Application Modernization

Panther Migration Assessment

Richard Adams

Jan22nd 2018

Contents

1 INTRODUCTION

After a number of years an application will become outdated…..
This is a fact; technology moves on, new devices and hardware become available. User expectations and ways of using applications have changed. Cloud, Internet of Things etc. are there to use and interact with.
As for the organization, new people with new skill sets join the team, and the existing ones gradually disappear taking their skill sets with them. Eventually the application looks 'old and tired' and now we don't know what to do with it.
So, along comes the first management decision. "Is the application functionality still relevant for the business?" In many cases it is; we have built our business around the application and it does everything our business needs, but we would like it to do more.

Next comes the question; **What shall we do with it?**

Basically, it boils down to this.

- ❖ Do we leave it as is (if it's not broken – don't fix it)?
- ❖ Or do we need to modernize it?

Let's consider modernize; the options we have are, **re-use, re-write, or re-place.**

Re-place
Let's start with the "re-place" option.
We can buy an application ready-made, and hopefully customize it enough so that we can still run our business the way we want. Having an application that "sort of runs" the way you want it to, and adapt your business to suit can sometimes work. Customizing the standard application to make it fit on the surface is a quick and cheap way of achieving our goal. But in reality, it can turn into a long expensive effort that never ends because you can never get it to do everything that your current application does. So, you end up changing the way you do business to suit the app. That can be acceptable to some.
However, we believe our app must do what it does now, so let's consider the "re-write" option.

Re-write
In my experience, the decision to re-write an application is influenced by:

- Lack of knowledge
- Skill shortage
- Desire for "new" technologies
- Not realizing the assets you already have
- Perceived rapid development

I have seen decisions based on the ability to knock out User Interface screens rapidly and then using that as a yardstick to estimate the total effort to create new applications. The mechanics of building UI screens is probably less than 5% of the total effort to build an application. Just look back over the time spent on the existing application, and realistically see how many developer/tester/production years it has taken to get the application into its current state.
Do not be fooled into thinking that a re-write will be that much quicker.
Remember, over time the application has been enhanced in ways the original application wasn't ever thought of, and more than likely there is code in there that nobody remembers. However, the business has evolved to depend on that system, and functionality can very easily get lost in the re-write.

Finally, the “re-use” option.

Re-use
So, having considered all that, say we decide to stay with what we have, but modernize it. I am a firm believer in anything new we do to an existing application, would be to migrate it to a Web-based environment.
This environment allows the mix of technologies that are not generally available in the “legacy” system and environment, let alone the total cost of ownership to manage that. It allows us to mix in “new” technologies without throwing away the old, thus removing any skill shortages we may have. We are not spending our development budget on “re-inventing the wheel” but using a portion of that to modernize the existing. You will also be freeing up the budget to spend on developing new features and functionality that will increase efficiency, productivity or develop new features/applications to increase business revenues.
Plus, this opens many more ways for people to access the application, possibly increasing the user base and perhaps more revenue streams.
Because of this new architecture, we can realize the benefits of the Cloud (external or internal), application maintenance (Dev-Ops), etc. which all lead to more cost-efficient applications and lower on-going maintenance.

1.1 Panther Web

The above applies to any application developed in any technology, but obviously this document concerns our JAM/Panther products and addresses the steps in modernizing a JAM/Panther client-server based system to the Web.
Really, the first step is to do a deep dive on the existing application and get a full understanding of the system, how it’s used, what resources it needs, printing, documents, databases, hardware, devices, etc. what technologies and or other systems it uses, and the size of the user base and how that relates to scalability.
One of the main goals of this deep dive is to determine whether it is a “*data-processing*” system, or more of a “*windows-y data entry*” system. This is important, as it can affect greatly the approach needed to modernize your application.
Data processing systems, like those developed in COBOL and “character-based screens” tended to be a linear programming model and very user interactive, such as field entry/exit and tabbing events. It would take a lot of screens to incrementally process the data before a function can be completed, leading the user “by the hand” throughout the process. This requires the application having to control cursor movement and display output.
Data Entry type systems tend to be more individual screen-based functionality, with most of the data processing done on the backend in services. The screens would be used to collect data then hand off processing to complete the process, similar to the current methodology with SOA (Service Oriented Architecture).
In fact, JAM/TPi, released in 1995, was really a precursor to the current architecture we have today. If the application was designed according to the MVC architecture, then it is really simple to move the application to the Web.
Another detail worth mentioning is that we are moving from a “conversational” application in a client server environment to a “get-response” application in a “stateless” environment. The applications process space is now shared among users so “global” data and user’s cache requirements need to be addressed.

2 Modernization

Having reached this point, then we have decided to modernize our JAM/Panther Application. The following describes an actual project I worked on in modernizing an existing "Data Processing" type system to the Web, I will detail all the work we did showing techniques we overcame and modifications we had to do to the code base to get it to work in the Web environment. All this had the mandate that NO business logic could be changed, and that the application had to process data the same way in the Web as it did in the character-based system.

2.1 User Interface

Before any modernization takes place, I would recommend bringing in a User Interface expert to review the existing application and mock up representative screens showing how it would work and what they would look like in the Web.
"Buy-in" of the business user was crucial at this stage of the project. It was important to demonstrate to them how the existing application would be migrated, and to ensure them that the existing functionality would not be "lost", but also at the same time showing what would be possible to be augmented to their system in the future. Be aware that going to the new architecture and environment brings in a host of new functionality, capabilities and features that the existing users, and possibly developers may not know are achievable.
The first thing a user will notice going from "client-server" to the Web is the navigation differences in a Web-based environment and how "windows" type systems are accommodated.
The result of this was a User Design Guide, Style Sheet, a framework and templates for the developers to utilize in their migration efforts. In our case we used Bootstrap as our main Web-based framework, jQuery for the JavaScript code and HTML templates for the Panther screens. Grid functionality was utilized with Bootstraps' DataTables and Panther's hidden widgets.
Panther can generate screens directly, and is useful for unit testing, but they are not good enough for user acceptance on a modern Web system. However, the ability to easily incorporate HTML Templates give us more control, and by placing tags in the HTML where we want Panther to emit the widgets, we can easily and quickly achieve our proposed User Interface. In fact, this is exactly the same method that you would develop JSP and ASP pages. Moreover, in another system I was involved with, we re-used some of these HTML Templates and with some slight modifications they were converted into JSP pages. We did this because we were developing new features and enhancements to our system, but developed them with Java instead of Panther, due to the skill sets of the developers we had at that time.

2.2 Deep Dive

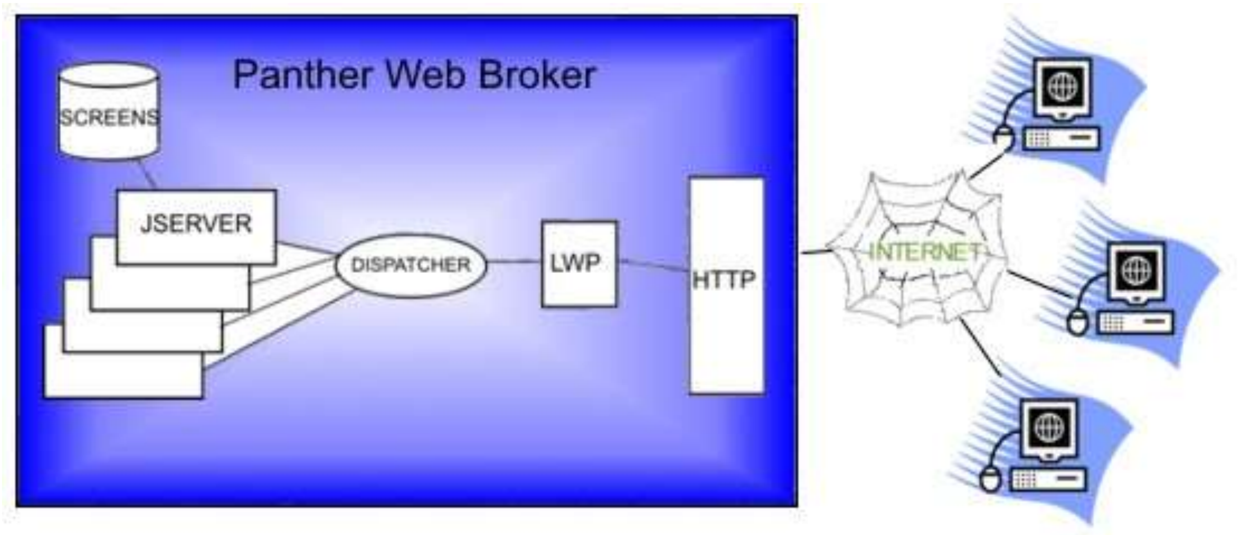
Again, before we undertake any modernization, it is necessary to get a full understanding of the existing system.
The main areas to consider:

- System Architecture and Development environments
- Multiple JServer – multiple servers
- LDB access – pass through and Global Data
- Screen Entry/Exit and Unnamed procedures
- Caching of Data
- Temp Tables
- Windows/form screens.
- Multiple windows – data-exchange
- Screen validation
- Field validation
- Field entry/exit
- JPL sm_gofield usage
- Cursor control
- Widget display control
- Function Key – keyboard control
- Sm_message_box functions
- Report writer – batch processing - printing

2.3 System Architecture

The existing application was terminal based with runtime JAM components tied to a terminal. All these runtimes were hosted on AIX machines. Panther Web basically moves these runtimes into an Application Server component that is hosted on an Application Server - typically Tomcat, or Liberty, Glassfish or any JEE Server.

Figure 1



LWP – Light Weight Process
We are not limited to just the one Panther Web Broker; there can be multiple servers running behind a Load Balancer.
Each JServer establishes a connection to the Informix Database; instead of each user connecting, Panther Web Broker will use a generic "web" user.

2.3.1 Virtualization:

It was proposed that the best design going forward was to base the system on a virtualized platform. This can be Cloud based or an in-house Cloud model. As the corporation was already planning on moving to a Virtualized system we decided to house it internally.

- Pros & Cons
    - Allows scalability by adding new servers to the system
    - Separates the batch processing from Users Runtime
    - Isolates processes from each other, minimizing risk
    - Add new servers for future functionality and interoperability (PEG*/SOAP, 3rd Party systems)
    - Quickly deploy new development environments minimizing the onboarding process for new developers
    - Offshore development through AWS Panther Cloud system
- Servers Platforms
    - RedHat 7.2
    - Windows Server (2012 or later)
- Informix on AIX vs Redhat
    - High Learning curve for Informix Admin
    - Licensing costs
    - HA and DR Configuration
    - Lack of support for Application servers and high cost

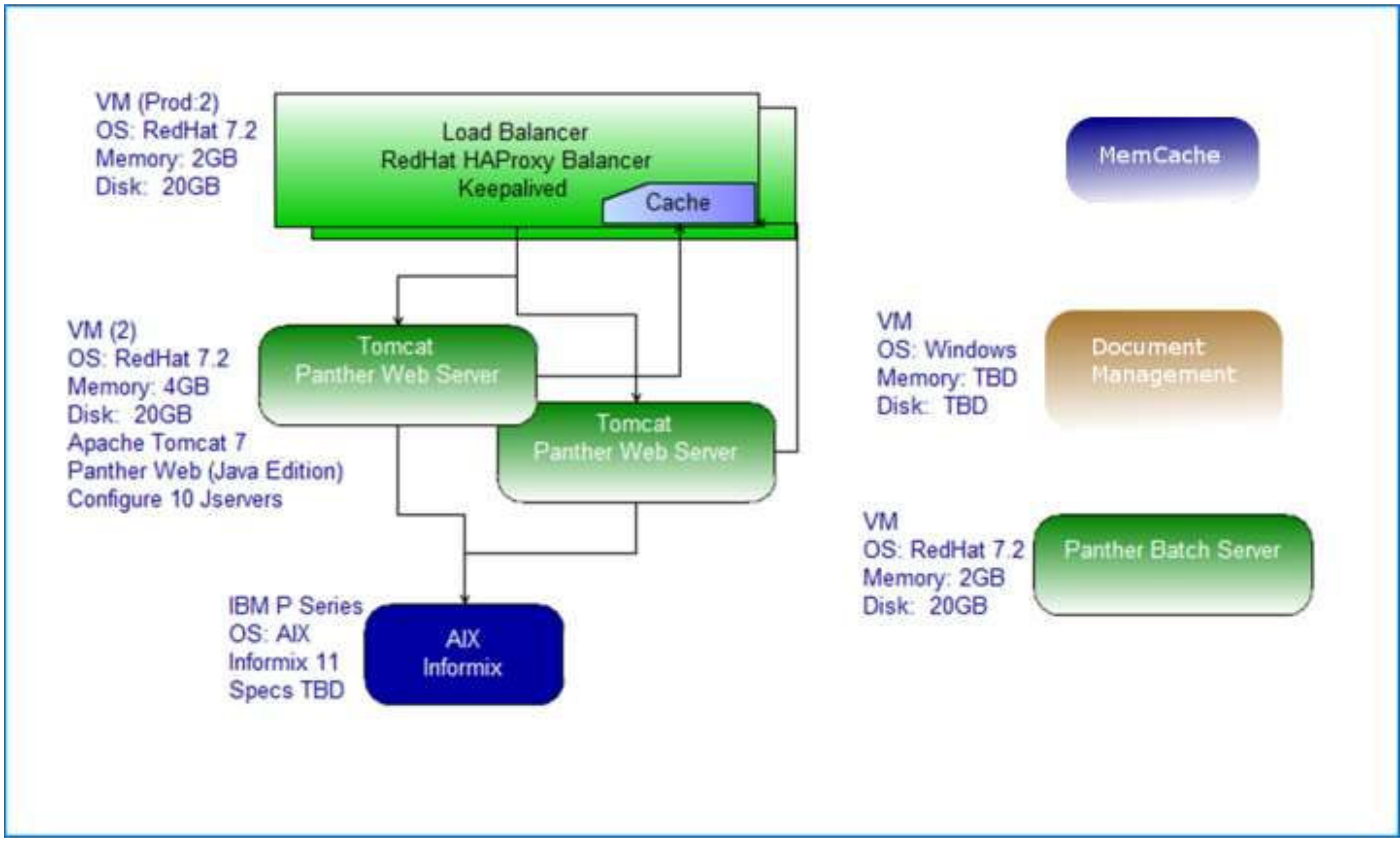
Informix will remain the DBMS on AIX as it was deemed not necessary just yet to move the database and had no impact on the modernization effort.

* PEG – Panther Enterprise Gateway

2.4 Virtualization Architecture

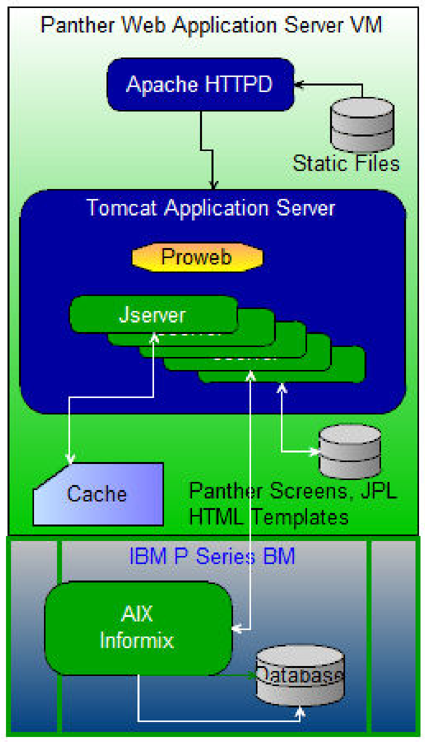
For the production system, each VM was based on RedHat Linux 64 bit. These VMs were configured using the Prolifics Panther Cloud images as templates. Moving forward, the plan is to go to Docker for better hardware utilization and deployment efficiencies in the future.
The Document Management System was a 3rd Party .NET system and hosted on a Windows Server, and was a major part of the new features and enhancements that were enabled by moving to this new architecture and where we can integrate the 2 systems seamlessly through the Java API.
Cache was provided by RedHat MemCached.

Figure 2



2.5 Development systems.

For the Test and Development systems we configured all the systems into one VM and all source code was managed by Team Foundation Server (TFS).



The PDC (Panther Developer Client) systems were used for all Web migration. This system is Eclipse based where Panther JPL code was migrated (JPL-E Plugin), JServer C executables were built (Eclipse C++ Toolkit), Java utility functions were created for application enhancements and Panther Screens were stored in ASCII. In addition HTML, JavaScript, and all the Style Sheets were developed using this Eclipse system.
Tomcat 8, run in Eclipse, was used as the Application Server for debugging purposes, The Redhat Apache HTTPd server was used for serving static data and bridged to Tomcat with the Proxy support module.
The Panther Web Debugger, which is browser based, was employed to debug the JPL Code. Panther Debugger uses a Websocket Server running in Tomcat.
The JPL Code in the development environments was all file based so that during debugging, any JPL changes would be immediately picked up, instead of having to reload libraries.
The Panther screens were modified in either Panther IDE, (this was the only time we used the Panther IDE) or ASCII files in Eclipse, depending on the type of modifications required.
And Eclipse macros/shell scripts were used to run the Panther utilities where necessary to rebuild form libraries, etc.
The Team Foundation Server plugin was used for source code control.

3 Migration Processing Strategy

Migration of the system can be split into 2 main areas, Foundational and Screens. To assist in these areas, a number of Java program utilities were created or customized from the Panther Migration Toolkit to assist in the migration effort.

3.1 Foundational

These were functions we needed to perform on screens and JPL code in the system.
For example, in JPL we can use global variables and normal vars declarations. Unfortunately, Panther is very forgiving in how and where these can be defined throughout the code. Over the years, developers can be “sloppy” in their use, so we used a Java utility to scour through all the code to identify the global vars and to correct the code. There were about 4 or 5 Mass Migration Utilities from the Panther Migration Toolkit needed for this system.

3.2 Screens and JPL

This is the procedure where the developer will run a series of Utility Programs to convert the screen and JPL code or generate code to be copied by hand (cut and paste) into the JPL, Panther Screen, HTML or JavaScript in Eclipse.
For our migration we had 7 Steps:

1. Pre Migration.
   Check JPL for syntax, clean up code ready for migration

2. Migration Step 1 for Data Entry or Grid-based screens
   Utility generated HTML and JavaScript code and fixed up Screens and field attributes required for the Web.
3. Migration Step 2
   Utility that generated the Zoom button code. This was to convert the Zoom feature in Panther to Web Buttons
4. Migration Step 3
   Utility that generated the Ctrl functions for individual screen-based menus.
5. Migration Step 4
   Utility that created a single line text widget from vars declared in unnamed section and modified JPL code to suit
6. Migration Step 5
   Utility that fixes up the XML Tags for the Memcached
7. Migration Step 6 (Menu in .mnu file)
   Utility that generates html and JavaScript from the mnu file.

4 Migration Considerations

The mandate was to **NOT** change any JPL Code, and to **NOT** lose any functionality. Effectively, migrating to the Web does not mean losing functionality, or substantially rewriting code, but it changes how you handle events and user actions. The user tabs around the screen, field entry and exit events are run, the user enters data into fields and data is validated. Once the foundation is in place, the major work needed to be done is making the Web system run the event code when screens are submitted.

4.1 Client-Server Web considerations.

In the Web environment a screen is “got” and displayed in the browser. When the screen is “posted” back, the screen is submitted; the submit function is executed and then the screen or a new screen is returned to the browser. This is a typical “round-trip”. In Panther Web the JServer can be thought of being the equivalent to the JAM/Prorun process used in Client/Server, and does actually include the Panther runtime system in it.

The main difference with Panther Web is that in the JServer, every time a screen is requested, it opens the screen, performs its entry events, processes and Control code, then exits and closes the screen again and processes all exit events. In a multi-JServer environment there is no guarantee that the same server will be requested the next time the user requests a screen, therefore any global data that has been added to that JServer process space will not be in the other JServer’s process space. For all intents and purposes, the JServer is “dead” to us. Therefore, one of the main things to do in the Web environment is to rebuild the user’s environment each time a JServer is accessed with a screen request.

Panther does have an inbuilt file-based Caching mechanism, but it does NOT store everything related to the screens, such as field properties, and is difficult to use in a Tabbed environment with multiple windows being displayed. It is also slow with large global data and difficult with multi-VM architectures.
In light of this, we decided to use our own cache (Memcached on Linux) where we could have finer control of the user’s state information.

4.2 Screen Processing Strategy

4.2.1 Sequence

When the user requests a screen, Panther Web will serve the screen as an HTML document. In the Web environment, for each screen request or submission, Panther Web performs these steps:

1. Restores any open bundles (created by the send command) and global JPL variables that come back from the browser or have been cached on the server.
2. Opens the screen by calling sm_r_form. This causes the following screen entry processing:
    a. Executes the unnamed JPL procedure.
    b. Initializes the transaction manager which issues a `START` command.
    c. Executes the default screen function.
    d. Executes the screen entry function.
    e. Merges with the LDB.
    f. Executes the `AUTO` control string.
3. On a `POST`, restores the contents of fields as specified by data from the browser and from the server cache.
4. On a `GET`, processes variable assignments from the `QUERY_STRING`.
5. If a button was pushed, calls sm_gofield for that button.
6. Calls the JPL procedure `web_enter`.
7. If a button was pushed, calls sm_ungetkey for the NL logical key.
8. Invokes the Panther executive to perform normal processing until key input is required. This includes:
    a. Execution of the default field function.
    b. Execution of the field entry function for the first field.
    c. Processing the NL key from Step 7.
    d. Execution of the application processing defined in the screen.
        i. Error messages, instead of requiring a user response, are processed by automatically executing the processing assigned to the default push button.
9. Calls the JPL procedure `web_exit`.
10. Generates HTML for the screen and sends that data to the browser.
11. Closes all open windows, which executes normal screen exit processing. Note that this cannot affect the already generated HTML.
12. Deletes appropriately marked JPL variables.

The important item to note in the above is that data saved in the cache by Panther Web is NOT available to the screen entry function, but is readily available in the web_enter() function. There are several ways to get around this and it really depends on how the existing system is programmed. In our case, we developed a solution using memcached as outlined below.
The existing screens are not simply 'GET' and 'POST' events; the character application will allow the user to step out to other screens, from menu picks, field exit events, etc. and therefore is a possibility that the screen may have data entered that could be subsequently lost.
In the character environment, each user runs their own process and so all these global variables and LDB are private to the user in their own process space. In the Web, the process space is shared and therefore could be visible to any user request.
So, it was necessary to remove ANY functionality from the "unnamed" procedure in JPL code to ensure the screens initialization code was run once and once only. All the screens' initialization code was moved to the screens' entry function and any variables defined were made into hidden screen fields. These would subsequently be stored now with the screen data.

We created a template screen entry function JPL code that we used in every JPL screen, and we moved any unnamed code into the appropriate place. At the same time, we moved ALL code defined in the screen into the file based JPL. That way we could maintain all JPL code in one place. The screen's data caching is NOT performed in the Default Screen events, but is executed programmatically via a Java call from the JPL. The web_enter() function will generally retrieve the screen's data, and the sm_jwindow replacement JPL function will save off the data to the cache. This was a Java function that, similar to the property caching, would create a JSON structure and be stored in Memcached. A hidden benefit with this approach is that another system, 3rd party or otherwise, can be integrated into our system and use Memcached to retrieve data that is present on the screen, which is safer than passing it in the URL.
Other hidden fields were added to every screen to enable the passing of data. So browser events would set up these "hidden fields" and submit the form. Then web_enter() would inspect these and then run the appropriate code. This was used for change events, running Control Functions, Tabbing events, etc. These were also used to notify us if we called a new screen or returned from a child screen (popup or otherwise).

4.3 LDB Global variables and cache

To minimize the impact on the existing code base, it was necessary to enable the LDB and global variables per user's request. The approach taken was to use a separate caching mechanism to hold the Users LDBs and apply them at runtime for each user's request.
By using Panther's built-in event handlers and Linux's Memcached In-memory Grid utility, we can ensure the data is recovered in time for the code to use it, and is automatically stored when the user or the application continues on its path through the system.
Panther has built-in XML import and export functions which were employed to save LDB data to the cache through the Panther's Java Interface.
All LDB fields were modified to include XML tags. This was easily automated and performed in a migration/conversion program.
By attaching a Java Class to each screen, this class acts as an automatic screen entry and exit function. These get run as the default screen functions; the user's LDB is restored from the cache before any screen processing takes place, and data is saved off to the cache after all screen processing is completed. This allowed the availability of LDB variables at the right time.

4.4 Cursor placement in character and the web

There may be instances in the JPL code where field numbers need to be verified to confirm cursor positioning, or use cursor control such as sm_gofield() etc. This will still work in the Web when the screen is submitted, and the field entry and exit events will be executed as though they were in client server mode. However, you need to be really careful in chaining all these events together.
Note that in Panther Web, Panther will move the cursor into the first "enterable" field on the screen. i.e. not protected from focus or input and will not put it where it was in the Browser.
So, for example, if a statement like
`if @field_num(@screen_num(0)->fldnum)->name == "s1234_widget"` is encountered, in the character application, it will get the name of the widget where the cursor is positioned. In the Web, on submit it will not be there, but our JavaScript functions populate the hidden fields with the field name and occurrence number, etc. so a safe way is to change the JPL code to use the value in the hidden field instead.
Because sm_gofield() cannot be called in any of the entry functions as it is overridden, we could put the sm_gofield() in an APP key and put it on the keystack to be processed later. However, this is fraught with issues, so the safe way is just to use the hidden fields.

4.5 Field Entry/Exit/Validation Events

In the Client/Server model or windows application, the application may be developed with a lot of user interaction within the screen itself. The user tabs out of the field and a backend or service action is invoked. The user changes some data and then backend processing is executed. Some applications leave all backend processing when the screen is saved. This later one is easier to handle but in our application, we had many instances of these field events taking place.
There was no easy fix for this. We looked at the field events, and determined whether they were necessary when the user changed the data or gained or lost focus. We took the approach that ALL field entry/exit/validation functions would be removed from the fields just in case a sm_gofield() was performed somewhere in the code. Then in the browser we applied the appropriate event to trigger this backend process. The JPL web_enter() module processed these hook functions on the browser's behalf.
Any interaction in the screen will require a full round trip submit of the form to the server; the screen will "flicker" and return to the user. On return though, cursor positioning was achieved in passing data back to the browser in the hidden fields and our "ready" function would position it accordingly.

4.6 Dynamic Field Properties

A Panther default screen function was developed in Java to automatically store and retrieve all the screen and field property settings to Memcached. These properties could be dynamically changed at run time, and Panther does not store these in its default cache. This Java function traversed the screens' widgets and created a JSON structure of the fields and all their properties. On Screen entry it would retrieve the properties from Memcached and apply the properties that were stored in Memcached. We employed the user's id, session ID, Tab Frame ID to uniquely identify the key in Memcached.

4.7 Main Screen

For the main screen or landing page for the application, it was decided that we would have a simple tabbed Interface where users could run up to 4 disparate functions at the same time. We created an iFrame template screen that we linked to the logon screen in the parent frame. After a user has successfully signed on, a menu is dynamically generated from the database.

4.8 Migration templates

In the system, we identified 4 or 5 screen designs that were used throughout the application. From these we created Html Templates that we used in the migration utilities. This ensured a common look and feel throughout the migration process.
Also based on the CSS Style Guide we added some CSS styles for our different widgets and grids, etc. However, the majority of styles we used was from the Style Guide that was based on the standard Prolifics Style Guide, but with some customizations for the client. (Mainly logos, images etc.)
Finally, we created a few JavaScript templates, again used in the migration utilities for the different functions that were generated by these utilities.

4.9 Menu Handler

The user's functional menu was generated dynamically from the database. The character application also used multiple JAM screens for the main menu navigation. Some Individual screens had JAM menus attached that had functionality to open screens, invoke Ctrl functions and JPL code.
One of the main recommendations of the UI design was to come up with a suitable menu replacement. There are many menu options, but for those systems that have a dynamic menu creation, it is best to let Panther create all the menu definitions and then let the Browser create whatever menu system the user wants from the menu data. This was passed in a hidden, multiline text field. The field was exported to html and we used Java to create a JSON format which most JavaScript menu systems require.

4.10 Screen Menus

Screens that have a Panther .mnu file associated with it were converted using migration utilities to HTML and JavaScript code, to use Bootstrap menus. This code was added to the HTML Templates. Generic menu functions were developed in JavaScript and JPL to handle any actions invoked by the user.

4.11 Temp Table Redesign

Due to the stateless environment, and the multiple JServer, each having their own database session, the temp tables created from the JPL will not be in scope if another JServer processes the request. However, if the lifetime of a temp table is contained in the server, then it can be left as is. If a temp table spans multiple requests, then it must be converted to a real table, and must be uniquely identified.
To make these temp tables unique to the user, it was decided not to alter these persistent temp tables structure by adding columns to contain the unique keys, etc., that would have been a huge undertaking, but we used a naming convention similar to the way we identify the user's session data in memcached.

4.12 Database Cache

Instead of using the File server cache, we do have code developed in another system that can use the database as a cache instead of the file system. Soon this will be changed to use Memcached as Panther's cache. That way there will be no issues with multi-server architecture, plus it will be extremely performant.

4.13 Error Handling

Error messages in Panther Web normally just appear as text at the top of the page, pushing down content that would otherwise be at the top. We replaced this by displaying the messages in a Popup alert style format using JavaScript code and the HTML class attribute.

4.14 Interactive Messages – WebSocket Server

One of the features of character Panther is to call sm_message_box() function to solicit users input midway through the processing. This is a total anathema for the Web. The Web is a complete

roundtrip get/respond mechanism. It cannot stop processing on the backend and then continue onto the next request.
Apart from totally rewriting the existing logic, we needed a "Push" notification feature built into Panther. We achieved this by creating a WebSocket server and added a WebSocket listener in the Main page using JavaScript code. We also included a Java replacement to the sm_message_box() function that sent a message to the WebSocket server that relayed on to the Browser. The browser would pop up the message, and send back the user's response to be relayed back to the Java client that was interrupting the backend code.
Obviously, this would tie up the JServer while the user responded, thus we had a timeout on the popup window that would return the 'default' value in a few seconds, just in case the user didn't respond.
Having multiple JServer mitigated any performance issues, as these messages were not that frequent and in a production system we could quickly deploy more servers to the cluster, if we needed to.

4.15 Report writer

Running the Panther Report Writer and displaying the output in character needed to be changed from the way it works in Client/Server mode, as the reports are now generated in the Web on a Webserver.
So, in the Web version, we ran the reports as before but generated PDF files. When this was complete, we made a call to the Websocket server passing the report name and location; the browser would then make an 'ajax' call to fetch the PDF files and display the reports in a popup browser window. The user could then print the reports to any of their attached printers throughout the organization.
This was a better solution than generating Postscript files, maintaining printer configurations on the existing AIX Server, and having to have Postscript printers to generate hard copies.
At the same time, these reports would be passed onto the document management system automatically.
A future goal of this migration is to eventually have a paperless system within the organization and fully integrate the 3rd Party document management system into the system.

4.16 Source Control Integration

As all development was done in Eclipse, we had the ability to use any source code control system. We chose to use Team Foundation Server which also provided Agile project management within the team, thus Panther migration could be fully incorporated into the organization's systems.

5 Conclusion

The above details all the steps we took in the migration of a Panther Client/Server character UNIX based system to a modern Web and mobile enabled system, resulting in a system on which future features and enhancements could be incorporated quickly and easily and allowing us to use whatever technology and skillset is deemed appropriate in the future.

Without throwing away years of effort, but **re-using** it wisely, we could - at a fraction of the cost - modernize our system, and start thinking about the future.

Prolifics.

IT:Customized to Your Advantage

Headquarters

24025 Park Sorrento, Suite 405
Calabasas, CA 91302
P: 818.582.4952 / F: 818.224.5269

NEW YORK
5 Hanover Square,
Suite 2001
New York, NY 10004

WESTBURY
865 Merrick Avenue
Suite 10S
Westbury, NY 11590

ORLANDO
20 North Orange Avenue, Suite 1211
Orlando, FL 32801
P: 407.641.0800
F: 407.420.9484

PLEASANTON
7041 Koll Center Parkway, Suite 135
Pleasanton, CA 94566
P: 408.653.2020

BERKSHIRE
Suite 1-10 | SoanePoint
6-8 Market Place
Reading | Berkshire RG1 2EG
United Kingdom

HAMBURG
Rodingsmarkt20
20459 Hamburg
Germany
P: +49 (0)40 890 667-88
F: +49 (0)40 890 667-99

OTTAWA
100-237 Argyle Avenue
Ottawa, ON K2P 1B8
P: 866.639.1451

TORONTO
100 King Street West
Suite 5600
Toronto, ON M5X 1C9

INDIA
Hyderabad (Registered Office)
5th Floor,
DHFLVC Silicon Towers,
Survey No 14, Kondapur, Hyderabad - 500032

Plot #226, Road #17
Jubilee Hills, Hyderabad - 500033
P: +91 40 39991999 / +1 818 301 4945
F: +91 40 23114651

8th Floor, Block - 3
Plot #229-232, APHB Colony
DLF Commercial Developer Ltd. SEZ
Gachibowli Village, Hyderabad - 500029

9th Floor, Building # 20
MindSpace SEZ,
K. Raheja IT Park Pvt. Ltd, HITEC City
Madhapur, Hyderabad - 500081

Cerebrum IT Park
Units 1&2, First Floor, Building# B3
Kalyaninagar, Pune - 411014

TVK Industrial Estate
7th Floor, R R Towers - III
Guindy, Chennai, TN - 600032