Why, where, how JavaFX makes sense!

by Björn Müller, http://www.CaptainCasa.com

CaptainCasa is an open community of mid-range business application software vendors – using and developing one common frontend infrastructure for their business applications – the CaptainCasa Enterprise Client.

The characteristics of business applications are:

- “Big.” - Mannny screens.
- “Complex.” - A lot of rules driving the application processing.
- “Used by employees.” - The core users of the application are employees, using the application as part of their daily work. Of course there always are other users (occasional users, anonymous users) as well – but these are not the core ones.
- “Long term oriented.” - The life cycle of an application is lonnnng. Example: The application is developed today, is sold for the next 7 years, and runs at customer site longer than 10 years. This is a life cycle of 17 years to be taken into consideration!

Companies using CaptainCasa provide software solutions in the area of: financials, controlling, logistics, manufacturing, human resources, administration and others.

The CaptainCasa community was founded in 2007 and – at this point of time! -decided to use a Swing-based implementation for its client. We are just moving over to JavaFX – and this is what this document is about.

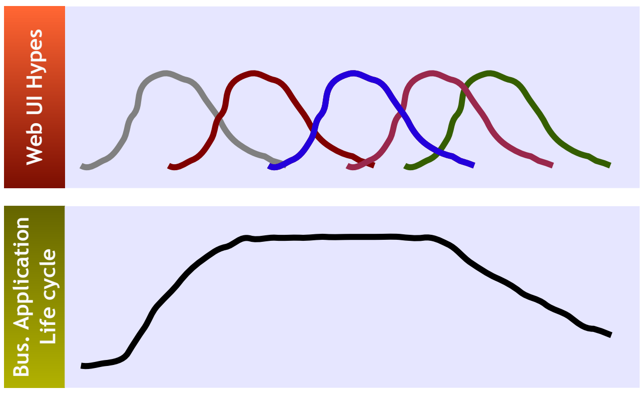
Everything in HTML5! - Or is there still a Window open for native Frontends?

Should you follow Hypes?

In 2007 everyone told us: “you are stupid not to use Adobe Flex!”. At developer conferences the Adobe sessions were overcrowded. ...it took some time and it took one mail from Steve Jobs to end this hype.

One year ago Google Web Toolkit was a hype. Now Google told the world that it will concentrate on Dart as new language for the dynamic web! So, the next hype is on its way...

Be careful with hypes in the area of frontends – they just change too often!



Of course “managers” always want to be on the hype side of life! “Everything must be HTML5” - how many times did I hear this clear and simple sentence, exactly knowing that the person behind does not even know what HTML5 really is... ;-)

You must not exclude yourself from hypes, not at all! You should ride on hypes, when and where it makes sense: you need HTML screens for a lot of scenarios, you need native Apps for other scenarios. So your architecture must be open enough to include any of these technologies in an efficient way.

But: you must not decide on core aspects of your application architecture just by following today's hype! The life cycle of your business application will definitely be longer than the life cycle of today's hype. You will not have the budget to adapt the core screens of your application every 5 years. And you will not have the customers to be amazed about you exchanging core parts of your application frequently...

Once again: there are technical Problems!

JavaScript, HTML5 and CSS – this is what HTML5 pages are made off. And any developer can iterate the problems associated, even when waking him/her up in the mid of the night:

- Ugly language.
- Browser incompatibilities.
- Performance issues.

The situation on the one hand is getting even worse today:

- The number of browsers is increasing: IE, Firefox, Chrome, Safari on desktop systems. And their “partners” on the tablet and phone systems – which are decoupled implementations, everyone with its own “features”.
- The task what can be done with JavaScript is more and more extended: it's not a language anymore to knit together some UI processing, but it targets to be a language for general client development purposes, e.g. with having access to the local file system etc.

On the other hand: There seems to be a way out - you need some framework to help you. Some framework, covering all the complexity or at least helping you to manage the complexity. Due to the complexity mentioned, these frameworks are complex as well.

There are mannnnny frameworks! Some of them are a hype today, some were the hype of yesterday and some will be the hype of tomorrow. And many of them will disappear or go the open source way – when the people behind stop development and maintenance.

What do you do if something does not work in the frontend of your application? E.g. there is a bug that only occurs with Firefox, not with IE – or vice versa? Then you have to be in very close contact with the framework guys...!
You are the application provider for your customer! So your customer will ask you for a solution. You will be the guilty one, you have to provide a solution!

Is “Zero Installation” the only efficient Type of Installation?

There are two options when it comes to implementing frontends:

- You go the HTML way – either by restricting yourself to a certain frontend complexity or by coping with the complexity – with or without framework to support you. Then you have to fight with the problems and complexity involved in going this way, you have to pre-reserve corresponding development and maintenance resources.
  (...BTW: what typically does not work at all is to dictate the browser - “we only run on

Firefox". Companies have browser standards that you have to follow, full stop.)

Or:

- You deliver sophisticated application frontends as easily installable programs (now called "Apps") that are running natively in the client operating system.

Both approaches have proven to work and have proven to be efficient:

- There is enough HTML(5) out in the world on the one hand! So, yes, we are in the era of HTML, of course.
- But it's also the era of installing native applications in a massive way. On mobile and table devices the loading and installing of apps is just a normal procedure – not to mention all the "old-fashioned" installations on desktop computers as well.

Be sceptical, if you get told that HTML is the only way of efficiently bringing functions to the user. It's just not true – thousands of apps and millions of installations per day prove that there is a valid second way as well!

Luckily, Java (FX) is moving into the right direction here: providing the possibility to deliver Java-programs as self containing bundles (should we call them "Java-Apps"?), that run from the scratch on the user's desktop, that do not require any pre-installation and that do require e.g. administrator rights to be executed.

Conclusion from Business Application Point of View

When it comes to the CaptainCasa community then the conclusion is:

The typical screens for employee users are the complex ones: much data on the screen, many ways of interaction (keyboard, mouse, right mouse button, shortcuts, ...), close integration to the desktop environment (file management, MS Office, ...) and sometimes also with direct integration of subdevices (scanners, card readers...). These screens typically form the core of the application and there is a high expectation, that they are part of the application during its whole life cycle. The screens must run reliably on customer site – with a minimum of dependency from the user's client system.

- For these screens the local installation of native client apps makes more than sense!

There are the screens for for anonymous users, where zero installation is a must. Most of these screens are quite simple ones (but styled in a nice way, of course!). The life cycle of these screens typically is much shorter compared to the core screens, because they are often designed for a certain usage scenario.

- Here of course HTML5 makes more than sense. You drive the complexity of the screens, so you know what effort to spend in the areas of browser compatibility etc.

So, we believe that choosing the right framework for the right category of user interfaces is essential – instead of drilling down one framework to be used for all. Of course we dream about one framework to cover all aspects, too – but we just currently do not find one. - And, we believe that it's a good idea to share as much as possible and reasonable if using two frameworks...

Everything is possible! - But what is the Cost?

...typical question: "Is there any functional restriction when using HTML(5) as frontend environment?". People asking this question hoping to get a technical answer like: "because of this and that technical reason, HTML(5) cannot be used". Managers want to have some simple proof, that going a non-HTML way for certain screens is the only option.

The clear answer is: "(Nearly) everything can be done with HTML5!" - It's just a question of cost and efficiency!

Typical business application development groups are not too big. Aside the very big ones (SAP, ...) there are many midrange companies and groups, with limited development resources. So efficiency is inevitable. This means:

- Homogenous development tools and development languages.
- Long-term-reliability of frameworks.
- Minimum of surprises and different runtime scenarios on customers' client site.
- Ability to debug, log, profile.

For small development groups you should think twice if the complexity of developing employee screens with HTML is the most efficient strategy. Or if it makes sense, to depend from a framework that will very likely not cover the life cycle of your application. - You should think twice and check, if the “App-way” may be the more efficient way to make your frontends work on client site.

JavaFX for Employee Screens!

Reasons for choosing JavaFX

We found out, that it's a very valid way to implement the employee's desktop frontend with some native technology. And of course JavaFX is a good fit:

- It's available on the leading desktop operating systems (Win, Linux, Mac)
- It has some painful history, but also a history that is proving a certain commitment of the company behind!
- It is and will be used by mannnny Java developers – as the successor of Swing. So regardless what will happen and if it will once a day be a super-hype or not: there will be a strong community of users.
- It provides a clear and clean architecture – with many enhancements compared to Swing – styling, event management, transitions, scene graph, ...
- It provides the possibility to develop up-to-date user interfaces: with animations, with multi touch, ...
- It is based on a clear and clean language – Java.
- It provides all the professional Java tooling that you require to debug, analyze, profile, log your client.
- It (now) enables simple, “App-like” installation on client side, without any prerequisites.

Indeed it's from our perspective the only native UI environment today that provides all these features.

Experiences with JavaFX

Since mid of 2012 we are developing with JavaFX. Our experiences are:

- Stability – We did not stumble over severe issues. We believe that the way we assemble our controls is a quite sophisticated one, so it's much more than some trivial screens, what we pass to JavaFX. Our overall message is “It's stable!”. - We currently focused on Windows as client platform, we do not have experiences with MacOS or Linux.
- Standard Control Library - ...is sufficient for us. Biggest win when compared to Swing: the browser component and the editor component. We are used to implement complex

controls on our owns, so the extensibility is very important for us. - The only severe issue we are waiting for...: the possibility to use HTML-formatted text for all standard controls (announced for version 3).

- Extensibility - ...building custom controls is no problem at all!
- Environmental Libraries - ...we were used to using some "big components" with Swing, which are not available in FX yet. E.g. the "PDF Renderer" and "Open Street Map Viewer" from the former Swinglabs, or the "SVG Viewer" from the Apache-Batik project. Here we have to check each "big component" how to provide some adequate function via JavaFX.
- Development Process - ...we use Eclipse, we ourselves do not use tools for building screens and as consequence do not miss them. All professional tooling (debugger, memory and performance profiler, ...) are available as part of the normal Java tool stack.
- Performance - The overall impression is: same level as Swing (which is a good one!) when it comes to assembling and rendering screens with many components. Much better performance in the area of transition and animation (well: here Swing only provided very limited functions...). Overall: performance is "no issue" yet, because it's just fast enough...
- Support - you ask a question in OTN (Oracle Technology Network), you receive an answer, often after some hours only. You post a bug in Jira, and the bug is processed. Wow!

So: the overall impression is very positive.

The Architecture around JavaFX

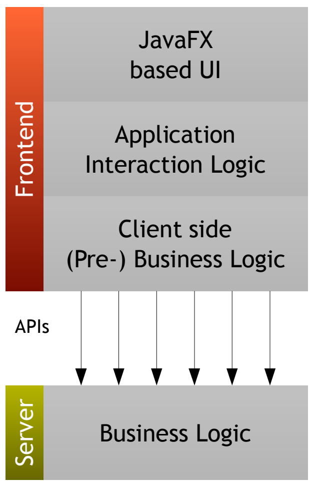
Having chosen JavaFX as UI technology environment for employee screen, we come back to have a look onto the architectural context in which the JavaFX UI is embedded.

The essential question is: how fat or how thin is your client?

(Please note: there is confusion about the terms "fat", "thin", "rich", "smart" client, so we explain what we mean...)

...fat Client Architecture

A fat client is one that not only is responsible for UI issues. It also holds a quite nice amount of application logic. In technical terms: events and data input from the user interface are directly (pre-)processed in application functions on client side. From time to time the client may communicate to some server, calling certain APIs e.g. in order to pass data that was input.


Frontend
JavaFX based UI
Application Interaction Logic
Client side (Pre-) Business Logic
APIs
Server
Business Logic

Scenarios in which a “fat client” makes sense are:

- You want the user to operate locally on the desktop with a minimum of communication to the server. You may even want the user to work independent from any network.
- You are working in scenarios in which you expect such many users, that you need to define your server as stateless server – all interaction and processing state is to be kept on client side.

JavaFX is a perfect environment for developing such fat desktop clients: you have all the control processing through JavaFX and you have all the application processing through Java. - Compare to HTML5, where you have HTML, CSS and JavaScript for doing the same task!

But a “fat client” also involves negative issues:

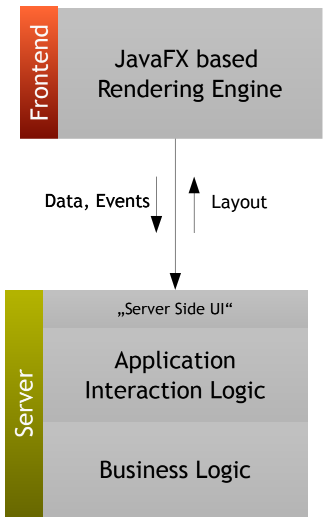
- There is application logic in the frontend... so it will be a quite big frontend in case you have a big application - and it will be rolled out quite frequently in order to fix bugs within its logic.
- There is a high number of server APIs (e.g. web services) that needs to be defined in order to fit to the frontend needs. And there is a high complexity in calling these server APIs in an efficient way – e.g. avoiding the re-reading of certain master data...
- Typically there is some client application logic and there is some corresponding server application logic behind the server APIs. And both have to fit to one another, sometimes resulting in double-implementations of the same logic both on client and on server side.

Typically the “fat client” approach is valid for simple scenarios. - Example: if you want to implement a mail client as JavaFX application, then there are only some few interfaces to read and send mails – and the logic involved is quite clear, too...

The fat client approach it too complex when coming to business applications with hundreds of screens. Imagine the client of an SAP business application to be implemented as fat client...

...thin Client Architecture

So, the way to get out is the thin client. In this type of architecture, the client is a pure rendering engine – receiving some abstract form from the server (e.g. as XML description), rendering this form (via JavaFX components) and passing back user input at the correct point of time back to the server processing.



In this scenario the server is the one to build up the form (XML) and to process user input. The interaction logic of an application, this means the business processing of the form takes place on server side. - The client does not know any business semantics behind the form it renders: if a form represents a purchase order or a form represents a vacation request... - the client does not know about.

The thin client architecture of course means that you need some infrastructure:

- You need the rendering client on client side
- You need the server infrastructure sending the form/layout and processing the user input that is sent from the client
- You need some protocol in between that is smart e.g. not always sending the whole layout as with HTML, but only sending changes.

But if you have this infrastructure then the advantages are:

- The client is independent from the application processing: application bugs need to be fixed on server side only. There is no constant redistribution of the client.
- The client size is small – and independent from the size of the application.
- There is only one interface between the client and the server – the layout channel. There is no need to publish a high number of detailed server APIs to be accessed by client side logic.
- The interaction processing (now on server side!) is very close to the server side application.
- There is only one single place of development – the server side. All coding is done on server side. There is no separation of the development team into “the frontend guys” and the “server guys”.

Conclusion

The choice of the correct architecture is essential.

The “fat approach” is the one you get in fastest – you just take JavaFX and start! But it has limitations when it comes to more complex scenarios.

The “thin approach” is suitable for business applications, that in general tend to have a certain size and complexity. For business applications,it both significantly increases the efficiency of development and decreases the cost of operation at runtime.

JavaFX in front of JSF - ...CaptainCasa!

When going the thin way, you have the option to either develop a client framework on your own or to use an existing infrastructure.

CaptainCasa is an existing thin client infrastructure, that provides a Java based client on client side and a JSF based processing on server side.

Java Server Faces (JSF) now being used for a JavaFX Client

“JavaFX with JSF” - this may first sound a bit strange, because JSF is known as HTML infrastructure. But: it is a J2EE server standard that from the beginning on was abstracted from the concrete HTML browser client – in order to support other rendering infrastructures as well. So it's a perfect fit!

JSF is a server side interaction framework. Its basic processing is:

- On server side a layout is kept as object tree of components. The tree is built up typically be some declarative XML but can be manipulated in any way at runtime.
- The server sends the layout to the client by recursively walking through the tree. Each component of the tree renders its content into some string using a certain format. In case of HTML each component is rendering its corresponding HTML representation. The result of walking through the tree is concatenated to form one layout description, that is sent to the client side.
  In case of CaptainCasa Enterprise Client each component renders itself into a specific XML statement. During rendering there is a delta processing so that the XML page at the end just holds this part of the component tree which really has changed.
- The JavaFX client now receives the XML and renders it. And again, the delta management ensures that not the whole layout is rendered, but that only these parts are updated that have changed.
- The user now does some input. The client registers changes (e.g. input into a field) and waits for significant events to send all changes to the server. An event could be the pressing of a button, or the selection of a menu item. Or it could be the user having changed a certain field.
- The server receives the request and passes all the changes into the corresponding components of the JSF component tree, where they are transferred into the application processing.

You see, JSF is used just the normal way, but now serving a JavaFX rendering client - and not the browser as in default scenarios. From network perspective there is no structural difference between a CaptainCasa JavaFX client and a normal browser – just instead of HTML being sent over the line, now a certain XML is sent.

XML Layout is (and must be!) abstracted from Client Rendering Infrastructure

We started our client in 2007 using Java Swing. We provide about 100 controls that can be added into forms. Now we re-implement(ed) our client in JavaFX .

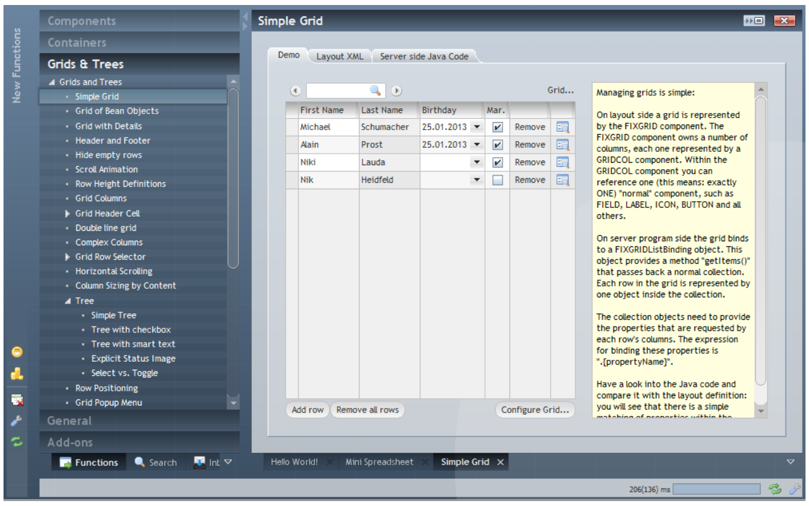
All this is done based on an XML layout definition that from the beginning on was abstracted from Swing. As consequence the JavaFX client is using the same layout description as the Swing client: both are compatible.

This is from our point of view a core issue of the architecture:

- On server side your interaction is working with a stable set of server side controls (JSF components) that is independent from any client side rendering technology.

- As consequence existing users of CaptainCasa can very fast switch from their current Swing client to the JavaFX client.

In short words: the exchange of a rendering technology on client side must not have any (severe) consequences or reimplementation effort on the server side.



	First Name	Last Name	Birthday	Mar.		
	Michael	Schumacher	25.01.2013	✓	Remove	
	Alain	Prost	25.01.2013	✓	Remove	
	Niki	Lauda		✓	Remove	
	Nik	Heidfeld		☐	Remove	

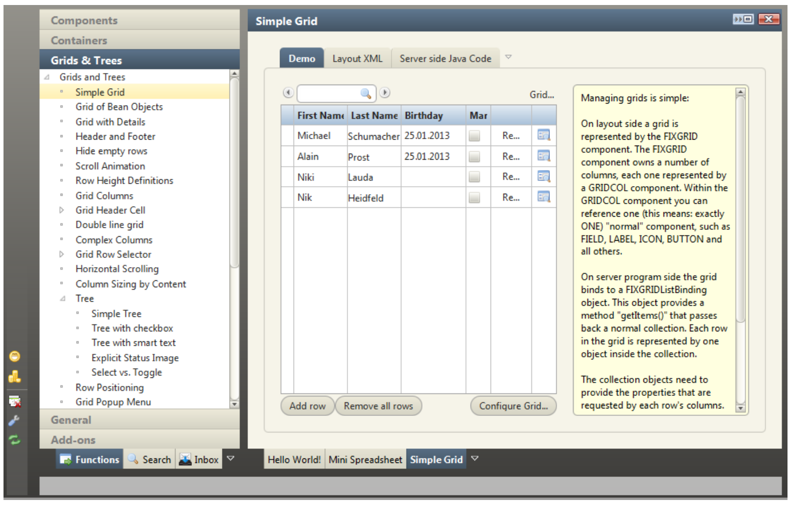
Managing grids is simple:

On layout side a grid is represented by the FIXGRID component. The FIXGRID component owns a number of columns, each one represented by a GRIDCOL component. Within the GRIDCOL component you can reference one (this means: exactly ONE) "normal" component, such as FIELD, LABEL, ICON, BUTTON and all others.

On server program side the grid binds to a FIXGRIDListBinding object. This object provides a method "getItems()" that passes back a normal collection. Each row in the grid is represented by one object inside the collection.

The collection objects need to provide the properties that are requested by each row's columns. The expression for binding these properties is ".{propertyName}".

Have a look into the Java code and compare it with the layout definition: you will see that there is a simple



	First Name	Last Name	Birthday	Mar		
	Michael	Schumacher	25.01.2013	☐	Re...	
	Alain	Prost	25.01.2013	☐	Re...	
	Niki	Lauda		☐	Re...	
	Nik	Heidfeld		☐	Re...	

Managing grids is simple:

On layout side a grid is represented by the FIXGRID component. The FIXGRID component owns a number of columns, each one represented by a GRIDCOL component. Within the GRIDCOL component you can reference one (this means: exactly ONE) "normal" component, such as FIELD, LABEL, ICON, BUTTON and all others.

On server program side the grid binds to a FIXGRIDListBinding object. This object provides a method "getItems()" that passes back a normal collection. Each row in the grid is represented by one object inside the collection.

The collection objects need to provide the properties that are requested by each row's columns.

The first screen shot shows a certain page rendered with our Java Swing based client, the second one is rendered with the JavaFX based client. Yes, there are differences in colors and edges etc. - but this due to a different styling being applied to the screens. - The behavior of the client against the server side processing is one and the same.

Summary

JavaxFX or HTML5? - Our clear response is: JavaFX for employee desktop screens and HTML5 for simple(r) scenarios and anonymous usage scenarios.

JavaFX status? - We do not have too many problems and are really satisfied.

Architecture around JavaFX? - Simple scenarios: jumpstart with directly implementing JavaFX. Business Application scenarios: use or define a thin client architecture!

JavaFX and JSF? - Fits very well: a thin client based on JavaFX standards, a server being based on J2EE standards!

CaptainCasa? - ...is “JavaFX on JSF” to be used for the employee frontends of business applications.