aid the design and development of a navigation shell for any team creating a XAML application above a minimal level of complexity.

Navigation shell capabilities

A navigation shell is designed to host a set of application modules, and it should accommodate new modules as the application evolves over time. It may also need to be designed to incorporate older technologies, such as Windows Forms or even older Win32 screens, side-by-side with newer functionality. This allows a gradual transition from older technology to new rather than requiring all screens and modules to be re-developed in XAML or other native technologies before the app can be deployed.

Navigation shells should also be designed to make it faster and easier for the average team member to create and integrate application views for business functionality. Navigation shells typical centralize functionality needed in multiple views, making it unnecessary to have duplicate and inconsistent ways to accomplish the same tasks. In essence, the navigation shell becomes a "sandbox" or framework that takes care of many routine application needs behind the scenes, allowing the developer of application screens to know less and get done more quickly, while the views they develop automatically include functionality such as saving favorites, or suspend-and-resume.

Application shells can be optimized for desktop usage, or for mobile usage, or designed to accommodate both. As a platform, a shell needs to be well architected, easy to use, and to support all the core capabilities that will be shared among modules or applications.

A navigation shell for desktop-based software usually needs the following basic functional areas:

- Begin by authenticating the user
- Check user's authorization permissions and configure available options for that user
- Display starting application screen, with menu options tailored to the user's permissions
- Accept user actions and display appropriate view
- Manage views using a URI-based identification mechanism (this provides for many extensible possibilities to be easily implemented, such as telemetry, favorites, and return to recent items)
- Support a centralized publish-subscribe capability for views and viewmodels to generate and receive events and notifications
- Provide for graceful shutdown by checking for unsaved data, and allowing the user to cancel the shutdown and return to the application to save the data
- Provide appropriate animations for changing of views
- Provide a standard wrapper control for views to centralize standard shell functionality, such as favorites and tear-off screens
- Provide a single standard location for developers to configure views handled by the shell and their associated navigation options
- Allow user to suspend a view and select another option, rather than having modal views that don't allow suspension
- Track open views and provide navigation back to open views
- Provide a means for views to indicate that they contain unsaved data
- Allow the user to set favorites to allow easy navigation to commonly used views
- Allow the user to access recently seen views
- Allow users to "tear off" views for placement on other monitors (similar to functionality in Visual Studio)

If older technologies must be incorporated alongside newly development modules, to allow for a more gradual transition to new technologies, then a desktop shell also needs:

- Host pages for old technology
- Extension of the URI-addressing to views based on older technology
- Plumbing to load older technology into "holes" in XAML views
- Allow the user to start up older, related applications, usually via command line activation

A shell for a mobile application would normally also have the following additional requirements:

- Touch-optimized controls for navigation, including swipes through lists and other standard touch gestures
- All screen elements for navigation, security, etc. appropriately sized for touch and generating appropriate touch feedback
- Robust handling of connectivity issues that can arise during mobile use

- Potentially, integration with various capabilities of the mobile device, such as camera, voice, and GPS location.
- If work is to be done in areas with low or no connectivity, local data storage sufficient to provide needed functionality when no connectivity is present, plus synchronization of data when connectivity becomes available.

More discussion of important functional areas

In this section, some of the basic functional areas above will be discussed in detail. For many organizations making the move to XAML, these details will help you start the decision-making process to narrow down the specifications for your shell.

Authentication and Authorization

Typically, an application shell begins by doing authentication of the user. This can be done in several ways. The most common and simplest option is to use the Windows login, allowing single sign-on for the user. Other options include such credential stores as Active Directory or custom database credential storage.

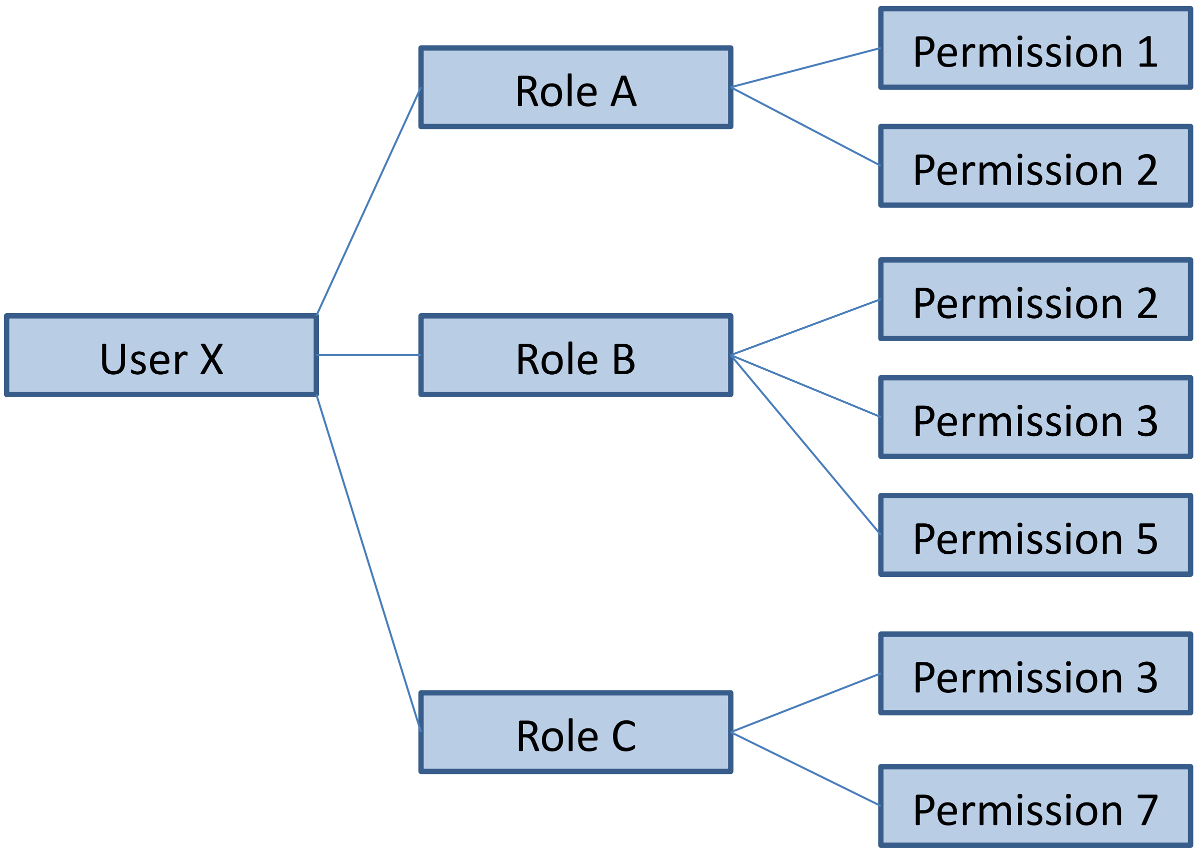
Some generic shell implementations, such as the one created by Next Version Systems, includes a stub in the shell for both authentication and authorization. This can then be customized as necessary for a particular implementation.

Authorization is a somewhat more complex requirement than authentication. The main requirement that needs to be pinned down is whether the applications in an organization need a fine-grained permission scheme.

The Windows roles structure for authentication and the Active Directory structure for authentication are both coarse-grained, and many packaged authentication designs have similar capabilities. The advantage of using them is that the application shell does not need to include any provisions for managing roles and permissions; the capabilities built-in to Windows or Active Directory can be used.

However, in our experience, many line-of-business applications (1) need more fine-grained permissions than Windows provides, and (2) prefer to do their own administration screens for authorization and authentication management.

If the application does need to handle its own fine-grained security, including administration, then a security framework would be needed as part of the shell. Such a framework has users, roles, and claims or permissions, which are related in the following way:



The permissions are then referenced in software to determine whether a particular capability is available. This might be an entire module, a screen or view, or an individual field on a view. If a user has a certain permission through any of that user's roles, then the user is granted access to a capability that requires that permission.

Most of our clients doing commercial software take the fine-grained option for the flexibility it gives in security, especially for applications that must span multiple platforms. A team should determine if that level of security flexibility is needed for its application shell. If it is needed, and the team does not already have a sufficient security framework for their next generation of applications, then the shell development is the logical place to design and develop security framework capabilities.

If so, then a finer grained security capability requires development of several pieces:

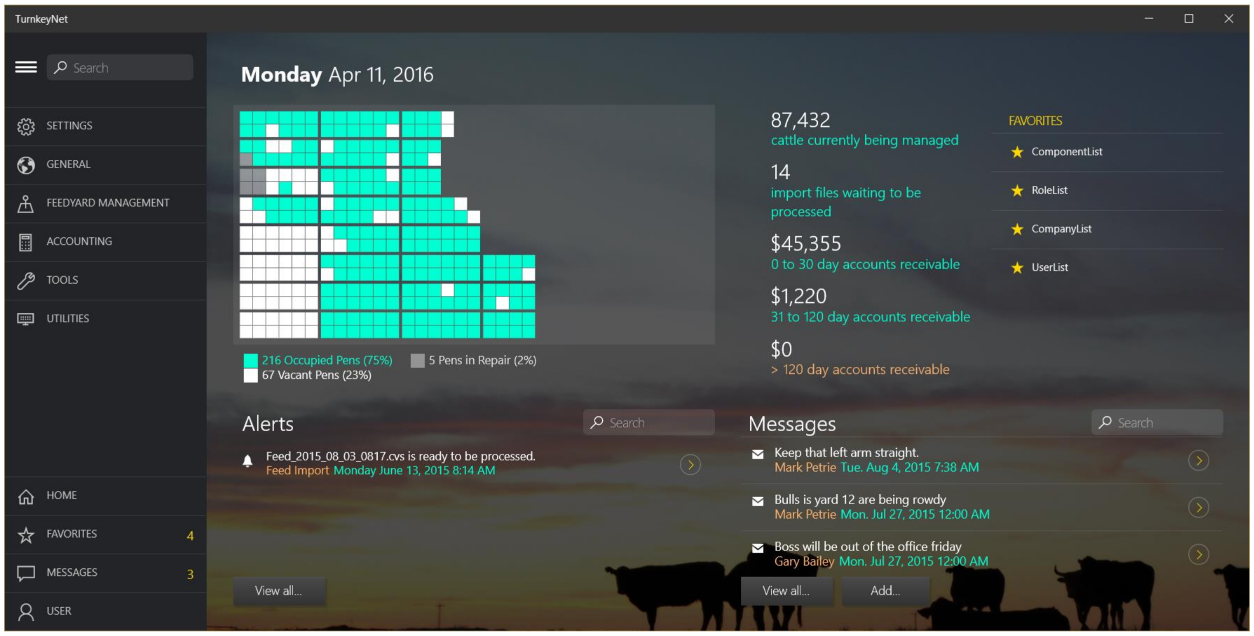
- A security database that uses the authentication keys to look up roles and permissions
- A small framework to allow checking for permissions

- Services that access the above functionality (note that these are typically separate from the services used for routine data access – often the routine data access services require the credentials from authentication and authorization)
- Administration screens to allow users, roles, and permissions to be maintained

Configuring the shell for the user

Based on the user's permissions, the shell checks after authentication to see what options are available to that particular user. Any options for which the user does not have authorization are typically not included in the shell's list of user choices – they are simply not loaded into any menu areas. Optionally, those options can be shown but disabled.

The shell then displays the first screen the user sees, based on available options. This is often some sort of home screen or dashboard. Here is an example of a typical shell with professional grade visuals:



Monday Apr 11, 2016

🟩 216 Occupied Pens (75%)
▨ 5 Pens in Repair (2%)
□ 67 Vacant Pens (23%)

87,432
cattle currently being managed

14
import files waiting to be processed

$45,355
0 to 30 day accounts receivable

$1,220
31 to 120 day accounts receivable

$0
> 120 day accounts receivable

Alerts

Feed_2015_08_03_0817.cvs is ready to be processed.
Feed Import Monday June 13, 2015 8:14 AM

Messages

Keep that left arm straight.
Mark Petrie Tue. Aug 4, 2015 7:38 AM

Bulls is yard 12 are being rowdy
Mark Petrie Mon. Jul 27, 2015 12:00 AM

Boss will be out of the office friday
Gary Bailey Mon. Jul 27, 2015 12:00 AM

User navigation

The initial screen contains actions for the user to select, such as looking up an individual record. Each such action has an associated view. Succeeding views may then provide navigation options to other views.

We normally design a shell so that all views are displayed by sending a command to the navigation shell. In our design, this is made simple by using a URI as an identifier for navigation.

The exact design of the URI scheme is designed for a particular set of applications, but here are typical URIs that might be associated with views:

http://yourdomain.com/ProductRecord

This could bring up a blank view to add a product record.

http://yourdomain.com/ProductRecord?ID=ABCD345

This could bring up a view for getting a product record with ID ABCD345.

http://yourdomain.com/ProductRecord?ID=ABCD345&Facility=04

This could bring up a view for getting a product record with ID ABCD345 at facility 04

http://yourdomain.com/Search?SearchType=ServiceOrder

This would bring up a search screen for service orders.

http://yourdomain.com/Search?SearchType=ServiceOrder&SearchString='lumber'

This would bring up a search screen for incidents and enter a default search string of "lumber".

Whenever a user takes an action that requires navigation to a view, the source that is handling the action informs the shell that navigation is desired, and passes on the URI needed to find the view.

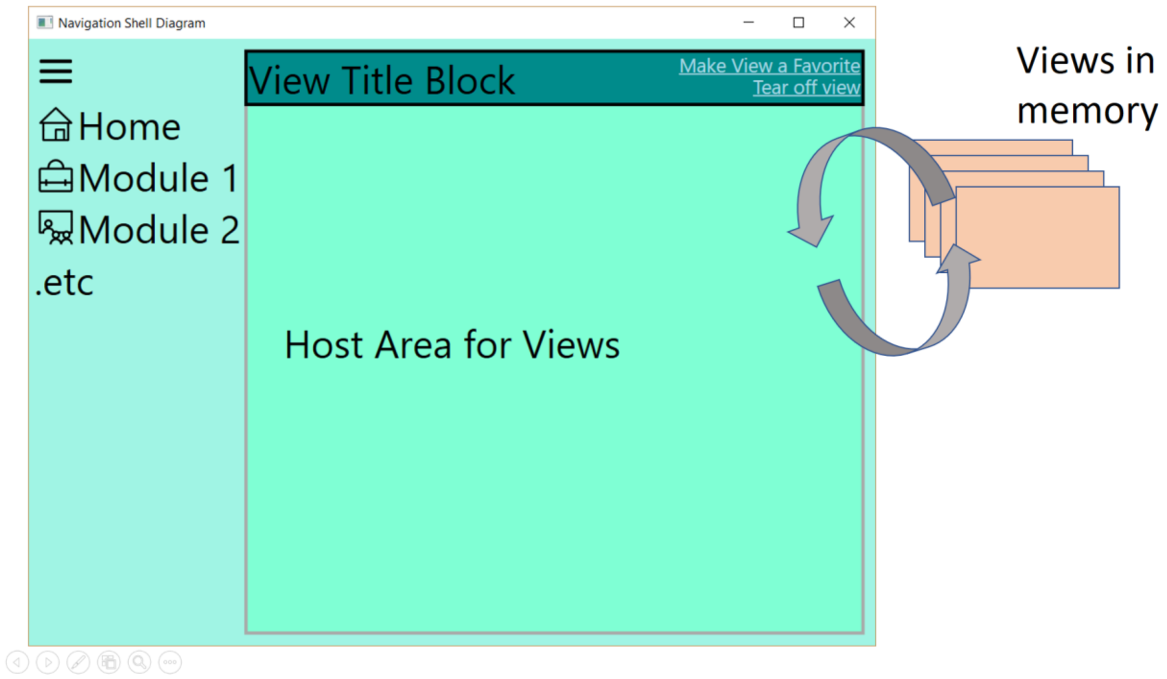
The URI design simplifies shell construction and view navigation, because it doesn't matter how a view is requested – the URI identifies the view and the shell navigates to it the same way, no matter what the source of the action might be. Also, URI provides for many extensible possibilities to be easily implemented, such as telemetry, favorites, and return to recent items. As a bonus, a URI navigation scheme makes it easy to extend the shell to show web content in views.

Tracking views

The shell should allow navigation to a view, even when an existing view is in an unsaved state. The shell then tracks open views and provide navigation back to open views. This lets the user suspend work on a view, and select another option to work on a different view.

In our experience, this ability to suspend-and-resume work on a record is one of the capabilities most appreciated by users who use multiple modules in an application. This capability is more helpful in desktop apps than in tablet apps, but it's still helpful to allow a user to suspend what he or she is working on, go do something else, and then come back and pick up where the user left off.

Normally, the shell maintains a dictionary of loaded views in memory, with the URI as the key to the dictionary. This allows the shell to quickly check if a view is already loaded when it is requested.



Support communication among views and modules

The shell should support a centralized publish-subscribe capability for views and viewmodels to generate and receive events and notifications. This will allow views to, for example, indicate that a list has changed, and any other views that use the list can monitor for that signal and refresh the list.

Unsaved data

Since views may not be displayed yet still be in memory with unsaved data, the shell should provide an interface for views or viewmodels to indicate that data is unsaved. The shell would display a standard user interface signal that unsaved data is present in the view when the view is on the screen. (We often use a gold asterisk, as seen in the famous StaffLynx demo, which you can see on YouTube by entering StaffLynx as the search term.)

If the user attempts to shut down the application while views are open with unsaved data, the shell will detect that condition, and provide for graceful shutdown by allowing the user to cancel the shutdown and return to the application to save the data.[1]

Visual capabilities for working with views

The shell should provide appropriate animations for changing of views, to give the user an intuitive sense of when views are opened, closed, suspended, etc.

[1] This mechanism requires views or viewmodels to properly implement unsaved data checking. Our generic data framework includes such a capability, but many commonly available frameworks do not, or include a very limited version of this functionality.

The shell should also provide a standard wrapper control for views. This allows standard styles to be applied to views, and for views to have some standard capabilities, such as a signal for dirty data, without that functionality being present in the view. The wrapper can be a simple, lightweight container in the initial version of the shell, and may have only minimal visual presence. Alternatively, it can be the location for user interface elements to make a view a favorite or tear off a view, as shown in the shell diagram above.

Creating and configuring views

The shell would typically have a single standard location where view types are discovered or “registered”. This part of the shell will determine which types of views are available to the user. For each type of view that is available, any settings needed to create that type of view and fetch associated data would be captured. Also captured at that point would be any information needed to display actions related to the view, such as the description in a menu or button to show the view.

Determination of available views can be done in several different ways. Available views can be loaded from a database table, from an XML manifest, or discovered dynamically with various techniques such as MEF (the Managed Extensibility Framework). We have used all of these options, because the right one varies among organizations. One feature they all have in common is that any assemblies that will be loaded dynamically to present views should be signed to prevent malicious replacement of them.

However, if the applications included within the shell are sufficiently static, and the number of views is not large, we recommend registering views statically in application code. The shell would then have a standard location for such code, and a few lines would be needed for each available view. This approach has the advantage of being simple and easy to learn for team members, and preventing any possibility of injection of malicious code.

The last missing piece is determining how views and associated viewmodels are instanced, and how data for a viewmodel is fetched through the service layer. There are several possible designs for this step, and those options are beyond the scope of this paper. When we do a shell, this is one of the areas in which we have extensive discussion with the team so that we can design an optimal approach.

MVVM support

For complex XAML applications, especially those such as mobile applications that must work with different form factors and orientations, clean separation of the layout XAML from data-related logic is helpful. While we are not an advocate of totally codeless views[2], we do advocate keeping code in views to a minimum to avoid duplication when multiple views work with the same data. For example, you might need separate portrait view and landscape view working against the same data in a tablet application.

[2] Our philosophy is that code in views is sometimes necessary or helpful to promote certain types of user experience. We think such code should be kept to a minimum, but we don’t think that “code-behind is evil”, which we regard as a hyperbolic, emotional statement instead of good guidance.

The most common way to separate layout from data-related logic is the Model-View-Viewmodel pattern, usually abbreviated MVVM. Most shells, including the ones we write, support this pattern, and some even require it. There are many web-based resources for learning more about MVVM if you are interested.

“Tear-off” views

For desktop use, the shell normally allows views to be placed in another, simpler window for display on another monitor. This capability is called tear-off.

Our standard design for tear-off is pretty simple – just press a button in the shell to get the view in a separate window, and then drag that window to an appropriate position. The user then closes that window when they are done using the view. More sophisticated tear off with dragging of views inside and back into the shell are certainly possible, but we find that the extra development cost isn’t usually justified because almost no one uses it.

Generic starting points for a shell

There are several potential strategies for creation of a shell, and the first task in starting work on the shell is to decide on the strategy needed. One strategy is, of course, for a client team write their own shell from scratch, but this is not usually a good idea for a team with little or no expertise in modern app technologies and layered client architectures. In such a case, it’s better to either have the shell developed for you, or use a starting point that already has a lot of the plumbing functionality in place.

Most of our clients prefer to have us write the shell, using our off-the-shelf starting point, which has been refined over a dozen or so projects for clients. Our shell includes both the plumbing described above and the UI construction to create a clean, compelling modern application.

However, there are other “starter kit” options available. For a team just starting serious XAML work, this certainly beats designing and developing a shell from scratch, both in time and quality. The three most commonly used are:

- Prism – produced by Microsoft’s Patterns and Practices Group (versions available for both WPF and UWP)
- MVVM Light
- Caliburn Micro

None of these shells have what we regard as a production quality visual design. It is necessary to roll your own, including how menus are positioned and managed, any animations for transition, visual theming, and control templates that provide functionality to coordinate with the shell.

At the risk of sounding biased, we think all of these options leave too much development for your team. They’re all viable, but they all have drawbacks that we think must be overcome. We also think they require too much learning on the part of individual team members that need to plug their views into the shell.

When we create a complete shell for a client, we vastly prefer designs that are “sandbox” based. That is, the shell is designed and implemented in a way that allows routine application developers to create and integrate modules and views while knowing very little about how the shell works underneath, and using “black-box” components for routine functionality such as data validation and notifications. Team members producing routine application views are given a “recipe” that goes through a defined set of steps to integrate their work into the shell, and they are given some training in any frameworks included in the shell, such as data validation. Otherwise, they don’t need to know much about the shell.

None of the shell starter kits are oriented around this approach – all require quite a bit of understanding from every developer using them. Plus none of these shell starter kits have some of the most important shell capabilities. They don’t include the visual design for the shell, tear-off views, favorites, recently-used items, etc. Only Prism has a URI-based navigation scheme built in. Therefore, even if one of these is chosen as the starting point, significant work remains to be done to make the shell useful for a complex, real-world business application.

Accordingly, most of our clients commission a custom shell using our own internally-developed technologies. Our shells tend to be dramatically simpler and with less code than the options above, and thus much easier for developers to understand. The tradeoff is that (1) we only put the capabilities in the shell needed by that client, and (2) there is no third party support for changing or extending the shell in the future.

Other capabilities often included in the client-layer framework

I’ve briefly mentioned some of the ancillary capabilities that are needed for effecting XAML client development. I mentioned dirty checking in your data framework, for example.

Other needed or helpful capabilities may include:

A dashboard

Most line-of-business applications need a dashboard of some kind. Our philosophy of dashboard design goes beyond just showing data to integrating work flows and helping the user realize what work they need to perform.

A good XAML dashboard is flexible enough to contain just about anything you can write in XAML, but should still integrate well with your navigation shell. If you would like demonstrations of dashboards to help you realize what’s possible, we’d be happy to do one for you.

A data validation framework

Data validation needs vary greatly from project to project. Some data frameworks have data validation built into code, usually exposed with the IDataErrorInfo interface. This is sufficient when the rules for validating data are static and don’t vary much between partitions of data.

Much of our work is in healthcare, which has quite demanding and dynamic needs for data validation. Our framework allows static validation rules embedded into the UI, or dynamic validation rules loaded from various sources, or a combination of both.

No matter how you do data validation, you'll need a way to expose data errors to the user, and a general way designed into the shell is a good idea. This often involves custom control templates that expose data via visual states, and sometimes a standard way to pass validation errors to the shell for unified presentation.

No matter how you do data validation, you should think about how the validation framework will be integrated into your shell.

A notes/notification framework and UI

Many line of business applications involve work done by a large group of users. It is often helpful for communications to these users to be exposed through the shell. While we don't advocate writing full messaging or email functionality, attaching notes to records can be a very useful capability.

About one third of our clients include a notification framework, with associated views and other user interface needs such as a visual indicator when new notifications arrive. In many cases, notifications from an external source, such as a manager or administrator, are included (e.g. "Production Line Z is down for maintenance"). Also, the application may have internal logic to generate appropriate notifications (e.g. "Customer X is getting low on fuel.")

Custom controls specific to the business

Advanced user experiences often require custom ways to enter and visualize data. For example, one of our clients was developing a fuel management package, and they needed a control to visualize a tank with its current fuel level throughout the application.[3] Another client needed a quick entry solution to the "they usually only pick one, but occasionally pick several" data entry situation.

In cases like these, you can gain faster, more consistent, and more professional-looking applications by creating custom business controls. However, if your team is comparatively new at XAML, they probably won't have the depth needed. You should get help for your early efforts, and in many cases the controls produced in your first round can be used as examples for the team to follow for their later efforts.

Integration with external products such as Outlook

Some businesses receive so much email from customers that assisted generation of replies by automating Outlook is a major productivity enhancement. Also, client contacts of various kinds may need information in Outlook.

Some applications need automatic generation of documents, and integration with Word or a PDF library may be a valued part of the shell framework. Since that kind of functionality is typically used throughout

[3] You can read more about the XAML that was created to do this visualization in the article "The Resurgence of XAML" at http://bit.ly/CodeMagBSH.

an application, instead of just by a particular view or even a particular module, making it part of shell development makes sense.

Shells for WPF vs UWP

The overall design of a navigation shell is quite similar in XAML for Windows Presentation Framework (WPF) and XAML for the Universal Windows Platform on Windows 10 (UWP). However, there are some significant differences to take into account.

If you are using UWP, and you wish to take full advantage of the mobile-optimized page navigation framework it includes, then there are significant differences in the navigation pipeline. This navigation framework in UWP is intended to provide memory and power savings. It isn't required – you can use a similar, memory-based navigation scheme as is normally used for WPF shells. Since it imposes additional learning and development time on your team, you should make a conscious decision about using it.

UWP also lacks the mature control set of WPF, so the client framework included with the shell may need additional controls. Some of the controls in UWP are very touch centric, and not a good match for rapid, keyboard-based data entry, and they may need to be replaced for desktop applications.

Overall, we estimate that a UWP development effort takes 20-25% longer than the equivalent effort in WPF. Various useful functionality that is built-in for WPF must be custom built for UWP.

However, UWP offers the advantage of longer shelf life because of Microsoft's investment in it.[4] Also, it is far superior to WPF for touch-based development. Many organizations begin their Windows 10 development in mobile applications for this reason. However, if you go beyond basic apps, a navigation shell is still a good architecture for a line of business application on UWP.

License and intellectual property notes

Many public navigation shell starter kits have open source licenses. The version of any shell we produce is not open source, but has unlimited rights for the client to use it in any way desired. ***Iit is not the case that the entire body of code is a work made for hire.*** Some parts of the code would be from our standard libraries of routines, and other parts will come from public sources such as online examples. For code from our standard routines, we retain ownership of the code, and grant an unlimited license to our clients.

[4] The Windows 10 shell and various parts of Office 2016 are written in UWP. This gives it excellent prospects for a long lifespan.

Conclusion

We think getting the shell right is the key to a successful XAML line-of-business application that is larger than just a few screens. It offers many benefits, including:

- Faster development
- Excellent navigation capabilities for the user
- Good visual appeal through common styling
- Easy expansion of the product line in the future
- Feasible transition from older technologies by allowing side-by-side operation of older UI with newly developed UI
- Consistent experience among product modules for both the user and the developer

I hope this white paper has stimulated your thinking about a navigation shell for your own XAML development effort. However, even though there are clear immediate and long term benefits, a successful navigation shell is a challenge to develop, or to refine from existing starting points.

If you think we can help you achieve those goals above for your own platform transition by working with you on a navigation shell and client application framework, the next step is to discuss some of the options, and let us frame some details on a working relationship. Go to www.nextver.com for examples of our XAML work and contact details. Or you can give me a call at (615) 333-6555 home office, or (615) 400-7678 mobile.

Billy Hollis