



IBM i Modernization - The User Interface

By Nathan Andelin

Introduction

Please acknowledge that this document is the result of a private research project, with me studying the articles of Nathan Andelin, but attempting to make it available as an offline resource for myself (I {Marinus van Sandwyk} am "old school" and prefer a piece of paper, a pencil and highlighter whilst reviewing anything of substance, as I like to make my own notes, sketches and comments based on what has been written).

The best value will be gained by accessing the interactive content, especially to review the behaviour of the active content. Such behaviour will be indicated in the text

This document was compiled with the kind permission of Nathan Andelin

(<https://www.linkedin.com/in/nathan-m-andelin-64241415>)

Copies of this can be retrieved from the [ILE-RPG Developers website](http://www.ile-rpg.org/) (<http://www.ile-rpg.org/>).

The ©Copyright of the contents in this document belongs to Nathan Andelin.

Article Index

<https://rd.radile.com/rdweb/info2/ibmiuix.html>

[Part 1](#) introduces the idea of moving from the IBM i green-screen paradigm to browser user interfaces which have fewer constraints while providing rich and responsive end-user experiences.

[Part 2](#) delineates a number of design patterns which enhance end-user productivity. But they're typically not found in GUI component palettes and page-at-a-time paradigms promoted by popular IBM i modernization tools.

[Part 3](#) introduces front-end driven and single-page application design paradigms. (FDAs and SPAs).

[Part 4](#) addresses the challenge of learning HTML, style sheets (CSS), and JavaScript, which are highly effective and enabling technologies - relevant to IBM i user interface modernization.

[Part 5](#) discusses how browsers "request" resources from IBM i servers, including the types of data and documents which may accompany such requests.

[Part 6](#) introduces a light-weight framework which enables browsers to "merge" HTML templates with JSON data objects to create stunning masterpieces (that which appears on the screen).

[Part 7](#) introduces the idea of UI widgets and how they might improve the application experience for end-users.

[Part 8](#) introduces a light-weight framework which facilitates asynchronous request processing paired with incremental updates to page elements (a.k.a. AJAX / Web 2.0).

[Part 9](#) delves into the creation of web user interfaces in which the layout automatically adapts to a variety of device types and screen sizes (i.e. cell phones, tablets, laptops, and desktops).

[Part 10](#) Continues deeper into "responsive design" with examples and code.

[Part 11](#) Shows and explains a simple application which delves into the shift from the traditional display-file paradigm to the client-server paradigm, and delineates benefits for end-users.

[Part 12](#) Shows and explains a simple application which helps us explore the idea of externalizing SQL in RPG programs; it introduces my *SQL cursor API*.

[Part 13](#) Introduces the idea of using HTML and CSS to produce nicely formatted and stylized reports, suitable for printing.

[Part 14](#) Reviews sample HTML, CSS, and RPG code which may be used to generate nicely-formatted, stylized output which is ready for print.



[Part 15](#) Demonstrates a browser version of a single-page subfile and explains how it works. As IBM i developers move to browser user interfaces, I think it helps to see how familiar design patterns such as subfile paging may be implemented.

[Part 16](#) Shows a browser utility which enables users to upload multiple files to IBM i servers. The files may be subsequently attached to database entities, email, SMS messages, etc.

[Part 17](#) Explores the idea of using web browsers and IBM i programs to send email.

[Part 18](#) Explores the idea of sending text messages to smart phones and similar devices from IBM i applications, using Twilio.

[Part 19](#) Explores the idea of using web forms to prompt for input and generating PDF documents based on the same.

[Part 20](#) Uses flowcharts to depict typical web application design processes, and other aspects of web application architecture.

IBM i Modernization - The User Interface (Part 1)

<https://rd.radile.com/rdweb/info2/ibmiui.html>

This piece offers opinions about options for moving from green-screens to browser user interfaces. It focuses on IBM i-centric alternatives as opposed to the migration of applications (or the UI portion of applications) to other platforms.

Our definition of "other platforms" includes language and runtime environments which run in PASE, because PASE is really a subset of AIX. Though interesting as they may be, the scope of this blog is limited to IBM i-centric options (ILE languages).

Organizations which are reviewing green-screen alternatives are often in a critical phase wherein the risk of failure is high. The loss of confidence which follows a modernization failure often leads to a negative stereotyping of IBM i developers, distrust of the IBM i platform, and costly migrations of applications to other platforms.

Quite a few vendors and branded products fall within the scope of IBM i-centric alternatives. We will not name them, nor review them in detail. But we will offer opinions about architectural constraints within most of them, and compare them to alternatives which have fewer constraints.

Display Files

A number of products support the traditional display file paradigm and extend it to browsers. The lure is that it is alleged to provide the easiest path for developers who are steeped in display files, and the fewest changes to existing code. The display-file paradigm includes "screen scrapers" and open-access handlers.

Experience suggests that the risk of failure of the display-file paradigm is high. End-user expectations in regards to UI modernization often differ from the interests of legacy-application developers. Display files have a number of constraints, and products which extend the paradigm to browsers generally impact the end-user experience in a negative way more than they improve it. Many users cling to green screens after trying an alternative. Others express disappointment.

Downsides from an end-user perspective include:

- Size of data streams increase from 1K-2K, upwards to 30K-100K per request.
- Loss of cursor movement.
- Reduction in keyboard control.

- Shifting back and forth between keyboard and mouse.
- Net reduction in end-user productivity.
- Increased latency between client and server.

Downsides are partially offset by a richer set of UI components, including drop-down lists, check boxes, push buttons, colors, and fonts.

The opinions of IBM i developers are mixed. They confirm that such transitions to browsers requires few changes to existing code. But they cite a loss in productivity by having to maintain traditional display files in addition to GUI extensions, using separate tooling.

More concerns are raised by web application developers who are experienced in UI design (using HTML, CSS, JavaScript, AJAX, responsive design and such). They view the display-file paradigm as a severe architectural constraint.

Open-access-based tooling generally includes a browser-based (JavaScript) applet which implements an interface comparable to a 5250 emulator (except the data stream is more verbose). The applet overrides the traditional browser interface, and introduces constraints. Some of these constraints will become more clear in subsequent sections.

Request-Response Cycle

Normally, web browsers initiate requests, then handle server responses. The display-file paradigm overrides this normal request-response cycle, and replaces it with a pattern of writing "records" to a "display", then reading "records" after "waits". This is problematic in that in many cases the user experience may be improved when browsers are allowed to initiate:

- Requests from multiple in-line frames.
- Requests from "timer" events.
- Asynchronous background requests.
- Requests from toggling UI elements, such as check boxes, radio buttons, and such.
- Long polling requests.
- Requests from keyboard events (i.e. Google Suggest).

In other cases, the user experience may be improved by enabling custom-written JavaScript to handle events which might otherwise require server logic and resources under the display-file paradigm.

Functionality and user experience is also enhanced when interfaces support push-notifications to browsers for such things as alerts, messages, and changes to screen content.

When evaluating user-interface options, please consider traits delineated in subsequent sections.

Fixed Positioning vs. Responsive Design

Beware of tooling which enables developers to drag, drop, position, and set properties of GUI elements (i.e. tables, input elements, etc.). In most cases, the tool enforces fixed positioning and sizing of elements, and page-at-a-time processing.

A better alternative is tooling which supports relative positioning and automated sizing of UI elements, and supports incremental updating of UI elements in response to asynchronous background event processing.



Responsive design is a fairly broad topic which could take several book chapters to summarize. In our opinion, it's an important practice and something which is NOT found in most IBM i-centric UI products.

Stateful vs. Stateless

There has been a lot of buzz about the "stateless" nature of web interfaces. Don't take that to mean that your applications should be stateless. Quite to the contrary, complex applications require stateful resources such as filtered result sets (cursors), record buffers, and variables which hold unique values for individual users. From a developer perspective, the relevant question is what may be required (if anything) in order to manage state.

In regards to IBM i-centric alternatives, state may be maintained automatically via interfaces which launch new "JOBS" when new users request URLs which map to application entry points. Or perhaps new JOBS are launched when users click on menu items in a web portal.

Alternatively, programmers can write code which retrieves "session" properties at the beginning of the request-response cycle, updates them while processing requests, and saves them for possibly subsequent requests from the same user. In complex applications, this may not be a trivial exercise. In our opinion, IBM i-centric interfaces should support both options for managing state, and especially the option of launching new JOBS which frees developers from a requirement of writing code to maintain sessions (they often have more important things to worry about).

The option of launching new JOBS when menu items are clicked is especially relevant to the development of complex, broadly-scoped applications. Partly because it frees developers to focus on application functionality rather than managing sessions via code. Partly because the request-response cycle is streamlined (i.e. CPU cycles are not not required to alternatively restore, then save session state.)

Does the interface enable end-users to "END" JOBS they started when they clicked on menu items? That's a key to operational efficiency. Does the interface enable administrators to configure a maximum inactivity limit for JOBS which may have been abandoned by users? Does the interface automatically "END" inactive JOBS when limits have been exceeded? Do users receive appropriately formatted messages when sessions automatically end?

IBM i actually provides unique value in its ability to manage thousands or even tens of thousands of processes (JOBS) concurrently. The value of this cannot be overstated; Particularly in regards to supporting broadly-scoped applications for thousands or tens of thousands of concurrent users on a single server.

Caching Resources in Browsers

User experience may be improved by interfaces which enable the caching of resources (i.e. HTML templates, JSON objects / arrays, JavaScript, CSS, etc.) locally (in browser cache). This facilitates certain types of screen updates without the need of making additional requests to servers. This is essentially the opposite of display-file and page-at-a-time paradigms promulgated in screen-scraper, open-access, and other IBM i-centric tooling.

This is an essential trait of the single-page paradigm which was made popular by Google.

Web Portal vs. Roll-Your-Own

Web portals provide infrastructure which may be shared by all applications which run within them. By "infrastructure" think authentication of user credentials, single sign-on, automated recovery/reset of user credentials, user profile and user group management, group authorities, menu configuration, application configuration, and such.



Web portals enable the launching of any number menu items concurrently in tabbed frames, and toggling between application instances, as opposed to exiting from one menu item in order to access another.

Does the vendor offer a web portal along with developer tooling, or do you need to build your own?

Code Generation and Utilities

Does the tooling include code generation for basic inquiry, maintenance, and reporting? Are the generated user interfaces an improvement over existing green screens? Does the code implement an MVC design pattern? Is the code modular and easy to follow? Do generated applications make use of utilities for such things as automated pop-up lists and report parameter prompting?

The value of adopting a framework which dramatically improves a shop's code base cannot be understated. We believe that a framework which includes a mixture of and an appropriate balance between code generation and utility is an optimal modernization strategy.

Choice and Preferences vs. All-In-One

Does IBM i-centric tooling provide for alternative preferences and choices for generating/editing/debugging HTML, CSS, JavaScript? Are developers allowed to have preferences for JavaScript frameworks? Or are developers required to adopt a single all-in-one set of tooling?

Multi-Tenant/Multi-Environment vs. Single

Does the tooling automate and facilitate the generation of separate environments for development, test, and production? And/Or environments for multiple tenants, separate products, separate development teams, separate user-communities?

Summary

Developers and administrators in many IBM i shops have been pre-conditioned to think in terms of face-lifting existing applications instead of a modernization track which actually improves the end-user experience in addition to improving a shop's code base. Fortunately, there are alternatives. LinkedIn comments regarding moving from green screens to browser user interfaces, including feedback to this piece, led me to write [Part 2](#).

IBM i Modernization - The User Interface (Part 2)

<https://rd.radile.com/rdweb/info2/ibmiui2.html>

This piece is a response to comments in LinkedIn discussions [here](#) and [here](#), which included feedback to my first piece titled [IBM i Modernization - The User Interface](#).

GUI Designers

One individual praised the GUI designer of one vendor saying that it was "drag and drop" and "easy". This caught my attention because I had just warned against tooling which enforced fixed positioning and other explicit (hard-coded) property settings pertaining to UI elements. Such tooling typically enforces page-at-a-time paradigms.

My main point was that it's better to allow the position, size, and other properties pertaining to UI elements to adjust automatically to screen sizes and resolutions, and to allow incremental screen updates rather than page-at-a-time updates.

There are other reasons for avoiding such GUI designers. For instance, how "easy" is the drag-and-drop paradigm when you can't find a UI widget in the "palette" which implements the attributes and behaviors which are called for in your design?

I'd like to share a few design patterns which enhance end-user productivity. But they're typically not found in GUI component palettes and page-at-a-time paradigms.

Split Screen (List and Detail)

A common design pattern under the green-screen paradigm enables users to browse lists of items, then optionally select items to show additional details and allow data editing. The "detail" screen is typically separate from the "list" screen due to row-column constraints in display files.

Web browsers aren't subject to display-file constraints. And it's often practical to combine scrollable lists and full-record displays in a single page.

Keyboard shortcuts and mouse clicks may be used to navigate list items. The moment an item is selected, the detail pane is updated. A process that normally requires two screens may be thus streamlined into one. And the toggling of "add", "change", "copy", "delete", "print" modes, and other options may not even require interaction with the server.

Filter Prompts

An extension of the split-screen pattern may provide pop-up prompting to enable users to filter list rows by criteria applied to any column in the data set. And again the pop-up behavior may not require any additional interaction with the server. The popup may be retained in browser cache.

List Rankings

Consider a process where users may optionally select list items, and rank them according to priorities. This may including toggling item-selection to one state or another. It may include automatically changing the rank of selected items when the priority of another is adjusted up or down.

It would be problematic if such interactions required a server to reorder and refresh entire lists when such toggling occurs in the browser. Best to handle this with browser cache and JavaScript.

Content Injection

Regarding the previous section on "list rankings", what if a user needs more information about an item in the list in order to prioritize? Additional details could be dynamically retrieved from the server when a row is clicked and "injected" into that row. There is no limit to row height in HTML tables. Additional clicking could toggle the display or hiding of details.

Graphical Representations of Physical Facilities

A simple example of this might be the configuration and depiction of classroom seating charts. A box which represents a "seat" in a "room" may contain a picture and name of a student assigned to the seat.

The amount of data entry which may be accomplished via graphical representations of physical facilities may surprise those who are steeped in the green-screen paradigm. Clicking on a seat to record attendance, to identify students who plan on participating in school lunch or any other type of activity. Clicking on a seat to activate a prompt for additional data pertaining to the item selected.

Calendars

Not to be confused with data pickers. Calendars can be used for scheduling any number of possibilities, and calendar navigation is intuitive.

Worksheets

Columns may represent data sets from one table; rows may represent data sets from another; intersecting cells may represent a third data set.

Worksheets may enable users to navigate cells with keyboard shortcuts, select blocks of cells, and update values in the most efficient way possible.

Touch-Screen Keypads

Particularly relevant to mobile devices, the user interface may call for a graphical keypad which overrides the device keypad and/or simulates say data entry for ATM or point-of-sale terminals.

Drag and Drop

Ironically the drag-and-drop behavior of GUI designers which may be praised by proponents was probably not implemented by the same. It required thinking (and working) outside of the box, so to speak.

Components Which Update the State of Another

A visual representation of a shopping cart, which is updated by clicking on an item, or by dragging and dropping the item into the cart. A point-of-sale system where items for sale are visually represented, and clicking on one generates a transaction.

Inline Frames

An email client may require inline frames representing multiple data sets such as email recipients, a sent mail list, new mail composition, and other data sets all on one screen.

The point is that the content of one frame may be updated independently of that in other frames. Or a change in one frame may trigger changes to others.

Multi-Step Dialogs

The point might be to cache the prompts in the browser along with entered-data until the aggregate is posted in the final step. Allow navigation back and forth between steps without interaction from the server in the meantime.

Dashboards

The point might be to push data from the server to the user for real-time updates. Or the point might be to facilitate the design of UI widgets which might not be part of preloaded GUI palettes.

Constraints

Preceding sections are intended to stimulate thinking - to help readers become more aware of the types of constraints which might be inherent in web interfaces which rely on GUI designers on the back-end; paired with runtime applets which are downloaded to browsers on the front-end.

Learning HTML, CSS, and JavaScript which enables developers to select from a variety of GUI widgets available in numerous frameworks, and/or design their own. This can be tremendously enabling, and liberating.

Summary

As indicated in [Part 1](#), green-screen modernization may be a defining moment for organizations, where the risk of failure is high, and failure to meet end-user expectations can negatively impact the perception of IBM i and developers of legacy systems.

This piece touched on design patterns which align browser user interfaces with end-user expectations and ensures an improvement over interfaces which extend the green-screen paradigm (which normally don't meet end-user expectations).

GUI designers which include drag-and-drop widget palettes paired with a screen design surface may appear "easy". But they are actually impediments when user interfaces call for UI elements and design patterns which are not included in the palette.

This piece briefly addresses only a few comments which were posted in LinkedIn discussion. This topic may call for a [Part 3](#) (sigh).

IBM i Modernization - The User Interface (Part 3)

<https://rd.radile.com/rdweb/info2/ibmiui3.html>

[Part 1](#) and [Part 2](#) in this series addressed constraints in IBM i-centric tooling which is commonly used to move green screens to browser user interfaces. I highlighted options which ensure an improvement in user experience and the type of infrastructure and tooling which facilitates web application development.

Comments and feedback surfaced during LinkedIn discussions which merit further review. In this piece, I'd like to address the topic of Front-End-Driven Applications (FDAs) and Single-Page Applications (SPAs).

Single-Page Applications

A good definition of single-page applications is found in [Wikipedia](#). It delineates both pros and cons. Key traits include:

- Content fits in a single page.
- Resources are downloaded in connection with an initial page request.
- May dynamically download additional resources which update page content.
- The initial page is never replaced.
- May give the impression of loading additional pages.
- May incrementally update the page in response to UI events.
- May incrementally update the page in response to push events from servers.
- Employs standard browser technologies (HTML, CSS, JavaScript, AJAX, Web Sockets).
- May provide a user experience similar to desktop applications.

Front-End Driven Applications

The key trait of front-end driven applications is that of providing a user experience similar to desktop applications. They may adopt platforms such as Microsoft Silverlight, Adobe Flash, and Adobe Flex in addition to HTML5, etc.

FDA's may be embedded in a single page. But proponents add that the applications tend to be self-contained and developers attempt to reduce or altogether eliminate server dependencies.

FDA and SPA Frameworks

FDA and SPA frameworks often include extensive client-based infrastructure (i.e. model-view-controller or model-view-view-model), large libraries of script-based UI widgets, developer APIs, and associated tooling.

Some vendors of IBM i screen-scraper and open-access tooling try to position their products as SPA frameworks, presumably to generate positive buzz. The emulator (JavaScript applet) which they download to browsers has some SPA traits. But the interfaces actually have substantial differences. JavaScript applets are not really frameworks. Frameworks are not really applets. Frameworks include developer components and programming APIs. Applets implement runtime environments. SPAs are client-centric. The display-file paradigm is server-centric.

FDA and SPA Developers

Developers and proponents of FDAs and SPAs tend to:

- Prioritize and base application design on and around the end-user experience.
- Acquire, develop, and deploy platforms and frameworks which have a similar orientation.



IBM i developers tend to be loyal, traditional, nostalgic, and focused on product domains. FDA and SPA developers tend to be geeky and focused on trending client technologies.

IBM i developers and FDA/SPA developers often differ because the IBM i platform is comprehensive, server-centric, database-centric, multi-user centric, and cohesive.

An SPA developer recently characterized me as living in the old paradigm where the server generates and controls the UI. Those who have read the pieces in this modernization series which precede this one should know - that's not an accurate depiction of me. I advocate for designs which enable browsers to be fully utilized while providing rich end-user experiences. On the other hand, I view myself as an IBM i-centric developer too.

Balance

In my opinion, it's possible with IBM i and browser technologies to strike a nearly ideal balance between:

- Platform-centric design and development.
- Data-centric design and development.
- Domain-centric design and development (object-oriented).
- Client-centric design and development.

There appears to be a tendency for developers to become dogmatic about one of the orientations listed above, even though they all have pros and cons. I find extremes in every community. The platforms and frameworks that support FDAs and SPAs tend to be uber-geeky and overkill. It's better to strike a balance.

The Problem with FDAs and SPAs

Wikipedia article delineates the following issues with SPAs:

- Search engine optimization.
- Browser history.
- Analytics (what and how much is being used).
- Size and performance of initial page load.

Given the context of IBM i application modernization, I would add:

- Not well suited for broadly scoped functionality.
- Not well suited for multi-user data concurrency.
- Popular frameworks tend to be uber-geeky.

In the remaining sections, I'd like to share a few tips in regards to achieving a balance between IBM i-centric technologies and browser technologies.

Multi-User Broadly-Scoped Applications

FDAs and SPAs may be great for running narrowly-scoped applications. But what about applications which entail working with dozens if not hundreds of shared data objects concurrently?

IBM i is exceptionally suited for this type of application. The implementation of such often entails many program calls, many data objects, and many screens within a single session (JOB) as users require options for sharing, navigating, and working with complex relational databases.

A browser-based implementation of a complex application might entail a URL such as the following:
<https://mydomain.com/path/{{session ID}}.ext?appID='at-your-service'>

Each part of the URL is relevant. However, in the context of this treaty - notice the {{session ID}} and the "appID" parameters. The {{session ID}} enables the request to be routed to an individual IBM i



JOB which provides stateful services for only one individual client. And the "applD" parameter enables calls to any number of individual applications.

The interface enables individual browsers to invoke any number of "applications" - all associated with a single session. There's no limit to the number of program calls, open data paths (data objects), or UI content which may be provided by the server.

The interface requires a web portal for launching any number of "sessions" from menu items, some infrastructure for routing requests to the corresponding IBM i servers, and a utility for automatically invoking any number of applications when an "applD" changes.

Individual applications may embody many of the traits of SPAs while the application is in use. But the interface also supports calls to other applications which may output new pages. Thus keeping applications modular, manageable, and easily extendable.

Middle Ground

It's both interesting and troubling to see the pendulum swing back and forth between client-centric and server-centric technologies. I believe that a nearly-optimum balance lies in the middle.

Rather than tasking browsers to download and host large FDA and SPA frameworks (which are essentially platforms themselves), developers can create applications which incrementally download modest amounts of HTML, CSS, and JavaScript, which deliver similar results as SPAs, while using modest amounts of browser cache and browser scripting capabilities.

Summary

The topic of moving from green-screens to browser user interfaces often evokes polarized debate between proponents of platform-centric, data-centric, domain-centric, and client-centric alternatives. FDAs and SPAs are client-centric options.

Much of the dogma surrounding alternative design and deployment options is not only polarizing; it's misleading and unhelpful. In my experience it is possible to achieve a nearly-optimum balance by an appropriate pairing IBM i-centric, data-centric, domain-centric, and client-centric options.

IBM i Modernization - The User Interface (Part 4)

<https://rd.radile.com/rdweb/info2/ibmiui4.html>

Learning browser technologies (i.e. HTML, Style Sheets (CSS), and JavaScript) is cited as a hurdle for traditional IBM i developers. Some vendors promise to bridge the gap by providing tools which don't require developers to learn (HTML, etc.). That can be a disservice.

Knowledge and skills with browser technologies is within the grasp of IBM i developers (speaking from personal experience). And learning them opens the world of web development.

The objective of this piece is to highlight key aspects of browser technologies, to review an example, and provide a basis for an application which IBM i developers should be able to relate to. I plan on writing a "Part 5" later which builds on this foundation, including showing some interaction with an IBM i server.

I'd like people to understand that even basic skills can be highly effective and personally enabling. People who master these skills rarely feel constrained by browser user interfaces. You can accomplish just about any UI design objective. This article will demonstrate some of the key points which I raised in [Parts 1-3](#) in this series.

Learning Resources

The key take-away from this section is that resources for learning, writing, and debugging HTML, style sheets (CSS), and JavaScript are plentiful and readily available. Every learner can find resources suited to their needs.

Find them using browser search engines and asking people you know. Technical forums (i.e. [Midrange Lists](#) can refer you to learning resources).

A favorite resource is [W3Schools](#). Also, code editors which support prompting and auto-completion are handy when learning and writing code.

Web browsers include developer tools which are used for precise pin-pointing (inspecting) the visual appearance (styling) of HTML elements. They enable visual properties to be changed on the fly. Google Chrome has a tool which shows the appearance in various screen sizes and resolutions. JavaScript debugging is included.

HTML Basics

HTML is generally composed of "text", "images", and other "objects" which are enclosed within *beginning* and *ending* "tags". Tags are words and highly-abbreviated character expressions which are prefixed by a less-than sign and end with a greater-than sign (i.e. <div>...</div>).

Tags have default properties which modify the appearance and behavior of whatever they encapsulate. A good HTML tag reference will delineate the purpose, attributes, and event listeners which might pertain to each.

An HTML "page", also known as a "document" begins and ends with <html>...</html>. Enclosed within that beginning and ending, you might include any number of other elements which would be wrapped in their own respective beginning and ending tags.

HTML is relevant to application modernization because it:

- Is a recognized standard.
- Is used ubiquitously.
- Is adhered to in all current browsers.
- Is the most efficient way to render a UI in a browser.
- Offers the best UI performance.

When a browser receives an HTML page, all tags are parsed and used to *instantiate* page elements in a container known as the "document object model (DOM)". The DOM contains objects which generally have a visual representation. DOM objects have parent, child, and sibling relationships which follow the hierarchy delineated in the HTML markup.

It helps a lot for developers to use browser developer tools to inspect the hierarchy and contents of the DOM after pages are rendered (right click on a page to see what tool options might appear). This gives you an inside perspective of what causes the UI to look and behave the way it does. It gives you ideas of how object attributes might be modified and extended in order to change their appearance and behavior.

An effective development technique is to use an HTML editor which assists with code completion and prompting, then preview the page in the browser, then use browser developer tools to inspect element attributes and hierarchical structure.

A basic template for an HTML page might go as follows (notice *beginning and ending* tags):

```
<html>

<head>

  <link href="ibmui.css" rel="stylesheet">

  <title>IBM i Page</title>

</head>

<body>

</body>

<script>

</script>

</html>
```

CSS Basics

In addition to the default appearance of HTML tags, an extensive list of attributes can be applied to HTML elements which further modify their look and feel. The best way to do this is to create an external text file known as a "cascading style sheet", which by convention has a (.css) extension. CSS files contain named objects which in turn contain named attribute-value pairs which define the look and feel of DOM objects.

HTML tags have a corresponding attribute named "class" which can be assigned one or more names which reference one or more objects in one or more style sheets. Over time the default look and feel of HTML tags has become less relevant as CSS standards have been extended, and as browsers have added support for animation and such via common CSS extensions.

Use Case

This section includes a series of dialog panels which prompt users to respond to an "IBM i Marketplace Survey". This is an example of an interactive application embedded in a single HTML page.

It shows what might be done with HTML, CSS, and a small amount of JavaScript. Implementation details will be delineated in subsequent sections. In my next article, I intend on extending this example to include interactions with an IBM i server. Please try it!

Welcome to the IBM i Marketplace Survey

Participating in this survey allows everyone to gauge trends in the IBM i marketplace.

Click the "Start" button to begin. Thank you for your input.

How Does It Work?

The survey consists of a series of HTML `<table>` elements such as:

```
<table class="dialog" cellpadding="10" id="step0">
<tr>
<td height="40" align="center"><h3 class="header">Welcome to the IBM i Marketplace
Survey</h3></td>
</tr>
<tr>
<td>Participating in this survey allows everyone to gauge trends in the IBM i marketplace. Click the
<em>"Start"</em> button to begin. Thank you for your input.</td>
</tr>
<tr>
<td height="40" align="center"><button class="next" onClick="dostep(1)">Start</button></td>
</tr>
</table>
```

Basic formatting is provided by row, cell, and other HTML elements. Extended formatting is provided by "class=" references to style objects which are defined in an [external CSS file](#).

A JavaScript function named `dostep()` is referenced by an "onClick=" event listener which is attached to `<button>` elements.

```
<button class="next" onClick="dostep(-1)">Back</button>
<button class="next" onClick="dostep(1)">Next</button>
```

`<script>` tags are used to reference JavaScript code in HTML pages, including references to external JavaScript files.

```
<script>
var step = 0;
var maxstep = 5;
```



```
function dostep(n) {  
  var o = document.getElementById('step' + step.toString());  
  o.className = 'ph';  
  
  step = step + n;  
  
  if (step < 0 || step > maxstep) step = 0;  
  
  o = document.getElementById('step' + step.toString());  
  o.className = 'dialog';  
}  
</script>
```

dostep() behavior is implemented by retrieving an object reference (o) to dialog panels and toggling their visibility on or off, via class name assignments ("ph" and "dialog").

A Tip

When you view the HTML, CSS, and JavaScript source behind pages at some of your favorite web sites - the complexity or seeming lack of order may be discouraging. A lot of web pages are *overly engineered*. Like any kind of software, the code can end up like a *big ball of mud*. It's important to apply appropriate design principles such as *separation of concerns*. Use external JavaScript files. Use external CSS files. Keep it simple.

Summary

Attractive styling can be applied to HTML elements via "class" names referenced in external style sheets. A single style sheet can be shared by any number of applications, which is a best practice. A relatively small amount of JavaScript can implement interactive behaviors in browsers without having to involve servers. However, as we'll learn in the next article, some background asynchronous communication with an IBM i server can easily extend functionality.

A nearly ideal balance of code and functionality can exist between browsers and IBM i servers by beginning with a set of building blocks such as those introduced here.

IBM i Modernization - The User Interface (Part 5)

<https://rd.radile.com/rdweb/info2/ibmiui5.html>

Say you've been tinkering with HTML, CSS, and JavaScript. Now you'd like to move on to requesting things from an IBM i server. How does that work? The short answer is that browsers send messages to the server which contain URLs (i.e. http://mydomain.com/give_me.ext).

When the IBM i HTTP server receives a message from a browser, it maps the URL to a "resource" which is available (hopefully) on the server. IBM i resources include static files and programs.

Programs may forward requests to web application servers and comparable runtime environments. Before we get into additional details pertaining to resources available on IBM i, we should first understand the types of data (and information) that browsers may **send** along with "requests".

Browsers may send, for example:

- URL query-string parameters (i.e. ?user=Nathan).
- Request Headers.
- HTML form data.
- Cookies.
- Formatted streams (i.e. JSON & XML).
- Attached files (documents, etc.).

Understanding the content which may be included in browser requests is a premise for understanding what may be required of IBM i frameworks and runtime environments in order to enable developers to handle them.

Query String Parameters

Query string parameters are a series of *name=value* pairs which are appended to URLs (i.e. <http://mydomain.com?user=Nathan&environment=development>).

The first *name=value* pair is preceded by a question mark "?". All *name=value* pairs thereafter are preceded by an ampersand symbol "&".

IBM i web application frameworks should include APIs which return the:

- Number of query-string parameters passed.
- Names and values, given their numeric index.
- Value, given the name.

Request Headers

Request headers are *name: value* pairs. Delimiters are slightly different than query-string parameters (colons & line feeds). They provide information which is used by the IBM i Apache based HTTP server. And they can be useful to developers too.

Here's an example of request headers sent from a browser:

Host: rd.radile.com

Connection: keep-alive

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)

Chrome/53.0.2785.143 Safari/537.36



Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: en-US,en;q=0.8
If-None-Match: "1d744-2bbf-53ecae2816500"-gzip
If-Modified-Since: Fri, 14 Oct 2016 03:33:08 GMT

The IBM i based Apache server stores request headers along with various HTTP server configuration settings in JOB environment variables, which can be accessed from programs.

IBM i web application frameworks should include APIs comparable to the ones listed for query-string parameters, except used for returning data and values pertaining to JOB environment variables.

HTML Form Data

HTML <form> input elements are the primary means of entering data in web applications. The IBM i Marketplace Survey, which was presented in [Part 4](#) included a form which contained several input elements. Browsers format this data, again as *name=value* pairs when "forms" are submitted.

IBM i web application frameworks should include APIs comparable to the ones listed for query-string parameters, except used for returning data and values submitted on HTML forms.

Cookies

Standards provide an option for servers to generate small files known as "cookies", which are returned to browsers, and stored on client devices. Servers can set the storage time-frame (i.e. session, whatever).

Browsers return cookies, appended to requests, when the requested URL matches a "path" specified in the cookie.

IBM i web application frameworks should include APIs which can be used for generating cookies, in addition to retrieving values stored in them.

Formatted Streams

Browsers may send data, formatted as [XML](#) or [JSON](#) text streams. This may include hierarchically formatted data objects.

IBM i web application frameworks should include APIs which enable developers to parse and extract XML and JSON formatted data into program variables and data structures, along with APIs for storing such streams in the integrated file system (IFS) if needed.

Attached Files

Browsers are capable clients for document management systems; They can upload files, attached to requests.

IBM i web application frameworks should provide a simple utility, which might be used to store file attachments in the IFS.

What Triggers Browser Requests?

Browser requests may be triggered by:

- Embedding URLs in HTML tags (page loads trigger additional requests).
- Clicking hyperlinks.
- Submitting HTML forms.

- Running JavaScript statements.

Submit HTML Form:

```
<form action="request.shtml">
```

```
...
```

```
</form>
```

```
...
```

```
document.forms[0].submit();
```

Click Hyperlink:

```
<a href="request.shtml">Click Me</a>
```

Assign new page location by using JavaScript:

```
location.href = 'request.shtml';
```

```
location.assign('request.shtml');
```

```
location.replace('request.shtml');
```

```
location.reload(true);
```

Browsers can send asynchronous (background) requests by using its XMLHttpRequest object. I may cover that in a separate article.

Summary

In order for browsers to interact with a server (i.e. IBM i), they send requests, using URLs. Requests may include a variety of data, documents, and information.

Knowing the types of content which may be included in requests, helps evaluate IBM i web application frameworks. This article builds on information provided in [Part 4](#).

IBM i Modernization - The User Interface (Part 6)

<https://rd.radile.com/rdweb/info2/ibmiui6.html>

This piece continues my coverage of browsers and client-side technologies. I'm looking forward to moving on to the IBM i side of the interface. But it may help to cover things which may be less familiar to IBM i developers, first. The material is intended to be progressive. A caveat is that this material may be somewhat advanced for some. I'll address it step by step.

Let's Begin

One aspect that carries over from the green-screen paradigm to browsers is the idea of merging "layout" with "data" to create your stunning masterpiece (that which appears on the screen). I hope you enjoyed the hyperbole.

The layout in this context is an HTML template (as opposed to a display file), which has been enhanced by attaching references to a style sheet. The data, presumably comes from an IBM i database, or some other data source which may be accessible from an IBM i program.

Under the display-file paradigm, and also under the traditional web-application paradigm, the merging of layout and data occurs on the server. That's still a valid paradigm. But it's also possible to perform the merge in the browser, and thus save some network bandwidth along with reducing server CPU cycles.

The "phones" application, which is embedded in the following HTML `<iframe>` element shows an example of merging HTML templates with JSON data objects - where the merge occurs in the browser rather than on the server.

Phones Application

Please navigate within the application by clicking on hyperlinks, images, and using the browser's back and next buttons. The UI consists of two (2) main panels. A phone list. And phone details.

How Does it Work?

Let's first have a look at the initial HTML page. Looks pretty simple, eh?

```
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1" device="mobile">
<title>Phones</title>
<link href="phones.css" rel="stylesheet">
<script src="../../apps/common/scripts/ht_merge.js" defer></script>
</head>
<body>
<h1>Phones</h1>
<a name="phones" ht-container="phones" ht-href="phones.html" ht-jref="phones.json" ht-repeat
ht-class="fade-in" ht-cache></a>
<a name="phone" ht-container="phone" ht-href="phone.html" ht-after="after_phone()" ht-
class="fade-in" ht-cache></a>
</body>
<script src="phonelist.js"></script>
</html>
```

Although a relatively small block of code, the HTML may look a bit cryptic. I'll break it down and explain it.

```
<meta name="viewport" content="width=device-width, initial-scale=1" device="mobile">
```

The <meta> tag causes the page to scale nicely on mobile devices. However, that behavior is somewhat thwarted by my having embedded the page in an <iframe> for this article. It works great when the application URL is evoked from a browser's address bar.

```
<title>Phones</title>
```

The <title> tag makes the word "Phones" appear in the browser's window bar or tab. However, that behavior is also thwarted by my having placed the page in an <iframe>.

```
<link href="phones.css" rel="stylesheet">
```

This application has some new styling which I was playing with, including some fade-in animation, which is referenced in a style sheet ([phones.css](#)).

```
<script src="../../apps/common/scripts/ht_merge.js" defer></script>
```

[ht_merge.js](#) is a relatively small JavaScript framework which merges HTML templates with JSON data objects, in the browser.

The <body> of the page includes a couple HTML elements which need further explanation:

```
<body>
<h1>Phones</h1>
<a name="phones" ht-container="phones" ht-href="phones.html" ht-jref="phones.json" ht-repeat
ht-class="fade-in" ht-cache></a>
<a name="phone" ht-container="phone" ht-href="phone.html" ht-after="after_phone()" ht-
class="fade-in" ht-cache></a>
</body>
```

Please notice the <a> tags. They contain names and a number of non-standard attributes and attribute-value pairs. All attribute names that begin with "ht-" are relevant to our framework which merges HTML templates with JSON data objects. The merged content is "injected" into that container when displayed. We use this same framework in several of our applications.

<a> tags define hyperlinks. But in this case, they link to themselves, which enables the framework to display the content of that element.

Normally the "content" of a hyperlink is just some brief text. But in this case the link serves as a container for complex DOM objects which are dynamically derived from a merger of HTML and JSON data objects - at runtime.

The "phones" container holds a merger of [phones.html](#) and [phones.json](#). Let's take a closer look at the HTML template (phones.html).

```
</a><div class="d">
<a href="#phone" onclick="get_phone('{{id}})">

```

```
<a class="link" href="#phone" onclick="get_phone('{{id}}')">{{name}}  
</a><br><br>{{snippet}}  
</div>
```

Notice the "names" which are delimited by the opening and closing curly braces (i.e. {{id}}, {{name}}, {{imageUrl}}, {{snippet}}).

Now go look at the contents of [phones.json](#). See the same names. Our framework maps JSON data values to their corresponding HTML "markers" during the merge according to their names.

Extract:

```
{  
  "age": 0, "id": "motorola-xoom-with-wi-fi",  
  "imageUrl": "http://angular.github.io/angular-phonecat/step-12/app/img/phones/motorola-xoom-with-wi-fi.0.jpg",  
  "name": "Motorola XOOM\u2122 with Wi-Fi",  
  "snippet": "The Next, Next Generation\r\n\r\nExperience the future with Motorola XOOM with Wi-Fi, the world's first tablet powered by Android 3.0 (Honeycomb)."  
}
```

The "phones" container has an attribute named "ht-repeat" which instructs the merge to repeat for each element in a JSON array. The merge generates a list! This is an example and type of "declarative" programming, as opposed to the typical procedural syntax that we normally use.

The "ht-cache" attribute instructs the framework to cache the HTML template in the browser rather than making additional requests from the server. It's also possible to cache JSON data objects in the browser, or alternatively request a fresh copy from the server when the container is displayed.

A similar merge occurs when a phone is clicked. That causes the phone details panel to be generated and displayed.

The JavaScript

The JavaScript for this application is stored externally in [phonelist.js](#). I'll break it down and explain it.

```
window.location.hash = '#phones';
```

Assigning the name '#phones' to window.location.path is a way to make the phone list appear. It runs when the page is first loaded. Technically "#phones" refers to a DOM object known as an [anchor](#). In this case "#phones" is a reference to the container named "phones". Our framework monitors for window.location.hash changes, then displays that container.

```
function get_phone(id) {  
  ht_merge('phone','phone.html','json/' + id + '.json');  
}
```

The get_phone() function may be invoked while the phone list is displayed, when a user clicks the hyperlink or thumbnail image of that phone. It invokes the framework ht_merge() function which dynamically generates the content in the phone detail container, and shows it.

```
function after_phone() {  
  window.scrollTo(0,0);  
}
```

The `after_phone()` function is invoked when the phone detail container is displayed. It causes the container's content to be scrolled to the top.

```
function after_image() {  
  var o = $('ix').getElementsByTagName('IMG')[0];  
  if (o) $('phone.image').src = o.src;  
}
```

The `after_image()` function is invoked as the phone detail content is being generated. Included in that content is an array of thumbnail images for the selected phone. The "source" of the first array element is assigned to the large image, which is also in the container.

```
function thumb_click(t) {  
  var o = ht_replaceNode($('phone.image'));  
  o.src = t.src;  
  o.className='fade-in';  
}
```

The `thumb_click()` function is invoked when users click on a thumbnail image in the phone detail panel. It replaces the larger image with the source of the thumbnail image.

That's About It

The "phones" application shows some the dynamic capabilities of browser technologies which provide for a rich, fluid end-user experience. It also shows a good balance between client and server roles and interactions. I hope it sparks some interest in browser technologies for traditional IBM i developers.

IBM i Modernization - The User Interface (Part 7)

<https://rd.radile.com/rdweb/info2/ibmiui7.html>

This piece introduces the idea of UI widgets and how they might improve the application experience for end-users. UI widgets combine layout (including styling), content (i.e. text, graphic), event listeners (i.e. mouse and keyboard events) and event handlers (functions which often change the widget's visual appearance).

While styling enhances the visual appearance of page elements, event listeners and handlers implement behaviors which may assist with data entry and navigation.

The HTML specification includes a number of standard UI widgets which are used for data entry (i.e. radio buttons, check boxes, drop-down lists, input elements, push buttons). In my experience, replacing these standard components with alternatives available in UI frameworks has not been helpful to end users. However, styling them, extending them and supplementing them may improve the application experience quite a bit.

UI Widget Frameworks

Settling on and standardizing on a UI widget framework can be difficult due to the prodigious variety in styling, behavior, and programming interfaces available. Technical jargon employed by framework developers and promoters is a problem.

I personally dislike defining widgets via JavaScript language & JSON constructs. That's mostly a matter of taste. I prefer using standard HTML elements. Then enhancing them with CSS styling. That's partly due to the availability of editors which assist with code completion. Also partly due to HTML and CSS being ubiquitous standards.

A library's complexity should be considered. Frameworks introduce constraints in addition to functionality. Many UI frameworks are overly scoped and excessively engineered. Perhaps relevant to end-users might be a framework's size, required downloads, and how much overhead a framework might add to an application. Mobile clients with cellular networks are particularly affected.

For application modernization, I suggest a strategy whereby standard HTML and CSS style sheets are used to define basic widget appearance and behavior. Then use small JavaScript frameworks or modest amounts of custom JavaScript to attach event listeners and handlers which implement prudent (discreet) behavioral enhancements.

Simple Sample

Developers coming from an IBM i green-screen paradigm understand the role of sub-files in display files; They show lists of rows and columns on screens. HTML tables provide similar appearance and functionality.

Moreover, tables can be visually enhanced by attaching styling to their rows, columns, and the table itself. Tables can be wrapped in other containers, which can be styled to look like components (i.e. attach scroll-bars). Finally, event listeners and JavaScript event handlers extend functionality beyond anything available under the green-screen paradigm.

The Phones Application which is embedded below shows how a simple HTML *table* might be transformed into a UI widget which implements behaviors in response to mouse and keyboard events.

Phones



Motorola XOOM™ with Wi-Fi	The Next, Next Generation Experience the future with Motorola XOOM with Wi-Fi, the world's first tablet powered by Android 3.0 (Honeycomb).
MOTOROLA XOOM™	The Next, Next Generation Experience the future with MOTOROLA XOOM, the world's first tablet powered by Android 3.0 (Honeycomb).
MOTOROLA ATRIX™ 4G	MOTOROLA ATRIX 4G the world's most powerful smartphone.
Dell Streak 7	Introducing Dell™ Streak 7. Share photos, videos and movies together. It's small enough to carry around, big enough to gather around.
Samsung Gem™	The Samsung Gem™ brings you everything that you would expect and more from a touch display smart phone – more apps, more features and a more affordable price.
Dell Venue	The Dell Venue; Your Personal Express Lane to Everything
Nexus S	Fast just got faster with Nexus S. A pure Google experience, Nexus S is the first phone to run Gingerbread (Android 2.3), the fastest version of Android yet.
LG Axis	Android Powered, Google Maps Navigation, 5 Customizable Home Screens

Samsung Galaxy Tab™	Feel Free to Tab™. The Samsung Galaxy Tab™ brings you an ultra-mobile entertainment experience through its 7" display, high-power processor and Adobe® Flash® Player compatibility.
Samsung Showcase™ a Galaxy S™ phone	The Samsung Showcase™ delivers a cinema quality experience like you've never seen before. Its innovative 4" touch display technology provides rich picture brilliance, even outdoors
DROID™ 2 Global by Motorola	The first smartphone with a 1.2 GHz processor and global capabilities.
DROID™ Pro by Motorola	The next generation of DOES.
MOTOROLA BRAVO™ with MOTOBLUR™	An experience to cheer about.
Motorola DEFY™ with MOTOBLUR™	Are you ready for everything life throws your way?
T-Mobile myTouch 4G	The T-Mobile myTouch 4G is a premium smartphone designed to deliver blazing fast 4G speeds so that you can video chat from practically anywhere, with or without Wi-Fi.
Samsung Mesmerize™ a Galaxy S™ phone	The Samsung Mesmerize™ delivers a cinema quality experience like you've never seen before. Its innovative 4" touch display technology provides rich picture brilliance, even outdoors
SANYO ZIO	The Sanyo Zio by Kyocera is an Android smartphone with a combination of ultra-sleek styling, strong performance and unprecedented value.
Samsung Transform™	The Samsung Transform™ brings you a fun way to customize your Android powered touch screen phone to just the way you like it through your favorite themed "Sprint ID Service Pack".
T-Mobile G2	The T-Mobile G2 with Google is the first smartphone built for 4G speeds on T-Mobile's new network. Get the information you need, faster than you ever thought possible.
Motorola CHARM™ with MOTOBLUR™	Motorola CHARM fits easily in your pocket or palm. Includes MOTOBLUR service.

An ordinary HTML table has been transformed into a UI widget. It has been enhanced by alternating background colors for odd and even rows. The current row is highlighted. Mouse clicks, up-arrow, and down-arrow keys may be used to select rows. The current row is automatically scrolled into view if necessary.

The phone image is changed when a new row is selected. So the table and image can jointly be viewed as being bound together - like a multi-component widget.

Key-press Behaviors

Let's review the key-press behaviors which have been attached to the table:

- Up-Arrow (navigate to the prior row).

- Down-Arrow (navigate to the next row).
- End-Key (navigate to the last row).
- Home-Key (navigate to the first row).
- Shift-Down-Arrow (select current and next rows).
- Shift-Up-Arrow (select current and prior rows).
- Ctrl-End-Key (select all rows from current to the end).
- Ctrl-Home-Key (select all rows from current to the first).
- Ctrl-A (select all rows).

Mouse-Click Behaviors

Let's review the mouse-click behaviors which have been attached to the table:

- Click (select a row).
- Ctrl-Click (multi-select individual rows).
- Shift-Click (multi-select a continuous range of rows).

How Does It Work?

Let's begin by reviewing the HTML.

```
<h1 id="phead">Phones</h1>
<p style="height:400"><img id="imgx"></p>
<div class="content" style="box-shadow:4px 4px 2px #718BA4;">
  <table id="tb" ht-container="phones"
    ht-href="../../phones/phonerow.html"
    ht-jref="../../phones/phones.json"
    ht-repeat ht-cache ht-after="after_phones()"
    width="100%" height="100%" cellpadding="6" cellspacing="0">
  </table>
</div>
```

Standard HTML and CSS elements are used to define layout and basic behavior. However this application uses a framework which dynamically generates table rows and cells at runtime by merging an HTML template [phonerow.html](#) and a JSON array [phones.json](#).

The framework which merges HTML and JSON was introduced in [Part 6](#). Let's review it again here by beginning with the following attributes assigned to the *table*.

- ht-container="phones" (names a container into which merged content is placed).
- ht-href="../../phones/phonerow.html" (refers to an HTML template).
- ht-jref="../../phones/phones.json". (refers to a JSON data object).
- ht-repeat (instructs framework to add a row for each item in a JSON array).
- ht-cache (instructs framework to cache the HTML template in the browser).
- ht-after="after_phones()" (instructs the framework to invoke after_phones() after completing the merge).

Now let's review the the custom JavaScript pertaining to the application ([phonetbl.js](#)).

```
function after_phones() {
  tb = new Table('tb');
  tb.inzTable();
}
```

The `after_phones()` function is invoked after completing the HTML-JSON merge which generates the table's rows and cells.

`tb = new Table('tb')` creates a new `Table()` object and returns a reference to it. The table's id (i.e. 'tb') is used to specifically identify the table.

`tb.inzTable()` initializes attributes pertaining to each row in the table (i.e. background color, event listeners, etc.). That's all that's needed to transform a standard HTML table into a widget which implements relevant visual attributes and behavior.

```
function _selectRow(row) {  
  var i = row.getAttribute('ht-index');  
  $('imgx').src = ht_containers[0].json[i].imageUrl;  
  $('phead').scrollIntoView(true);  
}
```

The `_selectRow()` function is automatically invoked by the `Table()` object whenever users select a new row. I'm using it to change the image URL, which is referenced in the JSON array object.

Framework Components

The framework supporting this interface is referenced via the following `<script>` tags in the page's header section:

```
<script src="../../rdweb/apps/common/scripts/binders.js"></script>  
<script src="../../rdweb/apps/common/scripts/keyhandler.js"></script>  
<script src="../../rdweb/apps/common/scripts/tables.js"></script>  
<script src="../../rdweb/apps/common/scripts/ht_merge.js" defer></script>
```

The framework size sums to something like 40KB (un-compressed), which is tiny in comparison to typical frameworks which include UI widgets and client-side infrastructure.

Much can be accomplished by a relatively small amount of JavaScript without burdening bandwidth and client-side compute resources.

Conclusions

1. Widgets can improve the appearance and behavior of the user interface.
2. Widgets can improve end-user productivity.
3. Widget libraries tend to be overly scoped and sized.
4. UI frameworks tend to be overly scoped, overly sized, and excessively engineered.
5. Standard HTML elements offer a solid UI foundation.
6. Supplemental widgets can be added via small frameworks & custom code.

IBM i Modernization - The User Interface (Part 8)

<https://rd.radile.com/rdweb/info2/ibmiui8.html>

In the year 2005, a Web designer named Jesse James Garrett wrote an [essay](#) which delineated an emerging web-page/application methodology. It was based on a set of browser technologies which he collectively assigned an acronym to - AJAX - short for Asynchronous JavaScript and XML.

Mr. Garrett suggested that these technologies represented a fundamental shift in the way that web applications would be designed. That was an understatement. The paradigm shift became known as Web 2.0.

Asynchronous request processing paired with partial updates to page elements is the subject of this piece.

XMLHttpRequest Object

Browsers implement a software component named XMLHttpRequest which provides an interface for making requests asynchronously. This means that requests may execute in background threads which allow users to continue viewing or using the page without disruption.

The typical API for making asynchronous requests also entails defining and coding JavaScript functions which are invoked at the completion of the request-response cycle. Call-back functions typically update the browser's document object model (DOM) based on data objects (XML or JSON) received in responses.

Updating a browser's DOM is a way of dynamically adding or changing page content. It provides a streamlined and fluid user interface.

A Slight Derivative

I'd like to introduce an API which uses the browsers XMLHttpRequest object under the covers. With it, requests run asynchronously in browser threads.

However requests are queued. So developers can count on requests completing in the sequence that they were requested. The next request in the queue is automatically released when the response to the previous one completes.

This enables developers to accurately predict the results of firing off many requests at the same time. Without going into details at this time, this has proven to be very useful for developers. Another feature of the API is that it doesn't entail specifying call-back routines. Instead, IBM i applications get to decide which (if any) JavaScript functions may be called in response to requests. For example, an IBM i application may choose to evoke a JavaScript error handling routine instead of one used to modify page content.

Rather than returning XML or JSON data objects via this API, IBM i applications return JavaScript expressions which are run in response. This has proven to be an effective means of reducing the size of data streams returned to browsers and improving UI performance. It's snappy!

This is not to suggest that other XMLHttpRequest wrappers are inferior. It is only to suggest an additional tool in the toolkit, based on a lot of application experience.

OO JavaScript

The other idea I'd like to briefly touch on is object-oriented JavaScript. This is NOT to suggest that application developers need to become proficient in OO JavaScript. In fact, frameworks which rely on developers mastering OO JavaScript skills have taken a wrong turn, in my opinion.

OO JavaScript makes more sense for framework developers than application developers. But application developers should at least be familiar with creating instances of objects by using the *new* operator. Maybe we can pick up the philosophical discussion about coding styles in a different piece. We use the *new* operator in many applications to instantiate UI widgets, as I will explain with the following embedded application.

Simple Sample

This application is intended to illustrate:

- Asynchronous request processing (ARP).
- A handy ARP API.
- The JavaScript *new* operator.
- UI widgets.
- Dynamic updates to page (DOM) elements.
-

Names

[Click Here](#)

[Click Here](#)

[Click Here](#)

[Click Here](#)

Key Points

Clicking on a table heading sends an asynchronous request to an IBM i server which downloads 500 names and appends them as table rows; Or alternatively clears all rows in the table.

Some web designers suggest that dynamic DOM (page) updates may perform poorly. This example shows that fluid and well-performing updates are possible, even when the changes are substantial in size and number of elements.

Clicking on table rows and pressing navigational keys (i.e. Arrow-Down, Arrow-Up, End, Home) shows that each table (UI widget) encapsulates its own event listeners and event handlers. That's rather object oriented.

Occasionally I hear of green-screen developers asking how to make sub-files appear side by side on a screen. The answers are never satisfactory. This example shows side by side tables when screen sizes permit. However the tables flow from top to bottom on cell phone screens. This is an example of "responsive design"; The layout automatically adjusts to screen width.

How Does It Work?

The HTML which encapsulates each table consists of an element hierarchy. Each element is styled via CSS class names.

An onClick() event listener is attached to table headings. Each table has a unique ID (i.e. "tb1", "tb2", "tb3", "tb4").

```
<div class="oc">
```

```
<center class="head" onclick="names('tb1')">Click Here</center>
```

```
<div class="ic">
```

```
<table id="tb1" width="100%" height="100%" cellpadding="6" cellspacing="0"></table>
```

```
</div>
```

```
</div>
```

A Table() object is created for each HTML table element and object references are stored in document variables.

```
var tb1 = new Table('tb1');
var tb2 = new Table('tb2');
var tb3 = new Table('tb3');
var tb4 = new Table('tb4');
var o = null;
```

The names() function is invoked when users click on a table heading.

```
function names(s) {
    $('hname').innerHTML = 'Names';
    o = eval(s);
    if (o.rows.length == 0) reqGet('ibmiui8.txt');
    else o.clearRows();
}
```

The reqGet() function is invoked if the table is empty. This sends an asynchronous request to an IBM i server for a URL ([ibmiui8.txt](#)).

All rows in the table are alternatively removed if the table has rows. The application heading is updated (shows "Names").

The ar() function is automatically invoked for each "name" when the "response" is received from the IBM i server.

```
function ar(name) {
    o.insertRowBottom('<tr key=1><td>' + name + '</td></tr>');
}
```

The ar() function simply appends the person's "name" as a row in the table. That delineates how the tables may be updated dynamically in response to "Click Me" events. The _selectRow() function is invoked when new rows are selected via mouse click or key-press events. This changes the application heading to show the currently selected "name".

```
function _selectRow(row) {
    $('hname').innerHTML = row.innerHTML;
    $('hname').scrollIntoView();
}
```


}

This application draws from a small JavaScript framework which is referenced from this page as follows:

```
<script src="../../rdweb/apps/common/scripts/binders.js"></script>  
<script src="../../rdweb/apps/common/scripts/keyhandler.js"></script>  
<script src="../../rdweb/apps/common/scripts/tables.js"></script>  
<script src="../../rdweb/apps/common/scripts/xmlhttp.js"></script>
```

The custom JavaScript for this application can be found in file [ibmiui8.js](#).

Summary

An application's user interface can be improved by moving away from page-at-a-time and green-screen paradigms; Moving toward a paradigm which is facilitated by asynchronous request processing. This includes the use of UI widgets and dynamic page updates which provide fluid screen transformations. It also includes responsive layouts which automatically adapt to various screen sizes.

These techniques can be supported by small JavaScript frameworks combined with modest amounts of custom JavaScript.

IBM i Modernization - The User Interface (Part 9)

<https://rd.radile.com/rdweb/info2/ibmiui9.html>

This article delves into the creation of web user interfaces in which the layout automatically adapts to a variety of device types and screen sizes (i.e. cell phones, tablets, laptops, and desktops).

This topic is commonly referred to as "responsive design". A vigorous ecosystem has erupted, including dozens of popular CSS and JavaScript frameworks which focus on addressing this concern.

The motivation should be obvious - the resources, requirements, and cost of creating and maintaining separate web sites and applications for separate device types is generally prohibitive. Web browsing via cell phones and tablets now exceeds that of laptop and desktop systems.

IBM i modernization tools which extend the green-screen paradigm almost always fail in regards to automatically adapting page layout in response to client device sizes.

This piece builds on articles which [precede](#) it.

Responsive Design and Frameworks

Learning responsive design and adapting allied frameworks to database applications can be a challenge:

- The variety and volume of information and resources available on the internet is enormous (a blessing and a curse).
- Much of the information is promotional.
- Available information and means are typically geared to the requirements of news, text, media, and brochure-style content rather than database applications.
- The fitness of much of the information is poor (too many examples to list - as the user experience often deteriorates measurably between different device types).
- Promulgated ideas and frameworks tend to be ill-suited for database applications. They also tend to be overly-sized, overly-scoped, and complex.

Requirements of Database Management Applications

Database inquiry and maintenance by and large is about working with tables (lists of rows), individual records, data-entry forms and widgets.

I'd like to share a number of traits which should be "responsive" to diverse device types and screen sizes. Layout should also automatically adapt to portrait and landscape viewing.

In regards to viewing nearly **ALL** content on small devices:

- UI component and character sizes should be readable (not tiny).
- Users should be able to (but not required to) magnify.

In regards to reading fluid text via small devices:

- Users should not be required to pan (or scroll) from side to side in order to see it.
- Text should wrap at screen width and flow from top to bottom.
- Line length should adapt to screen width.

In regards to viewing tabular data:

- Column headings should remain fixed while rows should scroll (or pan vertically) beneath them.

- Some frameworks truncate (remove) right-hand columns on small devices. However, in my opinion, the value of viewing cohesive tables generally exceeds the value of fitting tables to available width; It's okay for users to pan left to right in order to view fixed-width columnar data.
- However, users should not be required to pan to see the contents of individual columns - column data should line-wrap vertically if necessary to fit screen width.

In regards to full-record viewing and data entry forms:

- UI components by default should automatically flow left to right (or visa versa) and vertically in order to fit within screen width.
- When field labels are paired with input elements, they should remain together as components flow horizontally and vertically.
- Push buttons (i.e. for submitting data) should be sized appropriately for cell phones, as well as desktop monitors (users may have fat fingers ;-).
- Images and other UI widgets (i.e. text areas) by default should automatically resize in order to fit within screen width and height.

Form Examples

The following images depict how form components might automatically flow depending on whether viewed on an iPad, or an iPhone 4.

iPad landscape:

District	LPA	School	NORTH
Department	FINE ARTS	Course	DAN12
Extension		Long Title	Advanced Dance
Short Title	Advanced Dance	From Term	0
To Term	0	Terms To Complete	4
From Term	0	Credits Earned	1.00000
Part Credit Terms	0	Grade Point Weight	.00000
Min. Grade Level	7	Max. Grade Level	9
Min. Age	.00000	Max. Age	.00000
Genders Allowed		Approval Required?	<input type="checkbox"/>
Pass % (Minimum)	60.0000	Grade Scale	SECONDARY
X-Ref ID	02020000024	Course Not Offered	<input type="checkbox"/>
		Resource Link	0000001193

iPhone 4 portrait:

District	LPA
School	NORTH
Department	FINE ARTS
Course	DAN12
Extension	
Long Title	Advanced Dance
Short Title	Advanced Dance
From Term	0

Note that the iPad landscape view separates field labels and input elements into 4 columns horizontally, while the iPhone portrait view separates field labels and input elements vertically within a single column; users pan vertically to see additional input elements.

On a wide-screen monitor, the layout may shift and expand horizontally into 6 columns, as opposed to 4 columns on the iPad.

Note that the font sizes appear the same, no matter the screen size or resolution (pixels per inch).

Table Examples

The following images depict how table components might automatically flow depending on whether viewed on an iPad, or an iPhone 4.

iPad landscape:

List	North Campus Director									
District	School	Dept	Course	Ext	From Term	To Term	Title	Credits	Min Level	Max Level
LPA	NORTH	ADMIN	ADM4		0	0	North Campus Director	.00000	0	9
LPA	NORTH	ADMIN	ADM5		0	0	North ADA	.00000	0	9
LPA	NORTH	ADMIN	ADM7		0	0	Counselor 7-12	.00000	0	9
LPA	NORTH	ADMIN	ADM8		0	0	Counselor K-6	.00000	0	9
LPA	NORTH	ADMIN	AMD3		0	0	Special Education	.00000	0	9
LPA	NORTH	BUSINESS	BUS02		0	0	Computer Technology 1	.50000	9	9
LPA	NORTH	BUSINESS	BUS05		0	0	Computer Keyboarding	.50000	6	12
LPA	NORTH	COUNSELING	SCH01		0	0	See Counselor for Schedule	.00000	7	9
LPA	NORTH	CTE	CTE01		0	0	Introduction to CTE	1.00000	6	8
LPA	NORTH	CTE	CTE02		0	0	College & Career Awareness	1.00000	7	7
LPA	NORTH	CTE	CTE03		0	0	Exploring Technology	.50000	8	9
LPA	NORTH	CTE	CTE04		0	0	Digital Literacy	.50000	8	8
LPA	NORTH	ELECTIVE	ELC01		0	0	Debate 1	.50000	8	9
LPA	NORTH	ELECTIVE	ELC02		0	0	Debate 2	1.00000	8	9
LPA	NORTH	ELECTIVE	ELC05		0	0	Released Time (Seminary)	.00000	9	12
LPA	NORTH	ELECTIVE	ELC08		0	0	Office Aide	.50000	8	12
LPA	NORTH	ELECTIVE	ELC09		0	0	Teacher Assistant	1.00000	8	12

iPhone 4 portrait:

List	North Campus Di									
District	School	Dept	C							
LPA	NORTH	ADMIN	A							
LPA	NORTH	ADMIN	A							
LPA	NORTH	ADMIN	A							
LPA	NORTH	ADMIN	A							
LPA	NORTH	ADMIN	A							
LPA	NORTH	BUSINESS	B							
LPA	NORTH	BUSINESS	B							
LPA	NORTH	COUNSELING	S							
LPA	NORTH	CTE	C							
LPA	NORTH	CTE	C							

iPhone 4 landscape:

List		Detail		North Campus Director	
District	School	Dept	Course	Ext	From To
LPA	NORTH	ADMIN	ADM4		(
LPA	NORTH	ADMIN	ADM5		(
LPA	NORTH	ADMIN	ADM7		(
LPA	NORTH	ADMIN	ADM8		(
LPA	NORTH	ADMIN	AMD3		(
LPA	NORTH	BUSINESS	BUS02		(

Note that although tables may be truncated on the right side due to small screen size, headings are fixed while rows may be scrolled, so row and column contexts are preserved.

Users may optionally reduce magnify, and view all columns on small screens when desired. But the default look and feel should accommodate readability and usability (i.e. selecting a row with a finger tap, and panning vertically with finger swipes).

What to Look For Next

Responsive traits (how page layouts adjust automatically according to device type and screen size) is determined by HTML element groupings, CSS styling, and possibly a small amount of JavaScript. Some web designers accomplish this with frameworks.

In the next article, I intend to demonstrate and review the application shown in screen shots in this piece, including an explanation behind the HTML, CSS, and JavaScript code.

IBM i Modernization - The User Interface (Part 10)

<https://rd.radile.com/rdweb/info2/ibmiui10.html>

This piece continues the topic of web user interfaces in which the layout automatically adjusts to a variety of device types and screen sizes (i.e. cell phones, tablets, laptops, and desktops) and screen orientations (portrait and landscape).

[Part 9](#) showed a set of screen shots of an application wherein the layout automatically adapts to desktop, laptop, tablet, and cell phone screens.

Click on the following image to demonstrate the application in a new tab or window. Use arrowUp and arrowDown keys (if available) to navigate the list. I'd like to review the pertinent HTML, CSS, and JavaScript after you've had an opportunity try the UI.

List	Detail	North Campus Director								
District	School	Dept	Course	Ext	From Term	To Term	Title	Credits	Min Level	Max Level
LPA	NORTH	ADMIN	ADM4		0	0	North Campus Director	.00000	0	9
LPA	NORTH	ADMIN	ADM5		0	0	North ADA	.00000	0	9
LPA	NORTH	ADMIN	ADM7		0	0	Counselor 7-12	.00000	0	9
LPA	NORTH	ADMIN	ADM8		0	0	Counselor K-6	.00000	0	9
LPA	NORTH	ADMIN	AMD3		0	0	Special Education	.00000	0	9
LPA	NORTH	BUSINESS	BUS02		0	0	Computer Technology 1	.50000	9	9
LPA	NORTH	BUSINESS	BUS05		0	0	Computer Keyboarding	.50000	6	12
LPA	NORTH	COUNSELING	SCH01		0	0	See Counselor for Schedule	.00000	7	9
LPA	NORTH	CTE	CTE01		0	0	Introduction to CTE	1.00000	6	8
LPA	NORTH	CTE	CTE02		0	0	College & Career Awareness	1.00000	7	7
LPA	NORTH	CTE	CTE03		0	0	Exploring Technology	.50000	8	9
LPA	NORTH	CTE	CTE04		0	0	Digital Literacy	.50000	8	8
LPA	NORTH	ELECTIVE	ELC01		0	0	Debate 1	.50000	8	9
LPA	NORTH	ELECTIVE	ELC02		0	0	Debate 2	1.00000	8	9
LPA	NORTH	ELECTIVE	ELC05		0	0	Released Time (Seminary)	.00000	9	12
LPA	NORTH	ELECTIVE	ELC08		0	0	Office Aide	.50000	8	12
LPA	NORTH	ELECTIVE	ELC09		0	0	Teacher Assistant	1.00000	8	12
LPA	NORTH	ELECTIVE	ELC10		0	0	Video Production	.50000	6	12

Meta Tag

The first tip in designing pages which are "responsive" to various screen sizes and device types is to place the following meta tag in the page's *heading* block. It instructs browsers to automatically scale the size of content on cell phones and tablets so that it appears comparable to that on desktop monitors.

```
<meta name="viewport" content="width=device-width, initial-scale=1" device="mobile">
```

Cell phones and tablets have higher pixel densities per inch which traditionally make text appear small in comparison to desktop monitors. Prior to the availability of this meta tag, web designers



would use CSS scaling in order to make text more readable without magnification on small devices. This meta tag solves that problem without the use of CSS hacks.

The Body

The body of the HTML page is as follows:

```
<body onResize="win_resize()">
<div class="app">
<div class="tabs" id="tabs">
<a id="t1" href="#list" onClick="sel_tab(1)" style="width:50">List</a>
<a id="t2" href="#detail" onClick="sel_tab(2)" style="width:50">Detail</a>
<span id="titll" style="padding-left:20; width:190; word-wrap:break-word; display:inline-
block"></span>
</div>
<div ht-container="list" ht-href="ibmiui10a.html" ht-jref="empty.json" ht-cache ht-
after="after_list()" style="max-width:1040" id="dl"></div>
<div ht-container="detail" ht-href="ibmiui10c.html" ht-cache ht-after="after_detail()" style="max-
width:1040"></div>
</div>
</body>
```

The body consists of the following parts:

- a `<div>` container for the application.
- a `<div>` container for tabs.
- 2 `<a>` tags are used for "List" and "Detail" tabs.
- a `` tag is used to display course descriptions as users navigate the list.
- a `<div>` container for the list content.
- a `<div>` container for the detail content.

The `win_resize()` handler is triggered when the browser window is resized. It runs code which adjusts the height of list and detail containers in order to support scrolling underneath tabs and table headings.

This application uses a utility which "merges" additional HTML and JSON content which is requested from an IBM i server at runtime in order to populate the List and Detail containers. That utility was delineated in [Parts 6-8](#).

Table Headings

The following HTML template [ibmiui10a.html](#) is merged into the List container at runtime and shows the layout of column headings.

District	School	Dept	Course	Ext	From Term	To Term	Title
----------	--------	------	--------	-----	-----------	---------------------	-------


```
<td width="80" align="center">Credits</td>
<td width="70" align="center">Min&nbsp;Level</td>
<td width="70" align="center">Max&nbsp;Level</td>
</tr>
</table>
<div id="dr" class="scroll" style="width:1058">
<table ht-container="courses" ht-href="ibmiui10b.html" ht-jref="ibmiui10.json" ht-repeat ht-cache
ht-after="after_rows()" width="1040" id="tb" cellpadding="0" cellspacing="0" border="0">
</table>
</div>
```

Table Rows

The following HTML template [ibmiui10b.html](#) is merged with [ibmiui10.json](#) to populate the table rows.

```
<tr key="">
<td width="80" class="grid">{{DISTRICT}}</td>
<td width="80">{{SCHOOL}}</td>
<td width="150">{{DEPT}}</td>
<td width="80">{{COURSE}}</td>
<td width="60" align="center">{{EXTENSION}}</td>
<td width="60" align="center">{{TERMFROM}}</td>
<td width="60" align="center">{{TERMTO}}</td>
<td width="250">{{TITLL}}</td>
<td width="80" align="center">{{CREDITS}}</td>
<td width="70" align="center">{{MINLEVEL}}</td>
<td width="70" align="center">{{MAXLEVEL}}</td>
</tr>
```

Detail

The detail container is populated at runtime by merging [ibmiui10c.html](#) with the selected JSON array item when the Detail tab is selected.

Let's review the following HTML excerpt.

```
<div class="il fl">
<div class="il w2 label">Long Title</div>
<div class="il w2"><input disabled name="TITLL" value="{{TITLL}}" size="50"></div>
</div>
```

Up to this point I haven't explained much about how to design pages which are adaptable (responsive) to various screen sizes and device types. Note that field labels and input elements are grouped together and surrounded by <div> containers which have relevant CSS class names.

CSS Tips

Let's review the properties of the "il", "fl", and "w2" CSS class names.

```
.il {
display:inline-block;
}
```

```
.fl {
float:left;
}
.w2 {
width:180;
}
```

The "li" class overrides the default way that <div> containers are displayed and allows them to flow horizontally across the page.

The "fl" class allows elements to float from left to right until a "width" limit is reached - then wrap underneath and back to the left.

The "w2" class specifies a width of 180 pixels. Since it applies to <div> tags which encapsulate field labels, that makes all field labels occupy the same amount of width which creates a columnar (or grid) effect on wide screens, and places labels above input elements on narrow screens (i.e. cell phone portrait viewing).

Grid Systems

Popular responsive CSS frameworks implement a concept which is often referred to as a "grid system", entailing from hundreds to thousands of lines of code for CSS class definitions, and JavaScript. They often entail separate sets of CSS class names for extra-small, small, medium, and wide screens.

In my opinion, page designers can achieve better "responsive" effects, more simply by using a combination of say "il", "fl", and "w(n)" class definitions, where the (n) represents a number from say 1-12, and specifies widths which range from say 90-1080 pixels.

With grid systems, web page designers are often fighting with complexities and unknowns within the frameworks.

Container Height and Vertical Scrolling

When tabs and tables are employed in database maintenance applications, usability is enhanced when tabs and table columns remain fixed while other content (i.e. rows, data-entry forms) scroll vertically underneath. This can be achieved with a small amount of CSS and JavaScript.

The following CSS class enables scrolling within <div> containers.

```
div.scroll {
overflow:auto;
-webkit-overflow-scrolling: touch;
-webkit-transform: translateZ(0);
}
```

Which works when paired with a JavaScript routine which adjust the height of scrollable <div> containers when users expand or shrink browser windows.

```
function win_resize() {
  dl.style.height = dl.parentElement.clientHeight - dl.offsetTop - 48;
  dr.style.height = dl.style.height;
  if (dd) dd.style.height = dl.parentElement.clientHeight - dl.offsetTop;
}
```

U.S. Presidents

I created the course-list application in order to provide a basis for sharing tips about responsive design. The application lacks a lot of features in its present condition. It could be functionally expanded in order to provide a basis for basic database maintenance.

After the template was set and vetted, it only took a few minutes to adapt it to another JSON array (U.S. Presidents):

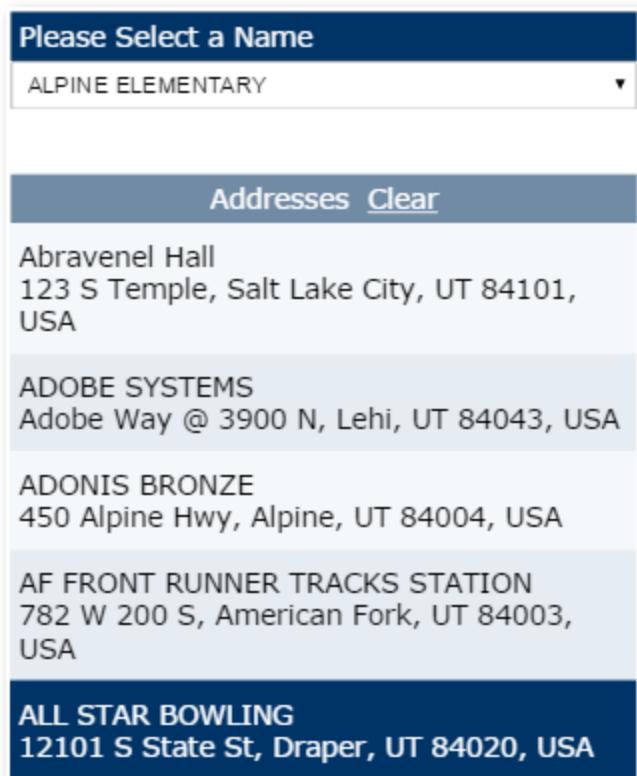
List	Detail	George Washington	
Name	Years Alive	Years in Office	Party
George Washington	1732/1799	1789-04-30 to 1797-03-04	No Party
John Adams	1735/1826	1797-03-04 to 1801-03-04	Federalist
Thomas Jefferson	1743/1826	1801-03-04 to 1809-03-04	Democratic-Republican
James Madison	1751/1836	1809-03-04 to 1817-03-04	Democratic-Republican
James Monroe	1758/1831	1817-03-04 to 1825-03-04	Democratic-Republican
John Quincy Adams	1767/1848	1825-03-04 to 1829-03-04	Democratic-Republican
Andrew Jackson	1767/1845	1829-03-04 to 1837-03-04	Democratic
Martin Van Buren	1782/1862	1837-03-04 to 1841-03-04	Democratic
William Henry Harrison	1773/1841	1841-03-04 to 1841-03-04	Whig
John Tyler	1790/1862	1841-04-04 to 1845-03-04	Whig
James K. Polk	1795/1849	1845-03-04 to 1849-03-04	Democratic
Zachary Taylor	1784/1850	1849-03-04 to 1850-07-09	Whig
Millard Fillmore	1800/1874	1850-07-09 to 1853-03-04	Whig
Franklin Pierce	1804/1869	1853-03-04 to 1857-03-04	Democratic
James Buchanan	1791/1868	1857-03-04 to 1861-03-04	Democratic
Abraham Lincoln	1809/1865	1861-03-04 to 1865-04-15	Republican
Andrew Johnson	1808/1875	1865-04-15 to 1869-03-04	Democratic
Ulysses S. Grant	1822/1885	1869-03-04 to 1877-03-04	Republican

In a future article I intend on addressing how application templates can be transformed into what I like to call "application models", which trivializes the amount of custom coding required, and which can be applied to all kinds of database entities.

IBM i Modernization - The User Interface (Part 11)

<https://rd.radile.com/rdweb/info2/ibmiui11.html>

This piece shows and explains a simple application which delves into the shift from the traditional display-file paradigm to the client-server paradigm, and delineates benefits for end-users. Click on the following screen shot to try the application in a new tab or window.



This application presents a drop-down list of named entities. When you select one, the entity's address is retrieved from an IBM i database and is inserted into the table located just below the entity drop-down list.

Feel free to look up as many addresses as you want (hint: press and hold down the rightArrow and leftArrow keys to navigate the entity list and to stress test performance).

How Does It Work?

In this piece we get to see some ILE RPG code which responds to a request (for an address). But first, I should point out that the application's entry-point is a *static HTML page* which is simply served by the IBM i HTTP server.

You're not forced to "call" an application in order to see the initial page. This is more efficient. The initial page remains cached in the browser until users clear history. It's a good approach for applications which don't require authentication and authorization.

The body of the page is as follows:

```
<body onResize="win_resize()">
<div class="hi-row w4 pd-5">Please Select a Name</div>
<select ht-container="locations" ht-jref="locations.json" ht-after="after_options()" id="ss"
onChange="select_option(this)" class="pd-5" style="width:370"></select>
```

```
<br><br><br>
<div class="bg-1 w4 pd-5"><center>Addresses&nbsp;&nbsp;<a href="javascript:clear_list()"
class="bg-1">Clear</a></center></div>
<div id="tc" class="scroll" style="height:270; width:388;">
<table id="tb" cellpadding="0" cellspacing="0" border="0"></table>
</div>
</body>
```

The body consists of the following parts:

- a <div> for the prompt "Please Select a Name".
- a <select> for the drop-down list.
- a <div> for the table heading "Addresses".
- a <div> for enabling scrolling the address list.
- a <table> for listing addresses returned by the IBM i server.

Populating the drop-down list:

```
<select ht-container="locations" ht-jref="locations.json" ht-after="after_options()" id="ss"
onChange="select_option(this)" class="pd-5" style="width:370"></select>
```

As with examples in previous articles, I'm using a generic utility to automatically generate drop-down options from a *static JSON array* ([locations.json](#)) at runtime. Just specify the URL for the JSON Array using the ht-jref attribute.

Say you have a list of values which don't change often (i.e. the names of incorporated cities with a state). You shouldn't have to generate the list from a database each time the page is requested. Just generate the JSON file beforehand. The data is automatically incorporated into the app without writing server-side code.

The point is that developers may include static content on pages and not have to write (or run) code to dynamically generate it. Social media sites operate on this principle. The amount of static content may not be trivial - there are 600+ options in the drop-down list in this example. And you want it to load quickly.

Requesting an address:

The drop-down list has an event listener (onChange="select_option()") which triggers a request to the server when an option is selected. The select_option() function is defined as follows:

```
function select_option(o) {
  reqGet('/rdcaller/get_address.shtml?rwappid=IUI100&tloc100k=' + o.value);
}
```

reqGet() sends an a request to the server asking for "get_address" from an IBM i program named "IUI100", and passes a query-string parameter named "tloc100k" which is assigned the "value" which was selected from the combo box. That value is the key to the location (address).

What is Happening on the Server?

An IBM i Apache based HTTP server thread receives the request and invokes a plug-in which forwards the request to an IBM i program which then "calls" the program which was named in the URL (rwappid=IUI100; a program named IUI100 is called in this case).



Before we review the IUI100 program source code, I'd like to make a point which is intuitive to most web application developers, but traditional RPG programmers may struggle with. Web applications implement a request-response cycle; specifically ILE programs:

1. Receive a request.
2. Process it.
3. Return a formatted response (if any). Repeat.

Every web application that I have written, deployed, debugged, and every one I'll illustrate in this series implements this request-response cycle.

I mention this because I've worked with a few RPG developers who have had trouble transitioning from the traditional green-screen program cycle:

1. Write record(s) to a display file.
2. Read record(s) from a display file.
3. Perform some process based on what was returned from the read(s). Repeat.

Remember, clients initiate requests. Servers receive them and return formatted responses. That's the cycle. The only exception that I'm aware of is when servers push content to clients, which may implement web sockets.

Program Scope

Most interactive database-management programs should be scoped to handle more than one type of request (i.e. create, read, update, delete, etc.) as opposed to only handling one type of request per program. The latter leads to excessive resource utilization (open files, etc.) and code fragmentation. Most of the programs I write are scoped to handle a range of about 6-12 types of requests; some more, some less.

Programs may include sections to handle "initialization", "branching" based on requested "actions" (i.e. create, read, update, delete, etc.), and "termination" (clean-up; covered in a later article). In the case of trivial applications (perhaps less than 5 types of requests) I prefer branching to subroutines in the same module to process requests and generate responses. Otherwise, I prefer calls to external service-program subprocedures which keeps source members small, modular, and context-specific (more on this in later articles).

Program Content

RPG source members are less cluttered and more readable when we don't mix code belonging to other language environments. I don't mix HTML, XML, JSON, and JavaScript code with RPG, for example.

I store formatted text streams (HTML, CSS, XML, JSON, JavaScript) in external stream files as "templates" and use a few generic procedures to insert program data into templates while generating formatted output streams, which are returned to browsers.

In the next article ([Part 12](#)) I intend on discussing the idea of externalizing SQL. And sometime after that, I'd like to address the idea of externalizing record-level access. This type of ILE application consists largely of references to external procedure prototypes, external data structures, and calls to external procedures.

This program handles only one type of request. So it has minimal structure. We'll look at more complex examples in future articles.

Entity name and address pertaining to this application are defined as columns in a table named TLOC100P. This program implements an interface wherein an externally described data structure named "rw" is passed as an entry parameter, which will be delineated in a future article.

I should explain the seven (7) procedures which are called in this program.

1. wtnOpen() - loads one or more "templates" into memory which may contain HTML, CSS, XML, JSON, JavaScript (any formatted text). The source of these templates resides in IFS stream files. These text streams may be peppered with "markers" which are delineated by double curly braces (i.e. {{ADDR}}. Markers are place-holders for program data which will be injected into the output-stream at runtime. A reference to the template is stored in a program variable.
2. wtnSetInst() - tells the template engine which template to use for generating output streams. This may be necessary when multiple programs are called within a single IBM i JOB, wherein each program may open different templates.
3. wtnQryGet() - retrieves the value of a query-string parameter by name. Query string parameters are passed on URL's. In this case the key to the DB table is passed.
4. strToInt() - converts a character representation of a number to an integer data type (includes error logic).
5. wtnRecSet() - tells the template engine which "record" in the template to prepare for program variable injection. Templates are divided into sections which I call "records".
6. wtnFldSet() - injects program data into the template's output stream.
7. wtnRecWrt() - instructs the template engine to output the record (stream).

The template used in this program contains:

```
<!RECORD:ADDRESS>
ar('{{ADDR}}');
```

This program is injecting a JavaScript function call into the browser which inserts a table row which contains an entity name and address.

The RPG program source

Please review the complete RPG source.

Program IUI100

```
//-----
// file specifications
//-----

ftloc100p if e      k disk

//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdwtnapi#1
/copy *libl/qrpglesrc,rdstrapi#1

//-----
// module level data
//-----
```

```

d rw      e ds      extname(rwpgmc) qualified

d s1      s      * inz(*null)
d addr    s      160a varying

//-----
// program entry
//-----

c *entry      plist
c      parm      rw

//-----
// retrieve reference to external template
//-----

/free

if s1 = *null;
  s1 = wtnOpen('IUI100');
else;
  wtnSetInst(s1);
endif;

//-----
// retrieve and return location address
//-----

tloc100k = strToInt(wtnQryGet('tloc100k'));

chain tloc100k tloc100p;

if %found();
  addr = %trimr(name) + '<br>' + %trimr(gaddrstr);
  wtnRecSet('ADDRESS');
  wtnFldSet('ADDR':addr);
  wtnRecWrt('ADDRESS');
endif;

return;

/end-free

```

Given the preceding explanation, I hope you can see what the program is doing. Calling this program 10 times consumes a total of 2 milliseconds of CPU time on an IBM i model 520 server, which underscores the efficiency of the interface.

Departure From the Display File Paradigm

This simple application demonstrates a departure from the display-file paradigm in a number of ways which I view as beneficial. Let's see if you agree.

- The application entry point is a static HTML page (it can be accessed without involving custom program logic or resources).
- A simple JavaScript utility merges content pulled from a JSON data object at runtime (again without involving custom program logic or resources).
- UI events can be handled on the client (i.e. clear all table entries without involving server resources; various key-press and mouse events too).
- Server requests are handled asynchronously (without locking the display).
- The size of the data stream output from the server is reduced.
- The request-response cycle is more flexible and supports a more event driven user interface than the display-file paradigm (i.e. write record(s) - read record(s)).
- The UI automatically adapts to various screen sizes and device types (desktop, laptop, tablet, & cell phones).

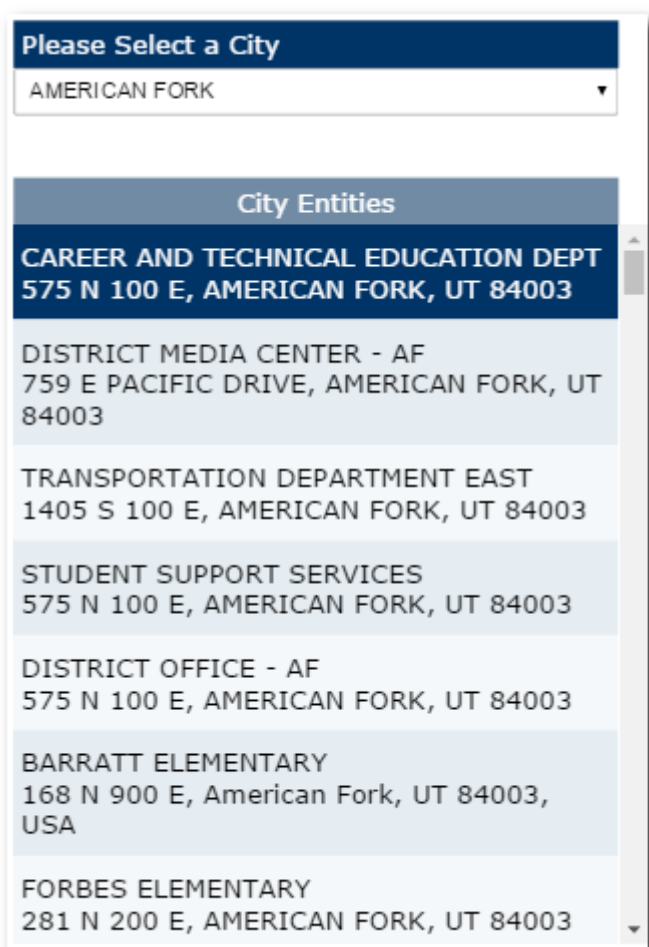
IBM i Modernization - The User Interface (Part 12)

<https://rd.radile.com/rdweb/info2/ibmiui12.html>

The example application which is reviewed in this piece is nearly identical to the one which was used in [Part 11](#). It might help for you to review that before proceeding here.

The main distinction between the two examples is that this one returns data from an SQL query (result set) rather than record-level access (i.e. RPG CHAIN). This presents an opportunity to discuss the idea of externalizing SQL I/O rather than embedding SQL in RPG source members.

Click on the following screen shot to try the application in a new tab or window.



This application presents a drop-down list of cities in Utah. When you select one, a request is sent to an IBM i server which returns a list of "entities" which are located in the "selected" city.

How Does It Work?

Since the client-side code is nearly the same as that which was explained in [Part 11](#), let's jump right into the the ILE RPG code.

Program IUI101

```
//-----  
// procedure prototypes  
//-----
```

```

/copy *libl/qrpglesrc,rdwtnapi#1
/copy *libl/qrpglesrc,rdcsrapi#1

//-----
// module level data
//-----

d rw      e ds      extname(rwpgmc) qualified

d c1      s      * inz(*null)
d s1      s      * inz(*null)
d addr    s      160a varying
d filter  s      80a varying
d quote   c      ""

//-----
// program entry
//-----

c *entry  plist
c      parm      rw

//-----
// retrieve reference to external template
//-----

/free

if s1 = *null;
  c1 = csrNew('TLOC100P': 'NAME, GADDRSTR');
  s1 = wtnOpen('IUI100');
else;
  csrSetInst(c1);
  wtnSetInst(s1);
endif;

//-----
// retrieve and return entity name and address
//-----

filter = 'CITY = ' + quote + wtnQryGet('city') + quote;

csrSetFilter(filter);
csrRefresh();

dow csrGoto(csr_next);
  addr = %trimr(csrColStr('NAME'))
        + '<br>' + %trimr(csrColStr('GADDRSTR'));
  wtnRecSet('ADDRESS');
  wtnFldSet('ADDR':addr);

```

```
wtnRecWrt('ADDRESS');  
enddo;  
  
wtnRecWrt('FINISH');  
  
return;  
  
/end-free
```

The first thing that you might notice is that the program has NO "file" specifications ("F" specs). This may be of interest to some because it addresses a problem which has nagged RPG programmers for years - the forced recompile of programs when the layout of database files (tables) change.

While the program has no "F" specs, neither does it have any embedded SQL. The procedural interface which I'm using is a high-level wrapper around the IBM i SQL-CLI interface which has proven to be quite functional, readable, and elegant.

The Cursor API

This program pulls in a set of procedure prototypes which reference service-program procedures which are bound to the program at compile time. The procedures implement a nice programming interface for working with SQL and SQL result sets (cursors). A cursor in this case is an SQL query against any externally defined database object (i.e. table or logical file such as an SQL view).

```
/copy *libl/qrpglesrc,rdcsrapi#1
```

A brief explanation of the "cursor" procedures used in this program:

1. `csrNew()` - creates a new instance of a cursor; returns a pointer to it. The 10-character data object name is passed as a parameter. A string containing a comma-separated list of column names is an optional second parameter.
2. `csrSetInst()` - tells the cursor engine which cursor to work with. This may be necessary when multiple cursors are open within a single program, or multiple programs are called within a single IBM i JOB, wherein each program may hold references to different cursors.
3. `csrSetFilter()` - defines an SQL "where" clause, which is passed as a string parameter.
4. `csrRefresh()` - generates a result set (cursor); dynamically runs an SQL SELECT statement.
5. `csrGoto()` - fetches a row from a cursor (i.e. first, next, prior, last, by row number).
6. `csrColStr()` - returns a string value from a named column in the current row. Other procedures return values of other data types (i.e. integer, real, float, date, time, timestamp).

In my opinion, RPG code is simplified (often significantly) when SQL statements are defined externally in SQL views, and referenced by a name. The remaining code is more readable. Other advantages include:

- No need for a pre-compiler.
- Error handling is simplified (i.e. no SQL code values to deal with).
- Programming interface is simplified (i.e. no SQL diagnostics or descriptors to deal with; fewer options for programmers to deal with).

Cursor Utilities

This example application includes a drop-down list of "cities" in Utah which is dynamically generated at runtime from the contents of a JSON array ([cities.json](#)), which is generated from the following SQL view:

create or replace view tloc100v1 as
select distinct city from tloc100p

I used the following program to generate the [cities.json](#) stream file from the SQL view:

Program JEXP002

```
//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdcsrapi#1

//-----
// miscellaneous variables
//-----

d cols      s      128a  varying
d stmf      s      128a  varying
d c1        s      *
d squote    c      ''''

/free

  csrInit();

  cols = 'CITY AS VALUE, CITY AS TEXT';

  c1 = csrNew('TLOC100V1':cols);

  stmf = '/rdweb/info3/iui12/cities.json';

  csrSetOrder('TEXT');
  csrRefresh();
  csrToStmf(stmf:'4');

  return;

/end-free
```

The procedure to note is *csrToStmf()* (cursor to stream file) which is a utility that generates a JSON array from an SQL result set. Actually the procedure transforms cursors to a variety of formats (i.e. XML, CSV, Plain Text).

If the stream file name is empty, the procedure outputs the stream to the HTTP server. So it can be used to dynamically generate JSON arrays and return them to browsers.

Summary

Transitioning from the display-file paradigm in order to support browser user interfaces provides an opportunity to adopt procedural interfaces which simplify and streamline database programming. The *cursor API* shown in this article is an example of that.

IBM i Modernization - The User Interface (Part 13)

<https://rd.radile.com/rdweb/info2/ibmiui13.html>

I believe that part of modernizing the user interface includes using web technologies to generate nicely formatted and stylized printer output, which is often an improvement over spool files (whether text, or text with overlays, or Advanced Function Printing (AFP) on IBM i).

Streaming printer output to IFS directories may also facilitate the exchange and archival of reports; That may be an improvement over retaining spool files in IBM i output queues.

Text based spool files are simple to create, though visually unappealing. Over a period of years it has surprised me to hear many situations where IBM i programmers struggle with getting IBM's advanced-function printing (AFP) to work for them. I believe that HTML and CSS offer a better alternative.

Producing reports is a little different than generating web pages; reports have specific page sizes (i.e. 8.5" x 11.0", and others), orientation (portrait and landscape; they may have page sides (i.e. front and back), headers and footers that may need to be printed at the top and bottom of each page (may include page number).

W3C specifications that support printed output tend to be less known than ones which support screen output. But they do exist, thankfully. This makes it nice in that your knowledge and skills with HTML and CSS can be used to produce printed output that may include graphics, charts, backgrounds, and other styling that visually enriches the output.

My Approach to Report Generation

My approach that I'll explain with sample code in the next article often entails:

1. Creating one or more SQL views that contain the data desired in the report (table joins, summaries, etc.).
2. Creating an HTML template that defines report bands (page headings, page detail, page-break summaries, page footers, final totals, etc.).
3. Using a standard CSS style sheet that includes declarations of how pages should be printed.
4. Writing an RPG program that includes invoking procedures that generate an IFS stream file that holds an HTML version of the report.
5. Invoking a procedure that transforms the HTML output into a PDF format.
6. Optionally outputting the report to a browser.

Note that this approach is intended for application developers as opposed to say wizard-style reporting utilities which may be designed for end users. Sometimes the distinctions between application developers and end users blur.

This approach is better for reports that may be placed on end-user menus (i.e. with parameter prompting), or generated in batch, or generated in connection with another process such as a payroll posting.

One advantage to this approach is its runtime efficiency. The database interfaces and APIs that we use for report generation (HTML and PDF) are entirely IBM i-based, and are many times more efficient than other tooling options.

Layout Concept

As with recent articles, I intend on sharing a working example, including code, and explaining how it works. However, as I began working on a sample report for this article, I decided to break the topic into two pieces - a prelude (if you will) that introduces a layout concept for the report's detail band in this piece - then cover page headers and footers, summary counts, database access, streaming to the IFS, and PDF generation in the next.

It seems that the detail band of most reports entails the use of tables with column headings. However in this case, I was inclined to generate output where the layout may automatically adapt to page size and orientation. This is quite comparable to the articles that I wrote previously about "responsive design", where the layout automatically adapts to the width of the viewing area.

Please *click* on the following screen shot to see how the layout might adapt to your browser window.

ABRAVENEL HALL 123 S TEMPLE, SALT LAKE CITY, UT 84101, USA	ADOBE SYSTEMS ADOBE WAY @ 3900 N, LEHI, UT 84043, USA
ADONIS BRONZE 450 ALPINE HWY, ALPINE, UT 84004, USA	AF FRONT RUNNER TRACKS STATION 782 W 200 S, AMERICAN FORK, UT 84003, USA
ALL STAR BOWLING 12101 S STATE ST, DRAPER, UT 84020, USA	ALPINE ELEMENTARY 400 E 300 N, ALPINE, UT 84004, USA
ALPINE SUMMIT PROGRAMS 1581 1000 S, OREM, UT 84058, USA	ALPINE TABERNACLE 110 E MAIN ST, AMERICAN FORK, UT 84003, USA
ALTA HIGH SCHOOL 11055 S 1000 E, SANDY, UT 84094, USA	AMERICAN FORK CANYON N CANYON RD, CEDAR HILLS, UT 84062, USA

How Does it Work?

The first point that I'd like to make is that I'm planning on using the same HTML template and CSS properties to produce printer output that I might otherwise use in a browser user interface. Let's take a look at the <body> of that web page:

```
<body>
<div ht-container="entities" ht-href="ibmiui13a.html" ht-jref="entities.json" ht-cache ht-repeat></div>
</body>
```

Like with previous articles, I'm using a JavaScript utility that automatically produces attractive layout via the merger of an HTML template with a JSON data object (array). This happened to be a convenient way to play with the layout prior to, or without the need of writing any server-side logic. By way of review, the *body* encapsulates a <div> container that has attributes that declare a merge of an HTML template [ibmiui13a.html](#) with a JSON array [entities.json](#).

Let's take a look at the [ibmiui13a.html](#) template that is used for generating "entity" details:

```
<div class="il fl h2 mw" style="margin:10">
<div class="il fl w4 hi-row pd-5">{{NAME}}</div><br>
<div class="il fl w4 pd-5 bs-1 h1">{{ADDR}}</div>
</div>
```

The HTML template consists of a set of <div> elements that have a series of CSS class name references that control the flow (float) from left to right, background colors, border shadows, margins, etc.

It has become my preference to combine a number of styling options via abbreviated CSS class names. I tried a number of CSS options before settling on a final appearance. It was helpful to have a utility merge the HTML template and JSON array to get a closer picture of the final output.

The template API that I use in RPG, as well as the JavaScript *merge* utility - they both use double curly brace "markers" (i.e. {{NAME}}, {{ADDR}}) to delimit the location where "data" is inserted at runtime.

The Cursor API

I introduced a set of procedures in [Part 12](#) that externalize SQL, that we've used in hundreds of interactive applications and reports. I plan on using it again to output live data in the coming sample report.

I raise the point now to show the ILE RPG program that I used to generate the [entities.json](#) file that I've been testing with.

Program JEXP003

```
//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdcsrapi#1

//-----
// miscellaneous variables
//-----

d cols      s      128a  varying
d stmf      s      128a  varying
d c1        s          *
d quote     c          ""

/free
```



```
csrInit();

cols = 'NAME, GADDRSTR AS ADDR';

c1 = csrNew('TLOC100P':cols);

stmf = '/rdweb/info3/iui13/entities.json';

csrSetOrder('NAME');
csrRefresh();
csrToStmf(stmf:'4');

return;

/end-free
```

The `csrToStmf()` - procedure is quite powerful in that it transforms any SQL result set into a JSON array, and outputs it to a stream file, or optionally outputs the JSON to the IBM i HTTP server.

Summary

I've taken an opportunity to introduce the idea of using standard HTML and CSS to generate nicely formatted and stylized reports. Most of the "how to" is reserved to the next article.

I thought it might help to first review how my merge utility and JSON generation utility might be used as a preliminary step to try different formatting options prior to using RPG to generate the report.

IBM i Modernization - The User Interface (Part 14)

<https://rd.radile.com/rdweb/info2/ibmiui14.html>

This piece is a continuation of the topic of using HTML, CSS, and ILE RPG to produce nicely-formatted, stylized output, which is ready for print. Please review [Part 13](#) to see where this article picks up from.

Please *click* on the following screen shot to open the PDF file which I generated for this article. There's also an HTML version [here](#).

City: **ALPINE**

Date: **11/30/16**

Entities

ADONIS BRONZE

450 ALPINE HWY, ALPINE, UT 84004, USA

ALPINE ELEMENTARY

400 E 300 N, ALPINE, UT 84004, USA

RIVER MEADOWS SENIOR LIVING

137 RED PINE DR, ALPINE, UT 84004, USA

TIMBERLINE MIDDLE SCHOOL

500 WEST CANYON CREST, ALPINE, UT 84004

WESTFIELD ELEMENTARY

380 S LONG DRIVE, ALPINE, UT 84004

City Entities: **5**

How Does it Work?

The report is based on an HTML template which looks something like the following (click on the screen shot to view the template in your browser):

City: **{{CITY}}**
Entities
Date: **{{DATE}}**

{{NAME}}

{{ADDR}}

City Entities: **{{COUNT}}**
Total Entities: **{{COUNT}}**

Double curly braces (i.e. `{{CITY}}`) denote "markers" where data will be inserted from an ILE RPG program at runtime.

The HTML template is divided into five (5) sections which are delimited by "RECORD" names. Let's take a look at each section, one by one:

The first section (RECORD:BEGIN) marks the beginning of the HTML page. Note that CSS styles are imported into the page at runtime. It helps to place HTML and CSS in separate files. That keeps the code looking tidy, and provides a basis for sharing a single common style sheet across many reports.

```
<!RECORD:BEGIN>
<html>
<head>
<style>@import url("/rdweb/info3/iui14/report.css");</style>
</head>
<body>
```

The next section (RECORD:PAGEHEAD) marks the beginning of the page heading which consists of a "city" marker, the report title "Entities" and a "date" marker which will be replaced with the date that the report is generated.

Take special note of the `class="pagehead"` `<div>` attribute. That is a key to understanding how page headings are automatically repeated in the top margin of every page, which I'll explain further down.

```
<!RECORD:PAGEHEAD>
<div class="pagehead">
<table width="100%" class="mg-5">
<tr>
<td align="left" width="300" valign="top">City: <b class="cl-1">{{CITY}}</b></td>
<td align="center"><h1 class="cl-1">Entities</h1></td>
<td align="right" width="300" valign="top">Date: <b class="cl-1">{{DATE}}</b></td>
</tr>
</table>
</div>
```

The next section (RECORD:DETAIL) marks the beginning of the report's "detail" band. It holds markers for "name" and "address" which will be retrieved from an IBM i database table at runtime and inserted into the report.

```
<!RECORD:DETAIL>
<div class="il fl h2 mg10 pbn">
<div class="w4 hi-row pd-5">{{NAME}}</div>
<div class="w4 pd-5 bs-1 h1">{{ADDR}}</div>
</div>
```

The next section (RECORD:CITY) marks the beginning of the report break which includes a summary count of the number of "entities" in a city.

```
<!RECORD:CITY>
<div class="mg-5 pba" style="clear:left">City Entities: <b class="cl-1">{{COUNT}}</b></div>
```

The next section (RECORD:END) completes the layout and includes a marker for a total count of all entities in the database.

```
<!RECORD:END>
<div class="mg-5" style="clear:left">Total Entities: <b class="cl-1">{{COUNT}}</b></div>
</body>
</html>
```

That completes the HTML template. It's only 27 lines of code and quite digestible. Let's take a look at some of the CSS entities and attributes which are pertinent to printer-ready output.

The CSS Style Sheet

The first thing to note is the "pagehead" class attribute which declares a "position" of "running(header)". That is another key to getting page headings to automatically appear at the top of each page.

```
.pagehead {
  position:running(header);
}
```

There's quite a bit of automation which is built in to the rather terse declarations shown below:

- A top margin of 1 inch and all other margins of one half inch on each page.
- Each page is standard letter size (8.5 inches by 11 inches).
- Page orientation is portrait (landscape is an option).
- Page headers are centered in the top margin.
- An automated page count is centered in the bottom margin.
- Other layout criteria.

```
@page {
margin:1.0in .5in .5in .5in;
size:letter portrait;
@top-center {
content: element(header);
vertical-align:bottom;
}
@bottom-center {
font-size:16px;
content: "Page " counter(page);
```

```
vertical-align:middle;
}
}
```

The "pba" class is assigned to an HTML element (city total count in this case), which causes a forced page-break to occur after a summary count is output.

```
.pba {
  page-break-after:always;
}
```

Browsers and other rendering engines may be asked to avoid page-breaking on any element which references the "pbn" class. This happens to be one instance where the output from our PDF generator is slightly out of sync with recent browser versions. But the difference wasn't a big concern.

```
.pbn {
  page-break-inside:avoid;
}
```

The CSS file encapsulates other attributes which modify the look and layout of the page. For brevity sake, I shall limit my commentary to the attributes which pertain specifically to printing. There are other CSS options which provide further print control such as front and back printing which I didn't need in this report. So that's about it.

The ILE RPG Program

The RPG code is delineated as follows:

- Imports procedure prototypes pertaining to SQL cursors and stream file output.
- Declares variables having module-level scope.
- Generates an SQL result set (cursor).
- Loads an HTML template into memory.
- Opens an IFS stream file for output.
- Fetches rows from the SQL cursor within a loop.
- Outputs report details.
- Outputs summary counts and page breaks when the city changes.
- Outputs final entity counts.
- Invokes a procedure to generate a PDF version of the report.
- Performs clean up and ends the program.

It's noteworthy that page footers and page numbers are automatically handled by CSS properties without the RPG programmer having to code for it.

This piece introduces a stream-file API which renders formatted output based on a template (HTML in this case). The syntax is nearly identical to the API which I use for generating HTTP streams for browsers.

Please review the complete RPG source. I hope it makes sense.

Program IUI102

```
//-----
// procedure prototypes
```

```
//-----

/copy *libl/qrpglesrc,rdstmap#1
/copy *libl/qrpglesrc,rdcsrapi#1

//-----
// module level data
//-----

d c1      s      * inz
d s1      s      * inz
d f1      s      * inz
d city    s      40a varying
d save_city s    40a varying
d city_count s    5u 0 inz
d entity_count s  5u 0 inz

/free

//-----
// generate an SQL cursor
//-----

csrInit();

c1 = csrNew('TLOC100P':'CITY, NAME, GADDRSTR AS ADDR');

csrSetOrder('CITY, NAME');
csrRefresh();

//-----
// load HTML template into memory and open a stream file
//-----

stmInit();

s1 = stmFmtLoad('IUI14');
f1 = stmFileOpen('/rdweb/info3/iui14/entities.html');

//-----
// begin report
//-----

stmRecWrt('BEGIN');

//-----
// fetch cursor rows (main loop)
//-----

dow csrGoto(csr_next);
  exsr check_city;
```

```
exsr write_detail;
enddo;

//-----
// finish report (output final entity count)
//-----

eval city = 'TOTALS';

exsr write_city;

stmRecSet('END');
stmVarSet('COUNT':%char(entity_count));
stmRecWrt('END');

//-----
// generate PDF document
//-----

stmToPDF(0);

//-----
// end program
//-----

csrTerm();

*inlr = *on;

return;

//-----
// check to see if city has changed
//-----

begsr check_city;

city = csrColStr('CITY');

if city = save_city;
  leavesr;
endif;

save_city = city;

exsr write_city;

endsr;
```

```
//-----
// output entity count and page break on change in city
//-----

begsr write_city;

if city_count > 0;
  stmRecSet('CITY');
  stmVarSet('COUNT':%char(city_count));
  stmRecWrt('CITY');
  city_count = 0;
endif;

stmRecSet('PAGEHEAD');
stmVarSet('CITY':city);
stmVarSet('DATE':%char(%date():*mdy));
stmRecWrt('PAGEHEAD');

endsr;

//-----
// write detail
//-----

begsr write_detail;

stmRecSet('DETAIL');
stmVarSet('NAME':csrColStr('NAME'));
stmVarSet('ADDR':%trimr(csrColStr('ADDR')));
stmRecWrt('DETAIL');

city_count += 1;
entity_count += 1;

endsr;

/end-free
```

HTML and CSS Tips

I use a utility which transforms HTML into a PDF file which is based on an open-source product named iText which renders output somewhat differently than browsers. The PDF version is better in some respects (page layout is more precise and better fitted for print). However there are a number of things which may be supported in browsers but not in PDF rendering (and visa-versa).

HTML must follow XML standards which enforce the use of beginning and ending tags (XML must be well-formed). I often use the stand-alone
 tag in web pages to implement line breaks after text. But that makes a mess when attempting PDF generation. You can use <p></p> instead.

When using CSS styles for layout (i.e. height, width, padding, margins, etc.) browsers default to measurements in pixels when the unit of measure is not explicit in CSS properties. Units of



measurement (i.e. pixels, inches, em's) must be explicitly stated in CSS styles for proper PDF rendering.

Some CSS3 effects may be supported in browsers but not PDF rendering. For example, in this case I used a border shadow effect which was not rendered in the PDF version. The PDF output still looked good. So I didn't try an alternative style.

Conclusions

HTML and CSS provide for nicely-formatted and stylized printing. The APIs I've shown in this piece perform this with remarkable efficiency in RPG programs.

Skills with these technologies make generating output like this within the reach of IBM i developers.

IBM i Modernization - The User Interface (Part 15)

<https://rd.radile.com/rdweb/info2/ibmiui15.html>

This piece demonstrates a browser version of a single-page subfile and explains how it works. It was inspired by an example of a green-screen program which was delineated in this [blog](#). Please *click* on the following screen shot to try the application.

<div> < < > > </div> <div>Page: 1 Of: 61</div> <div>Entities</div>	
Name	Address
ABRAVENEL HALL	123 S TEMPLE, SALT LAKE CITY, UT 84101, USA
ADOBE SYSTEMS	ADOBE WAY @ 3900 N, LEHI, UT 84043, USA
ADONIS BRONZE	450 ALPINE HWY, ALPINE, UT 84004, USA
AF FRONT RUNNER TRACKS STATION	782 W 200 S, AMERICAN FORK, UT 84003, USA
ALL STAR BOWLING	12101 S STATE ST, DRAPER, UT 84020, USA
ALPINE ELEMENTARY	400 E 300 N, ALPINE, UT 84004, USA
ALPINE SUMMIT PROGRAMS	1581 1000 S, OREM, UT 84058, USA
ALPINE TABERNACLE	110 E MAIN ST, AMERICAN FORK, UT 84003, USA
ALTA HIGH SCHOOL	11055 S 1000 E, SANDY, UT 84094, USA
AMERICAN FORK CANYON	N CANYON RD, CEDAR HILLS, UT 84062, USA

Page Navigation

The current page number is displayed, along with total number of pages. VCR-style buttons are used to implement page navigation.

You might see this type of navigation in certain web applications. It's more common in the green-screen paradigm. The use case may be applicable to database tables, or SQL views, or SQL result sets which hold many rows; You may not want to download all rows - all at once. But users may have occasion to "browse".

How Does it Work?

The RPG program which responds to "page" requests is using my "cursor API" which "externalizes" SQL. Procedures are used to "define" a page size for an SQL result set, determine the total number of pages in the result set, navigate to pages (given a page number), and fetch the rows pertaining to that page.

In contrast to the green-screen paradigm, a single IBM i Job may be handling requests from many concurrent browsers at the same time. That won't matter. Navigation from page to page with this API doesn't rely on any end-user or cursor being in any given state in order to work.

The RPG program handles essentially only 2 requests from clients:

1. Returns the total page count when the browser first "enters" the application (displayed on the screen).
2. Returns the rows pertaining to any "page" that any client may request.

Clients in this case means any browser which may go to the static HTML page pertaining to the application. The page is composed of 3 tables:

1. Page Status, and VCR Buttons.
2. Subfile Column Headings.
3. Subfile Rows.

Let's review the layout of each.

Page Status/VCR Buttons:

```
<table width="720" class="hi-row" cellpadding="5" cellspacing="0">
<tr height="50">
<td width="35"><button class="vcr" onClick="first()" title="First Page">|&lt;</button></td>
<td width="35"><button class="vcr" onClick="prior()" title="Prior Page">&lt;</button></td>
<td width="35"><button class="vcr" onClick="next()" title="Next Page">&gt;</button></td>
<td width="35"><button class="vcr" onClick="last()" title="Last Page">&gt;|</button></td>
<td>&nbsp;</td>
<td width="30" align="right">Page:</td>
<td width="25" align="right" id="cur"></td>
<td width="30" align="right">Of:</td>
<td width="25" id="pages"></td>
<td>&nbsp;</td>
<td width="120" align="center">Entities</td>
</tr>
</table>
```

Column Headings:

```
<table width="720" class="bg-1">
<tr>
<td width="40%">Name</td>
<td>Address</td>
</tr>
</table>
```

Subfile Rows:

```
<table width="720" id="tb" cellpadding="5" cellspacing="0"></table>
```

The "subfile" begins as an empty table. It's filled in with rows at runtime.

Java Script

A JavaScript function named goto() is invoked when a VCR button is clicked. It sends a request for a new page.

```
function goto(choice) {
  reqGet('/rdcaller/goto.shtml?rwappid=iui103&page=' + choice);
  cur_page = choice;
  cur.innerHTML=choice.toString();
}
```

The response from the server invokes a JavaScript function named ar() repeatedly to add rows to the subfile.

```
function ar(name,addr) {
  var s = '<tr height=50><td width=40%>' + name + '</td>'
  + '<td>' + addr + '</td>';
  tb.insertRowBottom(s);
}
```

Click [here](#) to view the complete custom JavaScript for this application.

The RPG Program

The RPG program introduces a few new procedures pertaining to the "cursor API".

1. csrSetPageSize(10) - Sets the page size to 10 rows.
2. csrPages() - Returns the total number of pages in the result set.
3. csrPage(csr_absolute:page_number) - Positions to the first row in the requested "page".

Program IUI103

```
//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdstrapi#1
/copy *libl/qrpglesrc,rdwtnapi#1
/copy *libl/qrpglesrc,rdcsrapi#1

//-----
// module level data
//-----

d rw      e ds      extname(rwpgmc) qualified

d c1      s          * inz(*null)
d s1      s          * inz(*null)
d row_count  s      10i 0
d page_number s      10i 0
d page_count s      10i 0

//-----
// program entry
//-----

c *entry  plist
c        parm      rw
```

```
/free

//-----
// set references to screen and cursor
//-----

if s1 <> *null;
  wtnSetInst(s1);
  csrSetInst(c1);
endif;

//-----
// branch to subroutines based on requested actions
//-----

select;
  when rw.action = 'INIT';
    exsr do_init;
  when rw.action = 'GOTO';
    exsr do_goto;
  when rw.action = 'PAGECOUNT';
    exsr do_page_count;
endsl;

return;

//-----
// initialization
//-----

begsr do_init;

s1 = wtnOpen('IUI103');
c1 = csrNew('TLOC100P': 'NAME, GADDRSTR AS ADDR');

csrSetOrder('NAME, CITY');
csrSetPageSize(10);
csrOpen();

page_count = csrPages();

endsr;

//-----
// goto the page requested
//-----

begsr do_goto;

page_number = strToInt(wtnQryGet('page'));
```

```

if page_number < 1 or page_number > page_count;
  page_number = 1;
endif;

csrPage(csr_absolute:page_number);

//-----
// fetch all rows on requested page and output to client
//-----

clear row_count;

wtnRecWrt('CR');

dow csrGoto(csr_next) and row_count < 10;
  wtnRecSet('AR');
  wtnFldSet('name':csrColStr('NAME'));
  wtnFldSet('addr':%trimr(csrColStr('ADDR')));
  wtnRecWrt('AR');
  row_count += 1;
enddo;

wtnRecWrt('FIN');

endsr;

//-----
// return page_count to client
//-----

begsr do_page_count;

  wtnRecSet('PC');
  wtnFldSet('pages':%char(page_count));
  wtnRecWrt('PC');

endsr;

/end-free

```

Final Thoughts

As IBM i developers move to browser user interfaces, I think it helps to see how familiar design patterns such as subfile paging may be implemented. The blog referenced in the opening paragraph of this article provides a good basis for comparing a green-screen implementation vs. a browser client-server implementation. In many respects they're not dissimilar.

I hope this encourages green-screen developers to see the possibilities of browser user interfaces. Externalizing SQL using my cursor API offers some nice features over record-level access and embedded SQL (once you become familiar with it).

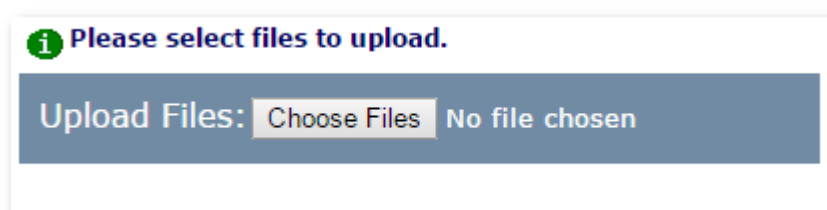
IBM i Modernization - The User Interface (Part 16)

<https://rd.radile.com/rdweb/info2/ibmiui16.html>

I'd like to show an IBM i-centric utility which enables users to select multiple files from their browsers and upload them to the server. Smart phones can optionally take camera shots and upload images.

There are quite a few use cases for this type of application. Schools may need to upload student photos and attach them to student records. Email clients may need to attach files to email messages. Help desks may need to attach files to trouble tickets. Case workers may need to attach files to items in their database. And lots more.

Please *click* on the following screen shot to try the application.



What Does it Do?

This utility is designed for browsers which have implemented the HTML5 specification for multi-file selection.

1. Users are prompted to upload files.
2. The local device's file-selection dialog is displayed when a button is clicked.
3. The file-selection dialog generates a collection of "file" objects including file attributes (name, size, last update date, etc.)
4. An alert is displayed if the sum of the sizes exceeds a maximum.
5. An RPG program generates an "approved" name for each file to be stored on the server.
6. A Net.Data macro is used to receive each file, validate its name, grant *public authority, and move it to an application directory.
7. A status messages is displayed for each file uploaded.
8. A final completion message is displayed after the last file is received.
9. A hyperlink is added to the page for each file received.
10. Users may click hyperlinks to view uploaded files.

Many (perhaps most) examples on the Internet upload all files selected by users - posting just one "upload" request. The interface implemented in PHP (i.e. the \$_FILES Collection) was designed with that technique in mind.

This example (in contrast) asynchronously iterates through the browser's collection of files one at a time so that status updates might be provided during the "upload" process. This enables servers to apply granular validation if needed.

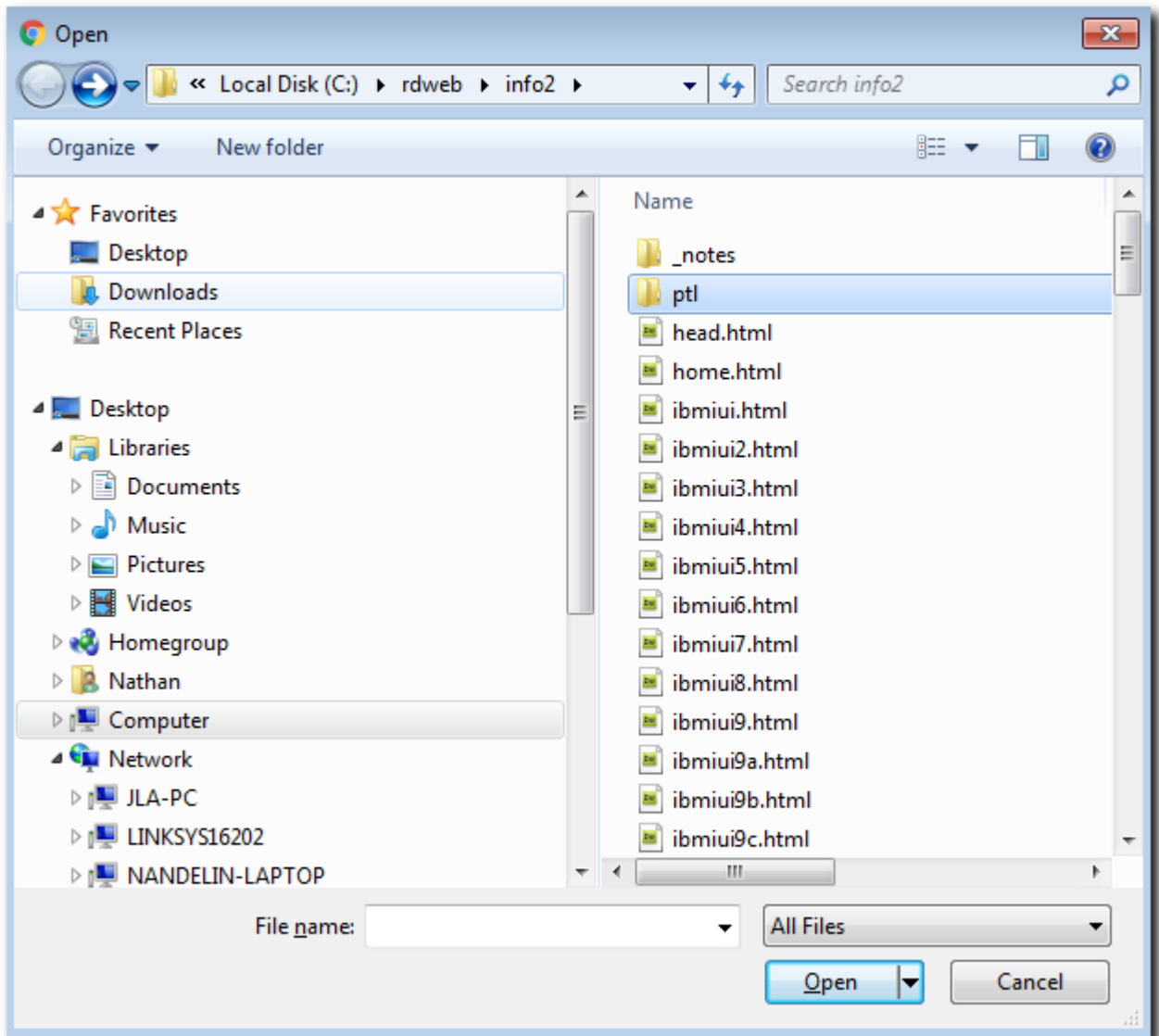
How Does it Work?

The device's file-selection dialog is linked to an <input> element which is defined as follows:

```
<input id="fctl" type="file" multiple onChange="check_size(this.files)">
```

- The type="file" attribute links the component to the device's file-selection dialog.

- The "multiple" keyword enables multiple files to be selected.
- The "this.files" refers to the "FileList" collection which is generated by the dialog.
- The onChange() event listener is triggered when users return from the file-selection dialog.



Let's review the custom JavaScript for this utility.

check_size() is invoked when users complete the file-selection dialog. The code is as follows:

```
function check_size(files) {
var sumSize = 0;
var maxSize = 2000000;

fileList = files;
listIndex = 0;
while (hlinks.childElementCount > 0) hlinks.removeChild(hlinks.childNodes[0]);

for (i=0; i < fileList.length; i++) {
    sumSize = sumSize + fileList[i].size;
}
}
```



```
if (sumSize > maxSize) {
    alert('Sorry, maximum size of files (2,000,000 bytes) was exceeded: bytes = ' + sumSize);
    return;
}
```

```
fctl.disabled = true;
req_obj();
}
```

High-level explanation of check_size():

- Any hyperlinks left over from the last upload are removed from the document.
- File sizes are summed and validated against a maximum.
- An error alert is displayed if the size validation is exceeded.
- The file-selection dialog is disabled by disabling its <input> element.
- req_object() is invoked to generate a authorized server file name (including IFS directory).

The req_object() code is as follows:

```
function req_obj() {
    reqGet('/rdcaller/toobj.shtml?rwappid=iui104&fname=' + fileList[listIndex].name);
}
```

High-level explanation of req_obj():

- The "toobj" action is sent to an RPG program named IUI104.
- The name of the file on the local device is included as a query-string parameter.

The upload() function is invoked in response from the IUI104 program. The code is as follows:

```
function upload(name) {
    toobj = name;
    reqUpload('/db2www/upload.ndm/doload',fileList[listIndex],toobj);
    um.innerHTML='Uploading ' + fileList[listIndex].name;
}
```

High-level explanation of upload():

- The file "name" generated by the server is assigned to "toobj" which has document-level scope (other functions can refer to it).
- reqUpload() is a framework API which sends the file to a Net.Data macro (script).
- The status message is updated.

The cf() function is invoked in response from the Net.Data script. The code is as follows:

```
function cf() {
    add_link();
    listIndex = listIndex + 1;

    if (listIndex < fileList.length) {
        return req_obj();
    }
    um.innerHTML='Upload completed normally.';
    fctl.disabled = false;
}
```

High-level explanation of cf():

- add_link() is invoked to add a hyperlink for the file which was just uploaded.
- listIndex is incremented to point to the next file in the fileList collection.
- req_obj() is invoked to repeat the upload process - for the "next file".
- At the end of the cycle a completion message is assigned and the file-selection dialog is re-enabled.

The add_link() code is as follows:

```
function add_link() {
var a = document.createElement('A');

a.className = 'fl il pd-5 mg-1 cl-1';
a.target = '_blank';
a.href = toobj;
a.innerHTML = fileList[listIndex].name;

hlinks.appendChild(a);
}
```

High-level explanation of add_link():

- A hyperlink is created (tagname="A").
- Attributes are assigned to the hyperlink.
- The hyperlink is appended to a <div> which is defined in the <body>.

An RPG program named UI104 responds to the "toobj" request from browsers. The code is as follows:

Program UI104

```
fxupl100p  uf a e      k disk  usroprn

//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdstrapi#1
/copy *libl/qrpglesrc,rdwtnapi#1

//-----
// module level data
//-----

d rw      e ds      extname(rwpgmc) qualified

d s1      s      * inz(*null)
d toobj    s      128a  varying

//-----
// program entry
//-----

c  *entry    plist
c          parm      rw
```

```
/free

//-----
// set references to screen and cursor
//-----

if s1 <> *null;
  wtnSetInst(s1);
endif;

//-----
// branch to subroutines based on requested actions
//-----

select;
  when rw.action = 'INIT';
    exsr do_init;
  when rw.action = 'TOOBJ';
    exsr do_toobj;
endsl;

return;

//-----
// initialization
//-----

begsr do_init;

  s1 = wtnOpen('IUI104');

endsr;

//-----
// generate and return a file name
//-----

begsr do_toobj;

  open xupl100p;

  toobj = '/rd/iui/iui104/' + %trim(wtnQryGet('fname'));

  toobj = strReplace(toobj: ' ':'_');

  setll (toobj) xupl100r;

  if not %equal();
    id = toobj;
    write xupl100r;
```

```
endif;

close xupl100p;

wtnRecSet('UPLOAD');
wtnFldSet('toobj':toobj);
wtnRecWrt('UPLOAD');

endsr;

/end-free
```

High-level explanation of IUI104:

File-upload utilities which are made available on web sites may be used by hackers to upload changes to web sites which exploit users. To prevent that, IUI104 generates names for uploaded files and stores their values in a DB table (i.e. XUPL100P).

The Net.Data macro checks to ensure that the entry exists which removes the possibility of a hacker overwriting other files which may be exposed on the site.

- Framework APIs are used to communicate with browsers.
- File names are generated and stored in table XUPL100P.
- In this case the local file name is used, but it is prefixed with an approved path. Space characters in the file name are replaced with the underscore (_) character.

The Net.Data upload() script is invoked by browsers which upload files. The code is as follows:

```
%function(dtw_system) chgAut() {
%exec { CHGAUT.CMD OBJ('${fname}') USER(*PUBLIC) OBJAUT(*ALL) DTAAUT(*RX)%}
%}
%function(dtw_system) moveFile() {
%exec { MOV.CMD OBJ('${fname}') TOOBJ('${toobj}') %}
%}
%function(dtw_directcall) checkObj(
in varchar(128) p1,
out char(1) p2) {
%exec {/QSYS.LIB/RDPTL.LIB/CHECKUPL.PGM %}
%}
%html(doload) {
@dtw_assign(path, ${toobj})
@dtw_assign(found, " ")
@checkObj(path, found)
%if (found == "1")
%if (@dtwf_rexists(toobj)=="Y") @dtwf_remove(toobj) %endif
@chgAut()
@moveFile()
cf();
%endif
%}
```

High-level explanation of Net.Data upload():

Net.Data tends to have somewhat terse syntax. But it is an exceptional tool for invoking calls to ILE programs, IBM i system commands, and other language environments (i.e. SQL).



- @checkObj() is a wrapper around an RPG program which ensures that the "toobj" is authorized (exists in Table XUPL100P).
- @chgAut() is a wrapper around an IBM i command which grants *public authority to the uploaded file.
- @moveFile() is a wrapper around an IBM i command which moves uploaded files from a temporary directory to a final one.
- When complete, returns "cf();" which invokes the function in the browser.

Wrapping Up

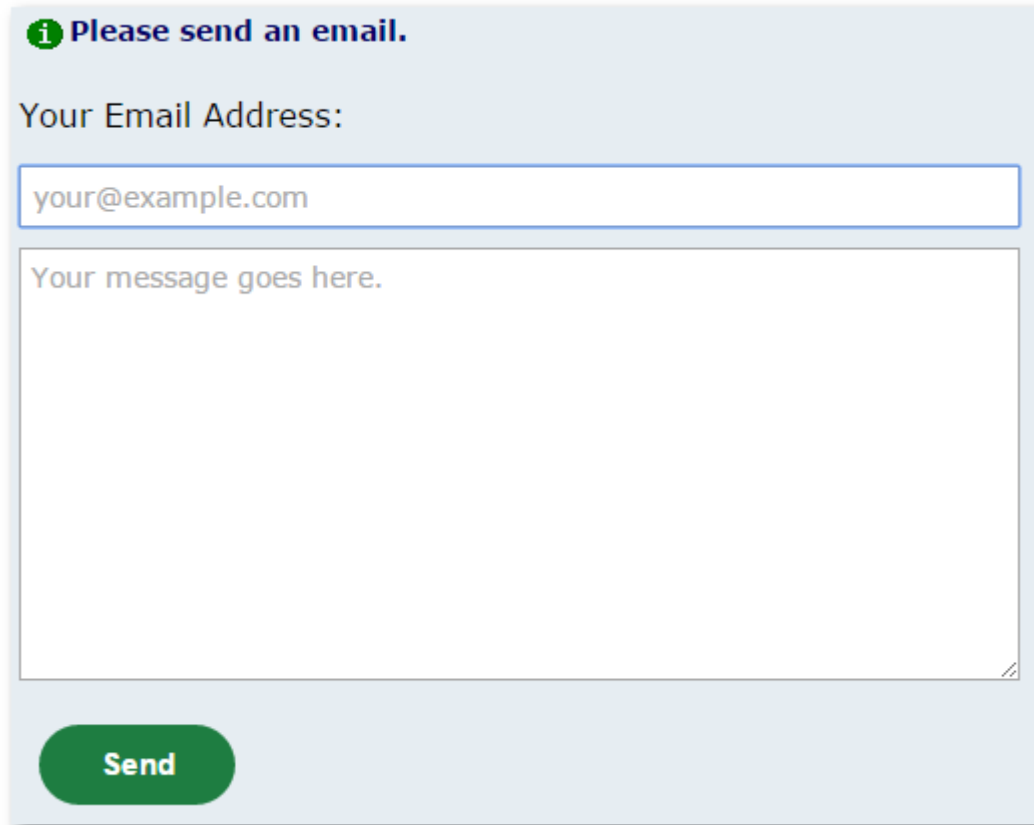
A file-upload utility like this provides an essential element for applications which upload and "attach" various types of files to database records, email messages, SMS messages, etc. I may delve further into that in the next piece.

As an alternative to the "look and feel" of the "input element" which browsers link to file-selection dialogs, some developers create their own widgets which may have a different look and feel. Some of the code could be "hardened" to further frustrate hacking (i.e. ensuring that browsers accept cookies and monitoring their "sessions").

IBM i Modernization - The User Interface (Part 17)

<https://rd.radile.com/rdweb/info2/ibmiui17.html>

This article discusses the idea of sending email from web browsers. Please *click* on the following screen shot to try it.



i Please send an email.

Your Email Address:

Your message goes here.

Use Cases

Email is of course a basic necessity for modern business. Large and small services providers offer robust web-based clients - some of which have become hugely popular.

I believe that IBM i would be a great platform for hosting email as a service for thousands of concurrent users similar to Google and Yahoo Mail. However, for purposes of this discussion, let's keep the requirements small.

Email may simply be an appendage to something like a help-desk application which provides a form for submitting requests for support. Email messages and attachments might be forwarded to technicians, for example.

Just The Basics

The form featured in this article is designed for browsers which have implemented the HTML5 specification. The application does the following:

1. Prompts users to enter their own email address and message text.
2. The Form's values are sent to an ILE RPG program named IUI105.
3. The IUI105 program sends an email message to a recipients list which is maintained in an IBM i data area. It sends a "thank you" email to the originator of the request. It returns a confirmation to the browser.

I considered adding attachments to email messages by using a file-upload utility similar to the one shown in [Part 16](#). I finally chose to keep the scope of the application smaller so that it might be more digestible.

Email Account Inputs

HTML 5 provides for an input element of type "email" for entering email addresses:

```
<input name="addr" type="email" maxlength="64" style="width:100%; max-width:500px; padding:5;" autofocus placeholder="your@example.com" required pattern="^[^ @]*@[^ @]*">
```

Email inputs invoke some built-in behaviors in HTML5 browsers. Smart phones may show @ and .com keys on their keypads, for example.

Pattern="regular expression" provides basic email-account validation against a regular expression. You can find recommendations for regular expressions via search engine results.

The Form's JavaScript

The send() function is invoked when the form is submitted (via button click or the "enter" key).

```
function send() {
  reqPost('/rdcaller/send.shtml?rwappid=iui105',fm.pd());
  return false;
}
```

High-level explanation of send():

- The form's data elements are posted to the server (program IUI105 is called to process it).
- returning "false" overrides the default form submit (reqPost() uses AJAX instead).

The us() function is invoked by the response from the IUI105 program:

```
function us(rt) {
  fm.sa('msg','value','');
  fm.sf('addr');
  um.innerHTML = rt;
}
```

High-level explanation of us():

- The IUI105 program returns a "thanks" message.
- The value of the "msg" input element is set to an empty string.
- The focus is set to the "addr" input element.

Program IUI105

An RPG program named IUI105 responds to the "send" request. The code is as follows:

Program IUI105

```
//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdstrapi#1
/copy *libl/qrpglesrc,rdemlapi#1
/copy *libl/qrpglesrc,rdwtnapi#1
```

```
//-----
// module level data
//-----

d rw      e ds      extname(rwpgmc) qualified

d s1      s      * inz(*null)

d recipients s      256a dtaara(recipients)

d xeml     e ds      extname(xeml100p) import
d          qualified

d xemlmsg  s      32765a varying import

d crlf     s      2a inz(x'0D25')

d addr     s      256a varying

//-----
// program entry
//-----

c *entry  plist
c        parm      rw

/free

//-----
// set reference to UI template
//-----

if s1 <> *null;
  wtnSetInst(s1);
endif;

//-----
// branch to subroutines based on requested actions
//-----

select;
  when rw.action = 'INIT';
    exsr do_init;
  when rw.action = 'SEND';
    exsr do_send;
  ends;

return;

//-----
// initialization
```



```
//-----
begsr do_init;

s1 = wtnOpen('UI105');

in recipients;

clear xeml;

xeml.subject = 'IBM i UI Modernization - (Part 17)';
xeml.sender = 'noreply@rd.radile.com';
xeml.replyto = xeml.sender;

endsr;

//-----
// send email to data-area recipients
//-----

begsr do_send;

addr = %trimr(wtnFldGet('addr'));
xemlmsg = wtnFldGet('msg');
xemlmsg = strReplace(xemlmsg:'\n':crlf);
xeml.recipients = %trimr(recipients);

emlSend();

//-----
// send "thanks" email to originator
//-----

xeml.message = ' ';

xemlmsg = 'Thanks for your email @ ' + %char(%date():*usa)
+ ' ' + %char(%time():*hms) + ' ';

xeml.recipients = addr;

if addr <> '';
  emlSend();
endif;

//-----
// send "thanks" to browser
//-----

wtNRecSet('US');
wtNFldSet('um':xemlmsg);
wtNRecWrt('US');
```

endsr;

/end-free

High-level explanation of program IUI105:

Program IUI105 binds to a service program named RDEMLAPI which implements an high-level email interface by exporting a procedure named emlSend(), an external data structure named XEML, and a variable-length string named XEMLMSG (in which you can store email message content).

Any ILE program or service program can bind to RDEMLAPI, assign values to XEMLMSG and XEML sub-fields, and call emlSend() to send email messages.

We should review the layout of the XEML data structure which is externally described in file XEML100P:

External File XEML100P

```

A          UNIQUE
A  R XEML100R      TEXT('Email Requests')
A  SENDER  128A    COLHDG('Sender Email Address')
A          VARLEN
A  MESSAGE  128A    COLHDG('Message File')
A          VARLEN
A  SUBJECT  128A    COLHDG('Subject')
A          VARLEN
A  RECIPIENTS 7936A COLHDG('Recipients List')
A          VARLEN
A  CARBONS  7936A    COLHDG('Carbon Copy List')
A          VARLEN
A  BLINDS   7936A    COLHDG('Blind Copy List')
A          VARLEN
A  ATTACHS  7936A    COLHDG('Attachments List')
A          VARLEN
A  REPLYTO  128A    COLHDG('Reply To Address')
A          VARLEN
A  ORG      64A     COLHDG('Organization')
A          VARLEN
A  CTYPE    4A      COLHDG('Content Type')
A  ERRTEXT  128A    COLHDG('Error Message')
A          VARLEN
A  ERRFLAG  1A      COLHDG('Error Flag')
A  KEEP     1A      COLHDG('Keep Message Flag')
A  TS       Z       COLHDG('Timestamp')
A  ID       10S 0    COLHDG('ID')
A  K ID

```

- SENDER - Assign the sender's email address.
- MESSAGE - Optionally assign an IFS stream file name to hold the email, otherwise a temporary IFS file is created.
- SUBJECT - Assign a string which appears in the email "subject" line.
- RECIPIENTS - Assign a comma separated list of recipient email addresses.

- CARBONS - Assign a comma separated list of carbon-copy recipients.
- BLINDS - Assign a comma separated list of blind-carbon-copy recipients.
- ATTACHS - Assign a comma separated list of IFS stream file attachments.
- REPLYTO - Optionally assign an email address to reply to (if different from SENDER).
- ORG - Optionally assign an "organization" which will appear in email headers.
- CTYPE - Optionally assign "HTML" if the content type is HTML-formatted text (otherwise plain/text is used).
- ERRTEXT - Contains any error message returned after calling emlSend().
- ERRFLG - Contains "Y" if emlSend() returns an error.
- KEEP - Assign "Y" if you want to keep the MESSAGE stream file (the default is to delete it after the email is sent).
- TS & ID - Timestamp and a unique key are assigned to each email sent.

You can make repeated calls to emlSend() - just altering the xeml.blinds list - when sending bulk email to many recipients. If you assign a new value to the XEMLMSG string, you need to assign a new value to the MESSAGE (IFS file) to send a different email message.

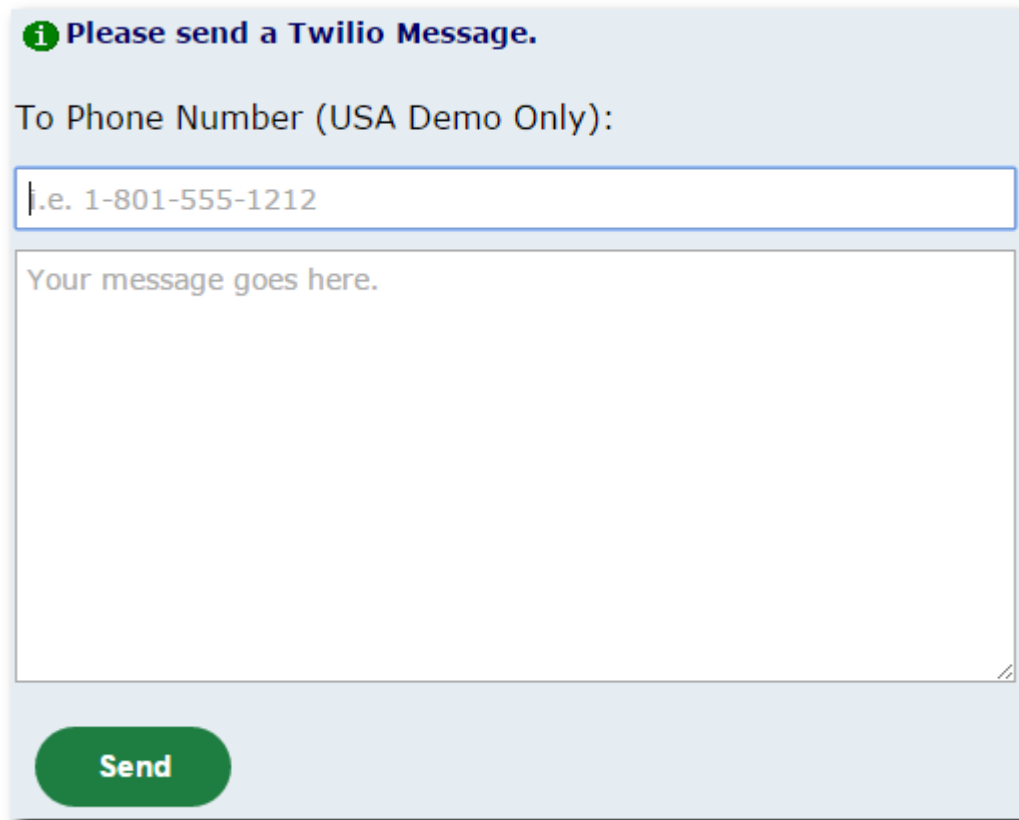
Wrapping Up

There are quite a few use cases for using web browsers and IBM i programs to send email to recipients. Hopefully this article stirs some consideration.

IBM i Modernization - The User Interface (Part 18)

<https://rd.radile.com/rdweb/info2/ibmiui18.html>

This article discusses the idea of sending text messages to smart phones from IBM i applications. Please *click* on the following screen shot to try it.



The screenshot shows a web-based interface for sending a Twilio message. At the top, there is a green circular icon with a white 'i' followed by the text 'Please send a Twilio Message.' Below this, the label 'To Phone Number (USA Demo Only):' is displayed. Underneath the label is a text input field containing the placeholder text 'i.e. 1-801-555-1212'. Below the input field is a large, empty text area with the placeholder text 'Your message goes here.' At the bottom left of the form is a green rounded rectangular button with the white text 'Send'.

Twilio

Twilio is a company which provides a number of web services, including a popular one for sending text messages to smart phones and similar devices.

Twilio issues an account number, a secret authentication token, and a phone number to individuals (or organizations) to enable them to send text messages via their web-service API.

Twilio bills your account for your use of the phone number (\$1 per month at this time), plus a small fee (less than 1 cent for phones in the USA) per message processed. Volume discounts are available. Twilio supports phone carriers all over the world. However I've limited the demo featured in this article to only phone numbers issued in the USA.

Use Cases

Text notifications and alerts for any type of application such as:

- 2-factor authentication.
- Appointment reminders.
- Dispatch notifications.
- Arrival notifications.

What the Demo Does

1. Prompts users to enter a smart phone number and message text.

2. The form's values are sent to an ILE RPG program named IUI106.
3. The IUI106 program connects with the Twilio web service, queues the message for dispatch, and returns a confirmation (or error) to the browser.

Twilio supports up to 10 file attachments per message. However I've limited the scope of the demo to only 1024 characters of message text.

Twilio message queueing is not very prone to errors. Exceptions include malformed phone numbers, insufficient account funds, and such.

Twilio offers an optional call-back web service in the event that you're interested in tracking phone-carrier transmission errors which may occur down line.

The Form's JavaScript

The send() function is invoked when the form is submitted (via button click or the "enter" key).

```
function send() {
  reqPost('/rdcaller/send.shtml?rwappid=iui106',fm.pd());
  return false;
}
```

High-level explanation of send():

- The form's data elements are posted to the server (program IUI106 is called to process it).
- returning "false" overrides the default form submit (reqPost() uses AJAX instead).

The us() function is invoked by the response from the IUI106 program:

```
function us(rt) {
  fm.sa('msg','value','');
  fm.sf('phone');
  um.innerHTML = rt;
}
```

High-level explanation of us():

- The IUI106 program returns a "confirmation" message.
- The value of the "msg" input element is set to an empty string.
- The focus is set to the "phone" input element.

Program IUI106

An RPG program named IUI106 responds to the "send" request. The code is as follows:

Program IUI106

```
//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdstrapi#1
/copy *libl/qrpglesrc,rdtwiapi#1
/copy *libl/qrpglesrc,rdwtnapi#1

//-----
// module level data
//-----
```

```

d rw      e ds      extname(rwpgmc) qualified

d s1      s          * inz(*null)

d twilio   ds        likeds(twilio_t) import

d crlf     s          2a inz(x'0D25')

d twisid   s          64a dtaara(twisid)
d twisecret s        64a dtaara(twisecret)
d twifrom  s          32a dtaara(twifrom)

//-----
// program entry
//-----

c *entry   plist
c          parm          rw

/free

//-----
// set reference to UI template
//-----

if s1 <> *null;
  wtnSetInst(s1);
endif;

//-----
// branch to subroutines based on requested actions
//-----

select;
  when rw.action = 'INIT';
    exsr do_init;
  when rw.action = 'SEND';
    exsr do_send;
endsl;

return;

//-----
// initialization
//-----

begsr do_init;

s1 = wtnOpen('IUI106');
```

```

clear twilio;

in twisid;
in twisecret;
in twifrom;

twilio.sid = %trimr(twisid);
twilio.secret = %trimr(twisecret);
twilio.from = %trimr(twifrom);

endsr;

//-----
// send message to phone
//-----

begsr do_send;

twilio.to = %char(strToInt(wtnFldGet('phone')));
twilio.body = wtnFldGet('msg');
twilio.body = strReplace(twilio.body:'\n':crlf);

if twilSendSms();
twilio.msg = 'Thanks for your message @ ' + %char(%date():*usa)
+ ' ' + %char(%time():*hms) + '.';
endif;

//-----
// send "confirmation" to browser
//-----

wtnRecSet('US');
wtnFldSet('um':twilio.msg);
wtnRecWrt('US');

endsr;

/end-free

```

High-level explanation of program IUI106:

- Program IUI106 binds to a service program named RDTWIAPI which implements an high-level interface with Twilio. Any ILE program or service program can bind to RDTWIAPI to send Twilio text messages.
- My Twilio account number, secret authorization token, and phone number are retrieved from IBM i data areas.
- A procedure named twilSendSms() is paired with a qualified data structure named "twilio" to send text messages.
- We should review the layout of the "twilio" data structure which is imported from the RDTWIAPI service program:

twilio data structure

```
d twilio_t    ds          qualified template
d sid         64a  varying
d secret      64a  varying
d from        32a  varying
d to          32a  varying
d body        1600a varying
d media       8192a varying
d msg         256a varying
```

- sid - your Twilio account number.
- secret - your secret authentication token.
- from - your Twilio phone number.
- to - your recipient's smart phone number.
- body - your message body.
- media - media attachments (up to 10 maximum).
- msg - message returned by Twilio (i.e. error).

Wrapping Up

There are quite a few use cases for sending text messages to smart phones from IBM i applications. Hopefully this article stirs some consideration.

This article was inspired in part by a GITHUB project of Rainer Ross, which uses Node.js to send Twilio texts from IBM i applications [See here](#).

IBM i Modernization - The User Interface (Part 19)


<https://rd.radile.com/rdweb/info2/ibmiui19.html>

This article discusses the idea of filling in web forms and generating PDF documents based on the input. *Click on the following screen shot to try it.*

Form

Preview

Download



Official Certificate of Marriage

paragraph 50(1)(b) Marriage Act 1961

Marriage was solemnised between the parties, details of whom are given below, on the 1st day of January, 20 17, at

Location _____

Detail	Bridegroom	Bride
Surname	Surname	Surname
Other Names	First/Middle/Suffix	First/Middle/Suffix
Usual occupation	Usual occupation	Usual occupation
Usual residence	Usual residence	Usual residence
Conjugal status	conjugal status	conjugal status
Birthplace	Birthplace	Birthplace

Background

A question arose on Midrange Lists in December 2016 asking about generating PDF documents from ILE RPG programs which could be filled in using Adobe Reader.

Rather than using Adobe Reader to fill in forms, I suggested generating PDFs from web forms. This article shows what that might entail.

Web forms are a good option for collecting input from people and at the same time producing PDF documents which cross-reference back to it - which is often better than extracting input from PDF documents (i.e. OCR or re-keyed by employees) which may have been filled in by customers.

What the Program Does

1. Displays a web form and tab panel.
2. Prompts users to fill in form values.
3. Stores form values in a database keyed by session (document) ID.
4. Generates and displays a PDF by merging a form template and input data.

The tab panel is paired with an inline frame which alternatively shows a web form or a PDF, depending on which tab is selected. Let's review the layout of the web form.

Form Template

The web form is based on an HTML template. The layout is as follows:

Form Template

```
<html>
<head>
<style>@import url("/rdweb/info3/iui19/fillin.css");</style>
</head>
<body>
<form action="post.shtml" method="post">

<center><h1>Official Certificate of Marriage</h1></center>
<center><h5>paragraph 50(1)(b) Marriage Act 1961</h5></center>
<p>Marriage was solemnised between the parties, details of whom are given below, on the <input
value="{{day}}" type="text" autofocus size="5" maxlength="5" name="day" placeholder="1st"> day
of <input value="{{month}}" type="text" size="10" maxlength="10" name="month"
placeholder="January">, 20 <input value="{{year}}" type="text" size="2" maxlength="2"
name="year" placeholder="17">, at <input value="{{location}}" type="text" size="82"
maxlength="90" name="location" placeholder="Location">.</p>
<table cellpadding="2px" cellspacing="0px">
<tr>
<td align="center" height="50px" class="hi-row">Detail</td>
<td align="center" class="hi-row">Bridegroom</td>
<td align="center" class="hi-row">Bride</td>
</tr>
<tr>
<td class="even">Surname</td>
<td><input value="{{surname_g}}" type="text" size="50" maxlength="50" name="surname_g"
placeholder="Surname"></td>
<td><input value="{{surname_b}}" type="text" size="50" maxlength="50" name="surname_b"
placeholder="Surname"></td>
</tr>
<tr>
<td class="even">Other Names</td>
<td><input value="{{other_g}}" type="text" size="50" maxlength="50" name="other_g"
placeholder="Other Names"></td>
<td><input value="{{other_b}}" type="text" size="50" maxlength="50" name="other_b"
placeholder="Other Names"></td>
</tr>
<tr>
<td class="even">Usual occupation</td>
<td><input value="{{occupa_g}}" type="text" size="50" maxlength="50" name="occupa_g"
placeholder="Usual occupation"></td>
<td><input value="{{occupa_b}}" type="text" size="50" maxlength="50" name="occupa_b"
placeholder="Usual occupation"></td>
</tr>
<tr>
<td class="even">Usual residence</td>
<td><input value="{{reside_g}}" type="text" size="50" maxlength="50" name="reside_g"
placeholder="Usual residence"></td>
<td><input value="{{reside_b}}" type="text" size="50" maxlength="50" name="reside_b"
placeholder="Usual residence"></td>
</tr>
```

```

<tr>
<td class="even">Conjugal status</td>
<td><input value="{{conjugal_g}}" type="text" size="50" maxlength="50" name="conjugal_g"
placeholder="conjugal status"></td>
<td><input value="{{conjugal_b}}" type="text" size="50" maxlength="50" name="conjugal_b"
placeholder="conjugal status"></td>
</tr>
<tr>
<td class="even">Birthplace</td>
<td><input value="{{birthpl_g}}" type="text" size="50" maxlength="50" name="birthpl_g"
placeholder="Birthplace"></td>
<td><input value="{{birthpl_b}}" type="text" size="50" maxlength="50" name="birthpl_b"
placeholder="Birthplace"></td>
</tr>
<tr>
<td class="even">Father's full name</td>
<td><input value="{{father_g}}" type="text" size="50" maxlength="50" name="father_g"
placeholder="Father's full name"></td>
<td><input value="{{father_b}}" type="text" size="50" maxlength="50" name="father_b"
placeholder="Father's full name"></td>
</tr>
<tr>
<td class="even">Mother's maiden</td>
<td><input value="{{mother_g}}" type="text" size="50" maxlength="50" name="mother_g"
placeholder="Mother's maiden name"></td>
<td><input value="{{mother_b}}" type="text" size="50" maxlength="50" name="mother_b"
placeholder="Mother's maiden name"></td>
</tr>
<tr>
<td class="even" height="50px" style="font-weight:bold;">Signatures</td>
<td class="bb">&nbsp;</td>
<td class="bb">&nbsp;</td>
</tr>
</table>
<table cellpadding="2px" cellspacing="0px">
<tr>
<td colspan="3" class="hi-row">Witnesses to the Marriage:</td>
</tr>
<tr>
<td class="even">Full Names</td>
<td><input value="{{witness_1}}" type="text" size="50" maxlength="50" name="witness_1"
placeholder="Witness full name"></td>
<td><input value="{{witness_2}}" type="text" size="50" maxlength="50" name="witness_2"
placeholder="Witness full name"></td>
</tr>
<tr>
<td class="even" style="font-weight:bold;" height="50px">Signatures</td>
<td class="bb">&nbsp;</td>
<td class="bb">&nbsp;</td>
</tr>
</table>

```

```
<p>I, <input value="{{performer}}" type="text" size="50" maxlength="50" name="performer"
placeholder="Performer full name"> certify that, on the date and at the place specified above, I duly
solemnised marriage in accordance with the provisions of the Marriage Act 1961 between the
parties specified above.</p>
<table cellpadding="2px" cellspacing="0px">
<tr>
<td>Signature</td>
<td width="60%" class="bb">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>
<td>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;<input value="{{date_perf}}" type="text" size="30px"
maxlength="50px" name="date_perf" placeholder="Date"></td>
</tr>
</table>
<p></p>
<table cellpadding="2px" cellspacing="0px" width="100%">
<tr>
<td width="50%">Performer number <input value="{{author_no}}" type="text" size="20"
maxlength="20" name="author_no" placeholder="Authorization Number">.</td>
<td width="50%" align="center">Marriage licence <input value="{{license}}" type="text" size="20"
maxlength="20" name="license" placeholder="License Number">.</td> </tr>
</table></form>
</body>
</html>
```

The layout consists of:

- An Image.
- A web form.
- Headers.
- Tables.
- Input elements.
- Paragraphs.
- Doubly curly brace markers (placeholders for data values).

Program IUI107

An RPG program named IUI107 handles the user interface and back-end processing. The code is as follows:

Program IUI107

```
//-----
// Marriage Certificates file
//-----

ffmar100  uf a e      k disk

//-----
// procedure prototypes
//-----

/copy *libl/qrpglesrc,rdstmapi#1
/copy *libl/qrpglesrc,rdsesapi#1
/copy *libl/qrpglesrc,rdwtnapi#1
/copy *libl/qrpglesrc,rd rptapi#1
```

```
//-----
// module level data
//-----

d rw      e ds      extname(rwpgmc) qualified

d s1      s      * inz(*null)
d stm1    s      * inz(*null)
d fmt1    s      * inz(*null)
d htm_file s      128a varying
d pdf_file s      128a varying

//-----
// program entry
//-----

c *entry  plist
c        parm      rw

/free

//-----
// set reference to UI template
//-----

if s1 <> *null;
  wtnSetInst(s1);
endif;

//-----
// branch to subroutines based on requested actions
//-----

select;
  when rw.action = 'INIT';
    exsr do_init;
  when rw.action = 'FILL';
    exsr do_fill;
  when rw.action = 'POST';
    exsr do_post;
endsl;

return;

//-----
// initialization
//-----

begsr do_init;
```

```

stmInit();

s1 = wtnOpen('IUI107');

fmt1 = stmFmtLoad('IUI107A');

endsr;

//-----
// update database record
//-----

begsr do_post;

key = sesId();

chain key fmar100r;

if %found();
  day = %trimr(wtnFldGet('day'));
  month = %trimr(wtnFldGet('month'));
  year = %trimr(wtnFldGet('year'));
  location = %trimr(wtnFldGet('location'));
  surname_g = %trimr(wtnFldGet('surname_g'));
  surname_b = %trimr(wtnFldGet('surname_b'));
  other_g = %trimr(wtnFldGet('other_g'));
  other_b = %trimr(wtnFldGet('other_b'));
  occupa_g = %trimr(wtnFldGet('occupa_g'));
  occupa_b = %trimr(wtnFldGet('occupa_b'));
  reside_g = %trimr(wtnFldGet('reside_g'));
  reside_b = %trimr(wtnFldGet('reside_b'));
  conjugal_g = %trimr(wtnFldGet('conjugal_g'));
  conjugal_b = %trimr(wtnFldGet('conjugal_b'));
  birthpl_g = %trimr(wtnFldGet('birthpl_g'));
  birthpl_b = %trimr(wtnFldGet('birthpl_b'));
  father_g = %trimr(wtnFldGet('father_g'));
  father_b = %trimr(wtnFldGet('father_b'));
  mother_g = %trimr(wtnFldGet('mother_g'));
  mother_b = %trimr(wtnFldGet('mother_b'));
  witness_1 = %trimr(wtnFldGet('witness_1'));
  witness_2 = %trimr(wtnFldGet('witness_2'));
  performer = %trimr(wtnFldGet('performer'));
  date_perf = %trimr(wtnFldGet('date_perf'));
  author_no = %trimr(wtnFldGet('author_no'));
  license = %trimr(wtnFldGet('license'));
  update fmar100r;
endif;

//-----
// generate names for HTML and PDF stream files based on session ID
//-----

```

```

htm_file = '/rdweb/info3/iui19/' + sesId() + '.html';
pdf_file = '/rdweb/info3/iui19/' + sesId() + '.pdf';

//-----
// generate HTML stream file
//-----

stmFmtInst(fmt1);

stm1 = stmFileOpen(htm_file);

stmRecSet('FILLIN');
stmVarSet('day':day);
stmVarSet('month':month);
stmVarSet('year':%trimr(year));
stmVarSet('location':location);
stmVarSet('surname_g':surname_g);
stmVarSet('surname_b':surname_b);
stmVarSet('other_g':other_g);
stmVarSet('other_b':other_b);
stmVarSet('occupa_g':occupa_g);
stmVarSet('occupa_b':occupa_b);
stmVarSet('reside_g':reside_g);
stmVarSet('reside_b':reside_b);
stmVarSet('conjugal_g':conjugal_g);
stmVarSet('conjugal_b':conjugal_b);
stmVarSet('birthpl_g':birthpl_g);
stmVarSet('birthpl_b':birthpl_b);
stmVarSet('father_g':father_g);
stmVarSet('father_b':father_b);
stmVarSet('mother_g':mother_g);
stmVarSet('mother_b':mother_b);
stmVarSet('witness_1':witness_1);
stmVarSet('witness_2':witness_2);
stmVarSet('performer':performer);
stmVarSet('date_perf':date_perf);
stmVarSet('author_no':author_no);
stmVarSet('license':license);
stmVarSet('doc_ref':sesId());
stmRecWrt('FILLIN');

//-----
// transform HTML file to PDF - stream PDF to browser
//-----

stmToPDF(60);

wtmHdrSet('content-type':'application/pdf');

if wtnQryGet('attach') = 'y';

```

```

wtnHdrSet('content-disposition':
    'attachment; filename=marriage.pdf');
endif;

rptToWeb(pdf_file:*off);

//-----
// close and delete temporary stream files
//-----

rptRmvFile(pdf_file);

stmFileClose(*off);

endsr;

//-----
// show the fill-in form
//-----

begsr do_fill;

clear fmar100r;

key = sesId();

chain(n) key fmar100r;

if not %found();
    write fmar100r;
endif;

//-----
// fill in the form from the DB record
//-----

wtnRecSet('FILLIN');
wtnFldSet('day':day);
wtnFldSet('month':month);
wtnFldSet('year':%trimr(year));
wtnFldSet('location':location);
wtnFldSet('surname_g':surname_g);
wtnFldSet('surname_b':surname_b);
wtnFldSet('other_g':other_g);
wtnFldSet('other_b':other_b);
wtnFldSet('occupa_g':occupa_g);
wtnFldSet('occupa_b':occupa_b);
wtnFldSet('reside_g':reside_g);
wtnFldSet('reside_b':reside_b);
wtnFldSet('conjugal_g':conjugal_g);
wtnFldSet('conjugal_b':conjugal_b);

```



```
wtnFldSet('birthpl_g':birthpl_g);  
wtnFldSet('birthpl_b':birthpl_b);  
wtnFldSet('father_g':father_g);  
wtnFldSet('father_b':father_b);  
wtnFldSet('mother_g':mother_g);  
wtnFldSet('mother_b':mother_b);  
wtnFldSet('witness_1':witness_1);  
wtnFldSet('witness_2':witness_2);  
wtnFldSet('performer':performer);
```

```
wtnFldSet('date_perf':date_perf);  
wtnFldSet('author_no':author_no);  
wtnFldSet('license':license);  
wtnRecWrt('FILLIN');
```

```
endsr;
```

```
/end-free
```

High-level explanation of program IUI107:

- Web form values are stored in a DB table named FMAR100, which is keyed by the user's session ID.
- At program initialization, two versions of the web form are read into memory (the version used to prompt for input contains <input> elements, while the one used to generate the PDF doesn't).
- A subroutine named *do_fill* is used to generate the web form and stream it to the browser.
- A subroutine named *do_post* stores input values in the FMAR100 table, generates a PDF, and streams it to the browser.
- PDF generation entails output to a couple of temporary stream files which are named based on the user's session ID.
- Procedure names beginning with "wtn" are used to generate browser output.
- Procedure names beginning with "stm" are used to generate IFS stream-file output.
- Procedure stmToPDF() converts the HTML stream file to PDF.

Wrapping Up

With some basic skills in HTML markup and ILE RPG you can create web forms and generate PDF documents based on input received from users.

Programs like this could be adapted to present partially filled in web forms, prompt only for input required to complete the forms, notify people who are parties to the document and provide links for them, etc.

Follow Up

I posted a reference to this article in a discussion on Midrange Lists which led to a perplexing exchange with Richard Schoen of Help Systems. It began at the end of December 2016 and carried into the first week in January 2017. He initially labeled my content as conceptual, synthetic, incomplete, and hardly useful. He repeatedly cajoled me to publish the black-box code behind the PDF generation - to put it in the public domain.

His sideswipes seemed even more incongruous after I took an opportunity to view and read numerous "resources" available at the Help Systems web site in regards to document management.

The content there IS abstract, embellished, riddled with promises and assurances, and leaves viewer/readers hanging with the intent of baiting them to discuss their interests with sales staff.

On a positive note, Mr. Schoen's criticisms did make me reflect more deeply about the content I had published, including pros and cons.

On the pro side, I think there is a valid message for organizations which design interactive PDF forms using Adobe Acrobat DC and publish them for people (i.e. customers) to fill out by using Adobe Reader. Although ubiquitous, that paradigm does not support placeholder prompts in input elements, validation, or appropriate feedback. The data keyed into Adobe Reader is typically re-keyed by employees in order to collect it into line-of-business systems. And there's typically no way to pre-fill form elements with data which may exist in a database.

On the con side, filling in web forms using hand-held devices (i.e. smart phones and tablets) is tedious when forms are designed for printing on 8.5 X 11.0 inch paper. The virtual keyboards on the devices overlay too much of the viewing area. This problem applies to web forms as well as Adobe Reader forms. On the whole, I would recommend designing web forms that adapt appropriately to hand-held devices (see my articles about "responsive design"), and use a separate HTML template for generating the PDF version.

Inputs and Outputs

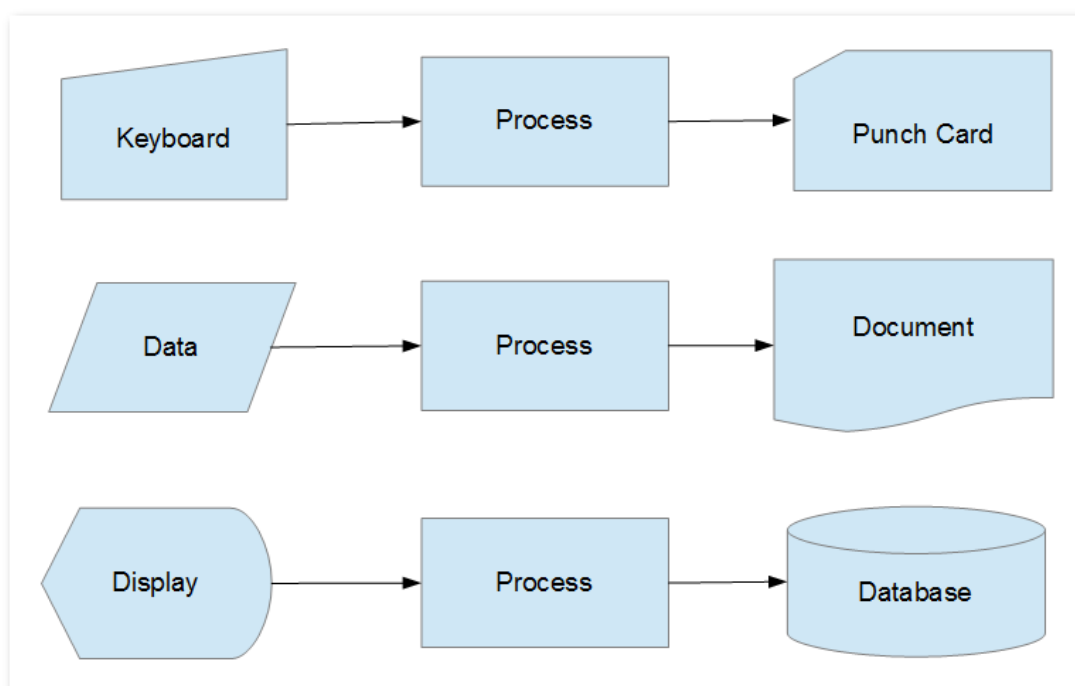
<https://rd.radile.com/rdweb/info2/ibmiui20.html>

This article pertains to the modernization of IBM i applications. But in this piece I return to more conceptual thoughts and opinions.

I've chosen a generic title for this piece (*Inputs and Outputs*) partly because it's an appropriate tag for the subject at hand and partly because I want to use unembellished terms and expressions. I'll explain the reason for that as I write.

Introduction

Pioneers in computing often included flow charts in their documentation and other writings that included various forms of input and output:



The elements depicted in that image are abstract representations of some historical input and output types.

Using object-oriented terminology, all sorts of inputs and outputs can be delineated, subclassed, extended, described, and discussed in a variety of ways. However, it generally requires discussing objects and their attributes in relation to a process in order to give them meaning.

Imagine a discussion or presentation about computing or developer tools, where the content entails few if any references to objects, their attributes, little or no material regarding whether an object is used for input or output, and little or no discussion about objects in relation to processes.

Say the discussion or presentation consists mostly of labels such as modern or legacy, right or wrong, better or worse, popular or old-fashioned, productive or time-consuming, secure or uncertain.

Intermix blocks of hyperbole throughout. Add to that - promises and assurances intended to lead you to pick a certain choice among alternatives.

What if one set of labels were applied to one choice while their antonyms were applied to another? Would you have the information you need in order to assess the suitability of the paradigm, process, or tool for your intended purposes? Would you be confused?

Contrast the previous hypothetical with one where the presentation or discussion incorporates flowcharts, including delineations of objects, their attributes, whether used for input or output, and their relation to a process.

Web Technologies

In the next three (3) sections, I'd like to show flow charts for designing and/or writing HTML templates, cascading style sheets, and JavaScript - and to offer a few opinions about the process and available tools that might assist with that.

First, I should digress to opine about the relevance of HTML, CSS, and JavaScript. A number of successful tool vendors who target the IBM i application space - they promote proprietary designers/editors, which generate proprietary alternatives to HTML, CSS, and JavaScript.

They opine that application developers should be shielded from such low-level implementation details and provided with tooling that saves time. Their tooling may generate proprietary XML or JSON stream files, which provide an interface with a proprietary utility instead.

The most obvious danger with the proprietary approach is the lock-in that occurs with the tooling and utilities.

Another concern is that the interface may be subject to constraints, which may not be obvious during the investigation or honeymoon phase of using it.

Such proprietary interfaces are prevalent with tools that facilitate document (i.e. forms) generation as well as interactive web applications. IBM announced that they will not be licensing their Advanced Function Printing (AFP) Utilities with IBM i release 7.3. Vendors of proprietary forms generation products were quick to offer alternatives.

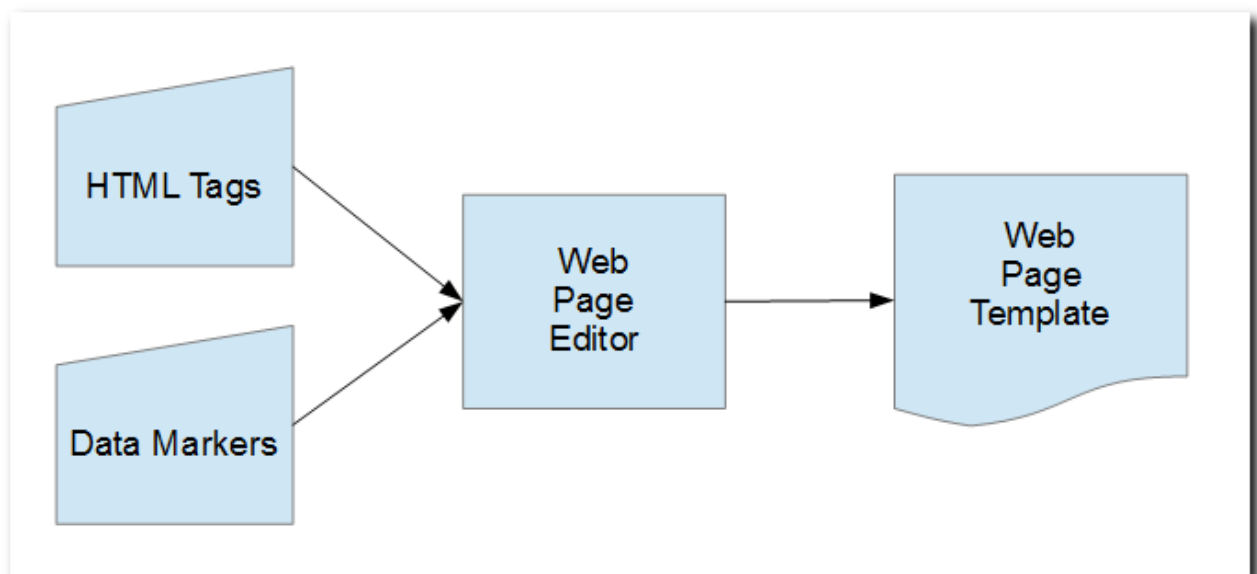
In my experience, and despite repeated promises and assurances from vendors to the contrary, the use of proprietary interfaces and utilities as I have described, embody unacceptable constraints, which become more evident over time. The time savings that were promised evaporate as developers struggle to adapt the tooling to unforeseen use cases.

Visual (WYSIWYG) designers are a common lure among vendors of proprietary interfaces and utilities. In that regard, I still stand by the assertions that I made in this modernization series in regards to responsive design. Visual designers, which generate precise positioning of form elements, are unable to automatically adapt to smaller hand-held devices (i.e. smart phones and tablets).

HTML, cascading style sheets, and JavaScript avoid the lock-in and limitations found in proprietary interfaces, tooling, and utilities.

HTML Templates

The following flow chart shows an HTML template, output from an editor. Inputs are standard HTML elements (aka "tags") and "Data Markers", which I'll define simply as *names*, which are delimited by curly braces `{{...}}`. Where do the tags come from? A W3C spec.



I've worked with a number of HTML editors. The most important feature in my opinion is having code prompting - providing for the selection of tag names from an up-to-date reference.

Some HTML editors include drag-and-drop widget palettes, property sheets, dialogs for generating HTML content, and visual design views. But even beginners tend to outgrow such interfaces fairly quickly.

Rather than designing HTML documents using a visual editor, it is quite a bit more productive to begin new projects by copying templates or smaller blocks of code from previous projects.

HTML Template

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<p>Hello {{name}}.</p>
</body>
</html>
```

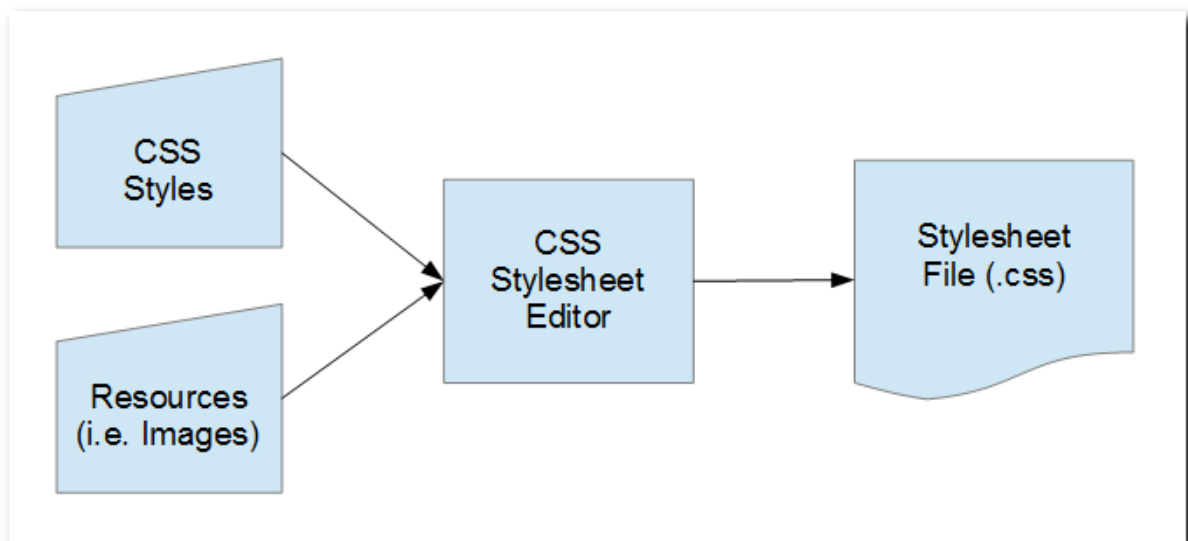
Many HTML editors provide options for pre-viewing pages in standard browsers. The design phase often entails toggling between editor and browser windows.

Even experienced HTML authors may toggle between the editor and say the W3 Schools web site, which provides both tutorials and references that can assist with authoring templates.

I view such interfaces as a benefit of picking standard technologies for document and template layout. Developers have options. Individual preferences, tooling, references, and helps may evolve over time.

Cascading Style Sheets

The following flowchart shows a cascading style sheet being output from an editor. Inputs include CSS attributes and resources such as images, which may be assigned as background properties.



As with HTML editing, look for an editor that supports automatic code prompting based on a built-in reference. Likewise, keep CSS tutorials and references open in a browser window.

CSS Example

```

input {
  font-family: "Palatino Linotype", "Book Antiqua", Palatino, serif;
  font-weight:bold;
  font-size:14px;
  border-left:none;
  border-right:none;
  border-top:none;
  text-align:center;
}
  
```

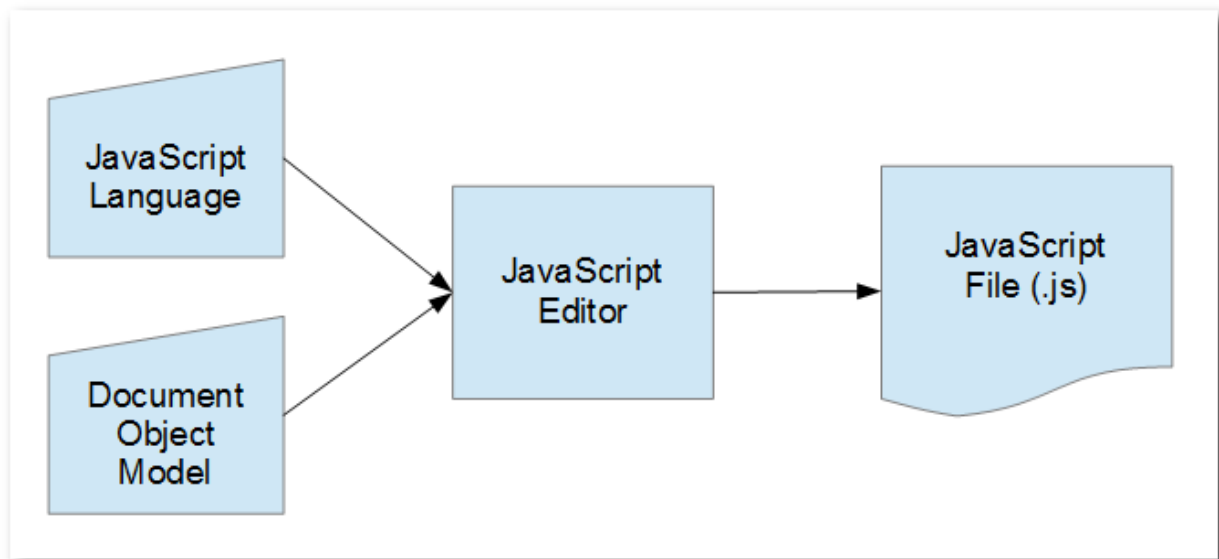
Some CSS editors display background colors and other styling in a separate panel as authors navigate from line to line in a .css file.

When defining a .css file, which is intended to be used as a standard across a variety of screens or printed documents, authors may supplement a CSS editor with say a color wheel, which is a great help for selecting complimentary foreground and background colors.

JavaScript

The following flowchart shows a JavaScript file, output from an editor. Syntax checking and error highlighting top my feature priorities for editors. Code completion is nice.

I included the browser document object model (DOM) as an input in the diagram. The DOM may not be part of the JavaScript specification, but most JavaScript deployed to run in browsers is invoked from DOM events, and is used to update DOM elements.



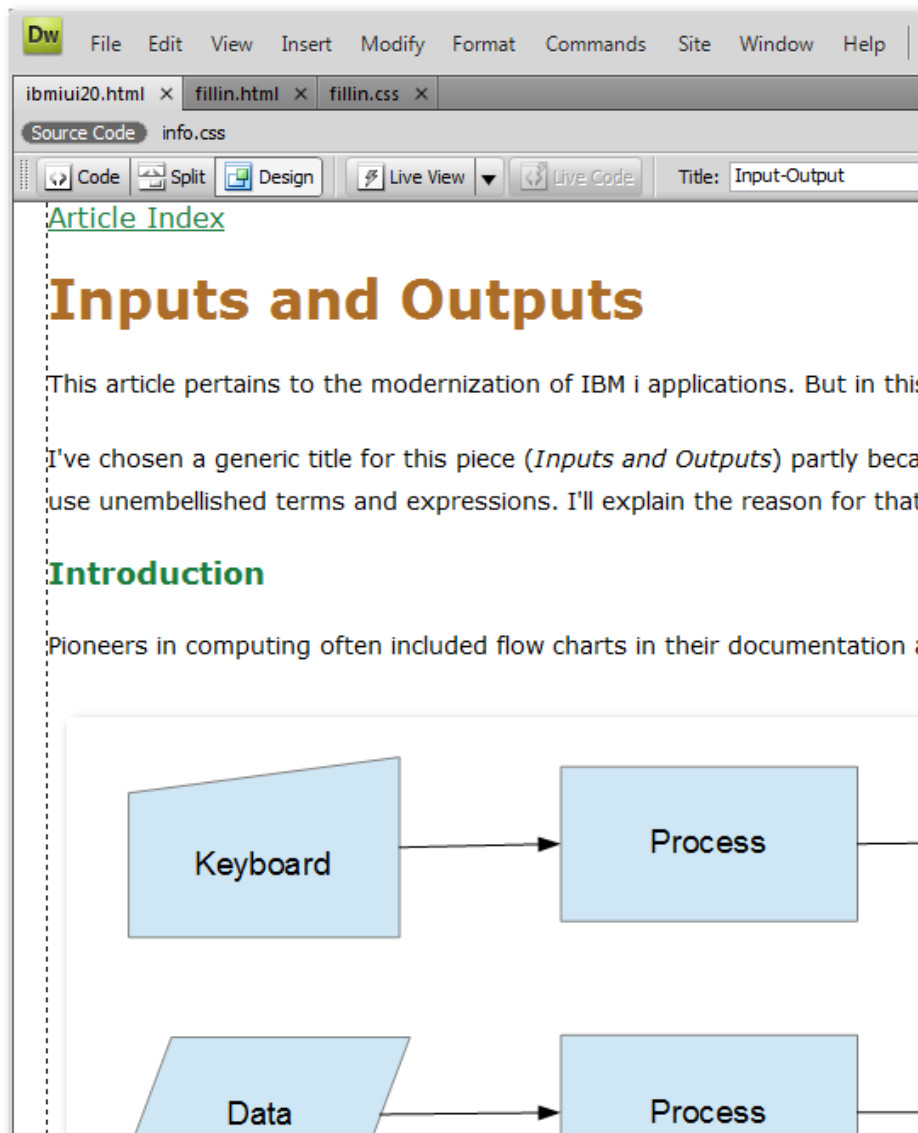
In addition to composing and editing JavaScript, developers become proficient with browser developer tools, which provide for monitoring variable values and debugging their logic.

JavaScript Example

```
function win_resize() {  
  var dl = document.getElementById('dl');  
  dl.style.height = dl.parentElement.clientHeight - dl.offsetTop + 0;  
}
```

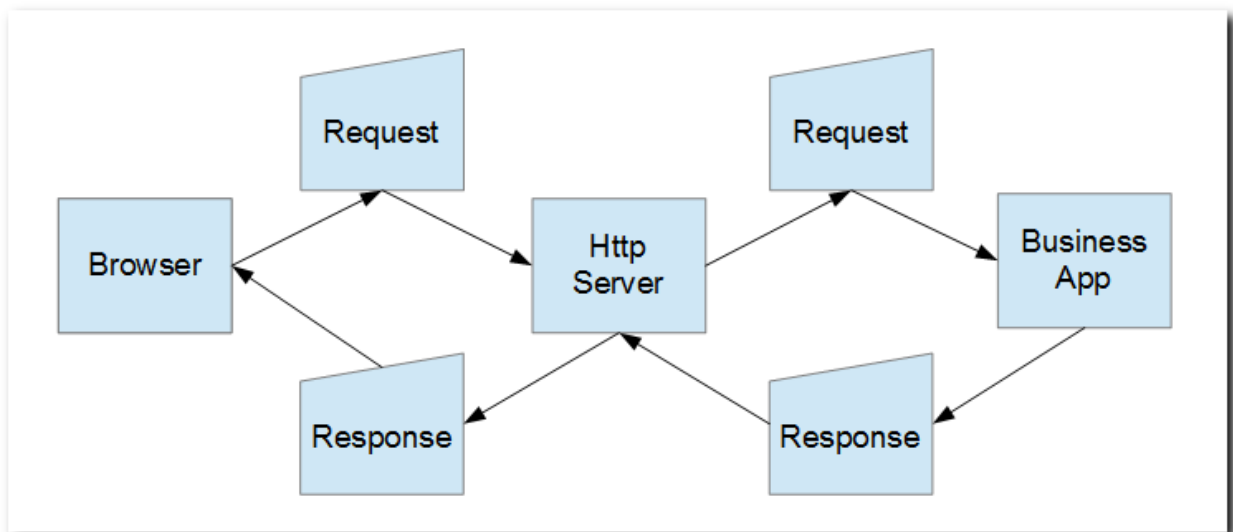
Hopefully, I've provided a helpful overview of "flows" that may be entailed in the UI design process. Whether composing or editing HTML, CSS, or JavaScript, I recommend that developers pick tools that adhere closely to standards. Pick tools that meet individual preferences and supplement them with tutorials, references, helps, and utilities, which are available in the public domain.

The following screen shot shows the design surface of Adobe Dreamweaver - a popular tool for editing HTML, CSS, and JavaScript.



Web Request-Response Cycle

Depending on the platforms, languages, developer tools, and resources selected for web application development, beginners may feel overwhelmed by the interfaces or anxious over the complexities. To obviate that, I like to recall the request-response cycle, which is the essence of all interactive web applications.

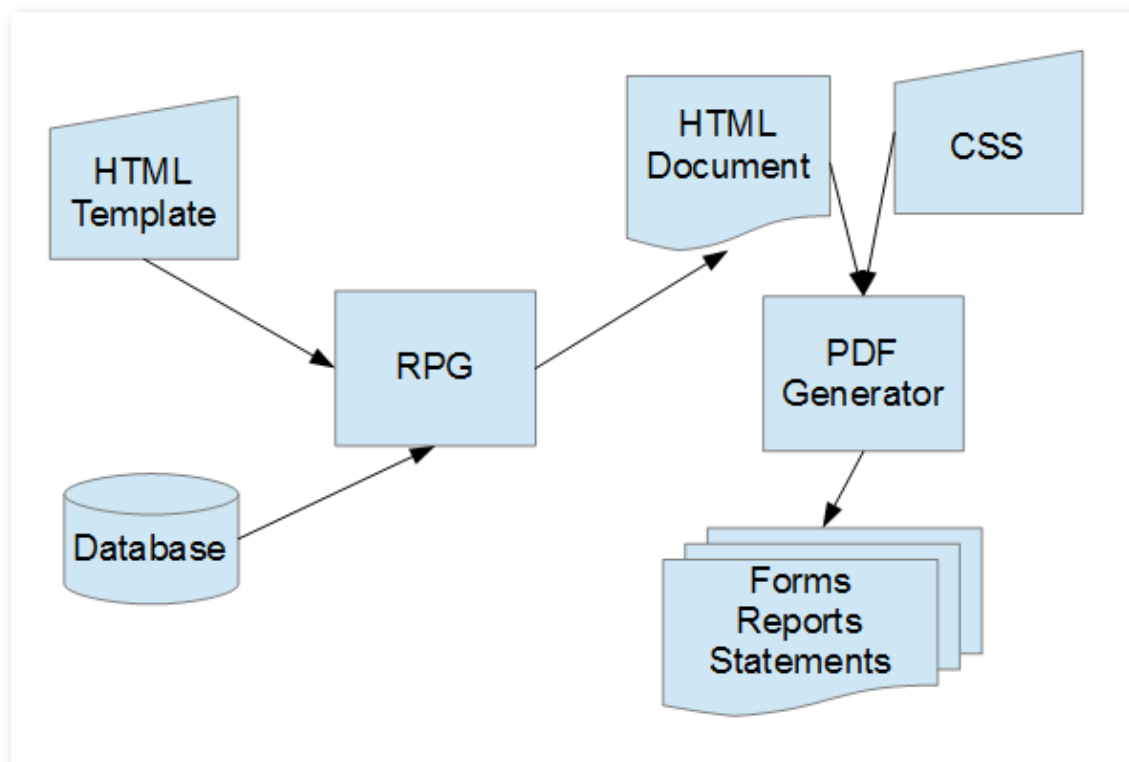


- Browsers send requests for resources to HTTP servers.
- HTTP servers forward requests to business applications.
- Business applications generate formatted streams (responses) and return them to HTTP servers.
- HTTP servers return responses to browsers.

The nature and attributes of requests were delineated in [Part 5](#). Responses generally are formatted as HTML, XML, or JSON streams, which may include references to static resources such as .css, .js, and .png files.

Generating Reports, Forms, and Documents

HTML and cascading style sheets are excellent, standard technologies, which may be used for generating stylized content, suitable for printing as shown in the following diagram.





- An ILE RPG program may extract data from a database and merge it with an HTML template to generate an HTML stream file.
- A Utility can transform an HTML stream file into PDF output - referencing one or more CSS style sheets to apply formatting and styling for the output.

The tools that developers pick for designing HTML templates and CSS style sheets can be used to generate nicely formatted, stylized output, suitable for printing, as an alternative to using proprietary interfaces and tools, packaged in forms products (i.e. Advanced Function Printing and associated design utilities, etc.).

Wrapping Up

Prior to my writing this piece, I viewed webinars and read articles that featured the wares of a number of vendors in the IBM i modernization space. Much of the content was so embellished and the solutions so overly hyped that I wished for depictions or diagrams to end my disorientation.

That led me to the idea of writing a piece and including flowcharts - depicting *inputs, outputs, and processes*. I hope this has helped.