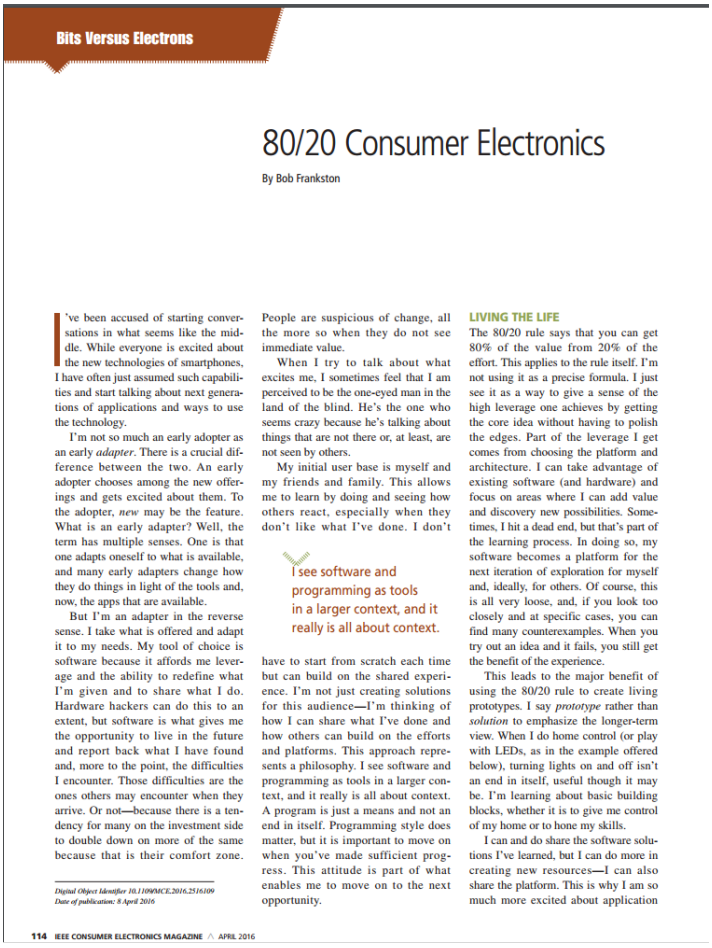
80/20 Consumer Electronics

Online Edition



Bits Versus Electrons

80/20 Consumer Electronics

By Bob Frankston

I've been accused of starting conversations in what seems like the middle. While everyone is excited about the new technologies of smartphones, I have often just assumed such capabilities and start talking about next generations of applications and ways to use the technology.

I'm not so much an early adopter as an early *adapter*. There is a crucial difference between the two. An early adopter chooses among the new offerings and gets excited about them. To the adopter, *new* may be the feature. What is an early adapter? Well, the term has multiple senses. One is that one adapts oneself to what is available, and many early adapters change how they do things in light of the tools and, now, the apps that are available.

But I'm an adapter in the reverse sense. I take what is offered and adapt it to my needs. My tool of choice is software because it affords me leverage and the ability to redefine what I'm given and to share what I do. Hardware hackers can do this to an extent, but software is what gives me the opportunity to live in the future and report back what I have found and, more to the point, the difficulties I encounter. Those difficulties are the ones others may encounter when they arrive. Or not—because there is a tendency for many on the investment side to double down on more of the same because that is their comfort zone.

Digital Object Identifier 10.1109/MCE.2016.2516109
Date of publication: 8 April 2016

People are suspicious of change, all the more so when they do not see immediate value.

When I try to talk about what excites me, I sometimes feel that I am perceived to be the one-eyed man in the land of the blind. He's the one who seems crazy because he's talking about things that are not there or, at least, are not seen by others.

My initial user base is myself and my friends and family. This allows me to learn by doing and seeing how others react, especially when they don't like what I've done. I don't have to start from scratch each time but can build on the shared experience. I'm not just creating solutions for this audience—I'm thinking of how I can share what I've done and how others can build on the efforts and platforms. This approach represents a philosophy. I see software and programming as tools in a larger context, and it really is all about context. A program is just a means and not an end in itself. Programming style does matter, but it is important to move on when you've made sufficient progress. This attitude is part of what enables me to move on to the next opportunity.

I see software and programming as tools in a larger context, and it really is all about context.

LIVING THE LIFE

The 80/20 rule says that you can get 80% of the value from 20% of the effort. This applies to the rule itself. I'm not using it as a precise formula. I just see it as a way to give a sense of the high leverage one achieves by getting the core idea without having to polish the edges. Part of the leverage I get comes from choosing the platform and architecture. I can take advantage of existing software (and hardware) and focus on areas where I can add value and discovery new possibilities. Sometimes, I hit a dead end, but that's part of the learning process. In doing so, my software becomes a platform for the next iteration of exploration for myself and, ideally, for others. Of course, this is all very loose, and, if you look too closely and at specific cases, you can find many counterexamples. When you try out an idea and it fails, you still get the benefit of the experience.

This leads to the major benefit of using the 80/20 rule to create living prototypes. I say *prototype* rather than *solution* to emphasize the longer-term view. When I do home control (or play with LEDs, as in the example offered below), turning lights on and off isn't an end in itself, useful though it may be. I'm learning about basic building blocks, whether it is to give me control of my home or to hone my skills.

I can and do share the software solutions I've learned, but I can do more in creating new resources—I can also share the platform. This is why I am so much more excited about application

114 IEEE CONSUMER ELECTRONICS MAGAZINE ^ APRIL 2016

You can read this article online. Note that that version is missing the pointer to the source code of the blink example. You can find it here.

Other columns are available here.

Software

I've been accused of starting conversations in what seems like the middle. While everyone is excited about the new technologies of smartphones I have often just assumed such capabilities and start talking about next generations of applications and ways to use the technology.

I'm not so much an early adopter as an early *adapter*. There is a crucial difference between the two. An early adopter chooses among the new offerings and gets excited about them. To the adopter, *new* may be the feature.

What is an early adapter? Well, the term has multiple senses. One is that one adapts oneself to what is available and many early adapters change how they do things in light of the tools and, now, the apps, that are available.

But I'm an adapter in the reverse sense. I take what is offered and adapt them to my needs. My tool of choice is software because it affords me leverage and the ability to redefine what I'm given and to share what I do.

Hardware hackers can do this to an extent but software is what gives me the opportunity to live in the future and report back what I have found and, more to the point, the difficulties I encounter. Those difficulties are the ones others may encounter when they arrive.

Or not. Because there is a tendency for many on the investment side to double down on more of the same because that is their comfort zone. People are suspicious of change, all the more so when they do not see immediate value.

When I try to talk about what excites me I sometimes feel that I am perceived to be the one-eyed man in the land of the blind. He's the one who seems crazy because he's talking about things that are not there or, at least, are not seen by others.

My initial user base is myself and my friends and family. This allows me learn by doing and seeing how others react, especially when they don't like what I've done. I don't have to start from scratch each time but can build on the shared experience. I'm not just create solutions for this audience. I'm thinking of how I can share what I've done and how others can build on the efforts and platforms.

This approach represents a philosophy. I see software and programming as tools in a larger context. And it really is all about context. A program is just a means and not an end in itself. Programming style does matter but it is important to move on when you've made sufficient progress. This attitude is part of what enables me to move on to the next opportunity.

Living the Life

The 80/20 rule says that you can get 80% of the value from 20% of the effort. This applies to the rule itself. I'm not using it as a precise formula. I just see it as a way to give a

sense of the high leverage one achieve get by getting the core idea without having to polish the edges

Part of the leverage I get comes from choosing the platform and architecture. I can take advantage of existing software (and hardware) and focus on areas where I can add value and discovery new possibilities. Sometimes I hit a dead end but that's part of the learning process. And in doing so my software becomes a platform for the next iteration of exploration for myself and, ideally, for others.

Of course, this is all very loose and, if you look too closely and at specific cases you can find many counter examples. When you try out an idea and it fails you still get the benefit of the experience.

This leads to the major benefit of using the 80/20 rule to create living prototypes. I say prototype rather than solution to emphasize the longer term view. When I do home control (or play with LEDs as in the example offered below) turning lights on and off isn't an end in itself, useful though that may be. I'm learning about basic building blocks, whether it is to give me control of my home or to hone my skills.

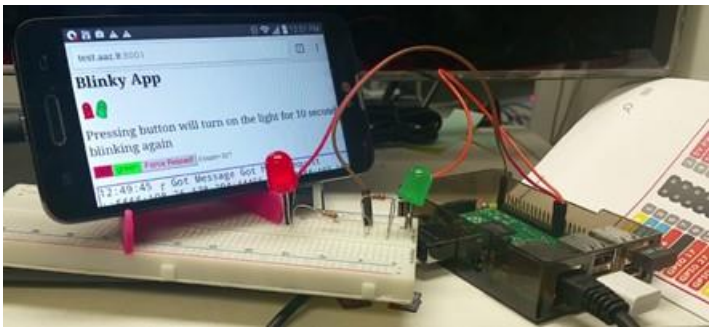
I can and do share the software solutions I've learned but I can do more in creating new resources. I can also share the platform. This is why I am so much more excited about APIs (Application Program Interfaces) then apps. APIs give anyone the ability to create their own solutions. Apps are wonderful but they should be built on open APIs -- **#APIFirst**. Though only a few will take advantage of this opportunity, the abundance they create benefits us all. History is replete with key innovations that transformed society.

The concepts of software can also be applied to hardware. Here too I look at hardware in terms of generative capabilities. Classic Lego™ blocks are a good example of hardware as a building block. The blocks themselves are very generic and the user decides what the function is. Recently Lego has produced blocks that are designed for specific themes and stories. With generic blocks I can build anything. If I depend on purpose-built parts, I'm both limited in what I can do with them and dependent upon what Lego chooses to make.

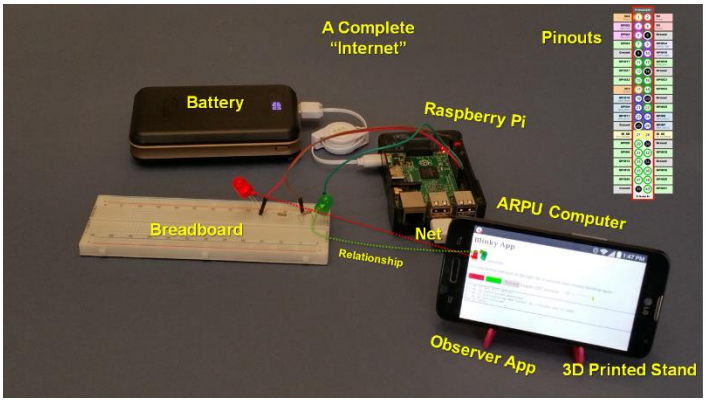
Back to Basics

I can easily get very philosophical and abstract so to keep myself honest I jump back to basics. Rather than going all the way to using a soldering iron and building from scratch I look for where I get leverage.

I've written about repurposing smart phones but choose to use the HTML5 environment rather than writing to the bare hardware. As wonderful as smartphones are they are generally designed to limit what applications can do in order to assure the integrity of the system.



This is why I've been experimenting with a Raspberry Pi from Adafruit which comes with full operating system support while still giving me access to the raw hardware.



The computer itself costs about $30 plus some money for an enclosure It's fairly simple to turn on and off the pins that control LEDs or other devices. The following code turns on a light by setting the corresponding pin to low:

```
pin.write(Gpio.GpioPinValue.low);
```

What may be surprising is that I'm running Windows 10, or to be more specific a version for devices (https://www.windowsondevices.com). The reason is that I can use powerful development tools so I can write programs on my main computer using Visual Studio and have my local development environment automatically deploy the applications while giving me the ability to debug the applications as if they were running on my desktop.

The application is written in TypeScript (http://TypeScriptLang.org) which is essentially JavaScript with annotations for the IDE (Integrated Development Environment). This means the development tools can assist in my writing the programs because, unlike plain JavaScript, I can annotate a

variable and mark it as a string and I will be warned if I assign a number. I still have the full capabilities and flexibility of JavaScript.

A big advantage is that I can detect many of my programming bugs at design time rather than having to discover them long after the code has been released. It also saves me a lot of time by allowing the IDE to provide me with hints. If I want to read a file it will provide me with the list of the methods (APIs) I can use for files as well as the parameters so I don't need to keep looking at manuals and help files. (See Figure Figure 1Hinting in the IDE)

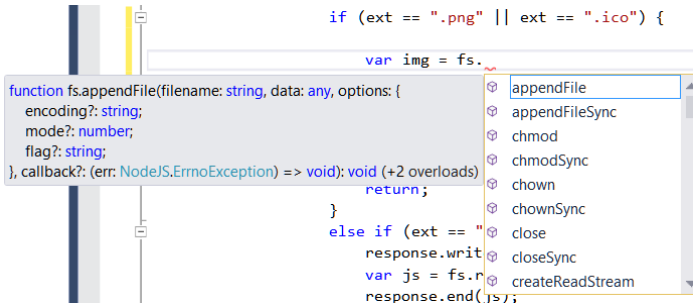


Figure 1Hinting in the IDE

I run the programs using Node.JS which is a JavaScript environment that can run on essentially any platform and includes tools for writing my own web server. There are also libraries of extensions so I can use features such as WebSockets. Since these libraries are available in source code I can also learn more by reading the code.

My test application is a simple web server with two blinking lights and it allows me to turn either one on for a period of time. The web client shows the dynamics status of the LEDs.

The sample programs offered me a good starting point. There was a server that blinked lights in C# and a sample code in JavaScript. In the end I had written a simple web server that runs in **node.js** (http://notejs.org) – a JavaScript environment that doesn't depend on a browser.

To give a sense of how simple it is to write a web server – this was my initial complete web server that reports the current value of the LED.

```
http.createServer((req, resp) => {
  resp.writeHead(200,
      { "Content-Type": "Text/plain" });
  resp.end(`Got LED ${currentValue}`);
```

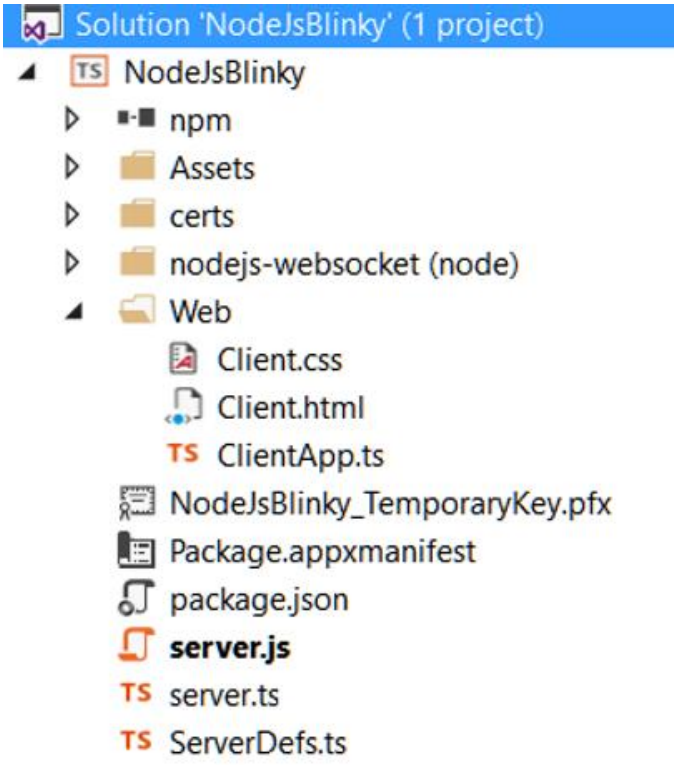
When I say complete, I mean that's it. No other support code. No Apache or other code.

For the purist, this version does use an http library. I had an early version of the server that used IP Sockets directly. It didn't require many more lines of code.

Turning on an LED is also simple – just a call to the `write` method.

The code tree consists of:

- **npm** The Node.js modules.
- **Assets** The image files for the bulbs
- **Certs** Certificates for https
- **WebSocket** The node module from GitHub which implements web sockets.
- **Web** The directory containing the html and code that runs on the client
- And, the server-side code.



I did enhance the server code and added support for https so I could practice using certificates for more serious applications. There is no web server as such. The node program reads the `Client.html` file. In the absence of a server I simply read the file and send it in response to the request. It's similar to the above example except the code is marked as HTML and the file is sent the `resp.end` call.

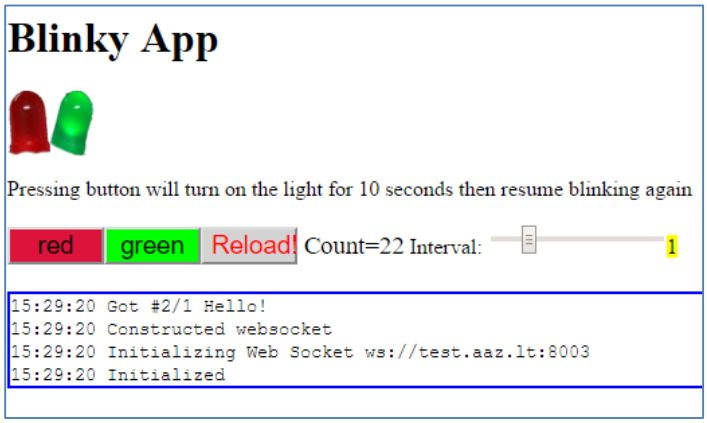
Rather than implementing a full web responder I insert the code the client code (`ClientApp.js`) into the file before I send it down. The client code is actually written in Typescript (just like the server code) and the ClientApp.js is

generated automatically by the IDE from `ClientApp.ts`. To avoid confusion, the client files (html and code) are kept in the Web directory. But both client and server share the definition of messages structures. This ability to share is one advantage of using the same language on both the client and the server.

The images of the bulbs (pictures I took of the bulbs themselves) are also inserted into the html file before being sent to the client.

When I started writing this article I had taken an expedient approach of reading all components files (client.html, ClientApp.js and the images) and combining them into one file which I then sent to the client. The approach wasn't very different from earlier web servers that mixed code and HTML to generate the page.

Today the norm is to separate code from markup (HTML) and data and use `src=` to combine the files on the client. It turned out to be very easy to use the URL parser that was part of node's http package and then to handle the requests. Since it is my own code I don't treat the paths as file system names which it's safer than using a generic browser. All told it took about 10 lines of code and, because of the relatively clean architecture of the approach it only involved local changes to the code.



As I continue to tweak and enhance the code I gain a better understanding of not just how web servers work but why they work in particular ways. For example, I found that I needed to look at the file type to know how to handle the file which is a simple version of the mime mapping in web services.

Rapid Innovation

By the time you read this article I'm sure that the app will be much more advanced as I learn and explore. This is in keeping with my general approach to implementation – always have a working application that can act as scaffolding for the next stage. This is a powerful dynamic that is feasible thanks to the ability to quickly innovate with software.

This is sharply different from designing systems based on hardware that requires special silicon that can take years to perfect.

With software we can learn by doing. By having a program that not only works but is useful at any stage we get the benefit from what it does and are constantly checking against reality rather than a rigid specification. My goal is to have prototypes that are capable of by used by others so I can get onto the next project.

VisiCalc was an example. I evolved the initial prototype to a shipping product. This approach was an important part of its success because early users were using a real product. We see this in Google's insistence that it's products are in beta testing even when used by millions of people.

This thinking also applies to how I view an Internet of Things. I'm not so concerned with a one-off solution such as the carefully engineered automated home. The bigger opportunity enabled by the Internet's common infrastructure for exchanging packets is our ability to interconnect devices without prior planning or careful coordination.

Today much of what we call IoT is embedded systems design using embedded computing to add intelligence to devices. The future is going to be much more open as awareness of how to mix and match devices increases and, for some people, how to add their own programming.

Peering

The client application runs in an HTML5 browser. It establishes a WebSocket connection with the host on the Raspberry Pi. All subsequent exchanges between the two endpoints are done using the socket as a peer connection. Though technically the Raspberry Pi is acting as a web server and the application running in the browser is a client I want to think of in terms of a peer architecture.

Client and server are roles in the relationship between the two endpoints but the basic Internet protocols are fundamentally peer protocols. The asymmetry of the web protocols reinforces the narrative of the Internet as something we access. We do access web sites but that's a way we use the Internet not what defines it.

Things have peer relationships with requests and responses. Once I've established the initial connection using http I can

continue with a peer relationship

For those interested in the details:

Exchanges are done using JSON (as noted in my #APIFirst article http://rmf.vc/IEEEAPIFirst). The red/green lights blink to show the state of the physical lights and are, in fact, pictures I took of the bulbs themselves in the on and off states! Not entirely necessary but the project should be fun too and it's a cute touch.

The input slider is used to set the interval for blinking. When there is a change that change is sent to all the observers (browsers). The server doesn't fully trust the clients so checks the blink rate value to make sure that is in range.

One of the dangers is that it is so easy to add features and make the program very powerful. Even in its simplest form it supports an arbitrary number of connections and can report the status to all listeners. The client side is resilient so that if the connection is lost it simply reconnects. It automatically listens to multiple IP address (both V4 and V6) by default.

The server itself is a full operating system so I can run multiple application as well as connect to the machine to manage it. It comes with a fair number of drivers out of the box. I can connect to it using command lines or via a web-based management screen.

I chose to use Windows because of the tools available but node apps can also run on Linux though I may need a slightly different supporting library for accessing the hardware pins.

Form Matters

One interesting option for the Raspberry Pi is a Lego-compatible case. Lego did Mindstorms nearly two decades ago but the computing engine wasn't very powerful which was one limitation on what one could do with the device. We can now have very powerful processing and I/O capabilities.

Form can be an essential part of the function of a device but software can evolve faster. This is why I will first try to repurpose a smartphone and see how far I can push it.

The converse happens when we take an existing appliance such as a refrigerator and add capabilities using software. In those case the form does define the function and role. But that is changing as users can start to repurpose devices and reinvent them. As I wrote in http://rmf.vc/IEEEDeconstructingTV this trend is already well along in consumer electronics.

3D printing holds the promise of allowing us to experiment with form. We are only at the earliest stages of that journey. If you look closely at the picture of the Raspberry pi, you may notice that the cell phone with the browser is sitting on a 3D-printed base.

Connectivity

Programming at this level strips away a lot of the mystery surrounding web servers and also shows how simple they can be. A commercial web server is far more complicated but the basic protocol is very simple and it is important to understand that.

The importance of the Internet approach is that it allows us to focus on the two endpoints in the relationship – the server and the client. I use HTTP (HyperText Transport Protocol) with a URL specifying the web address and parameters. And that's all. I don't need to think about anything between two endpoints!

That's the architecture essentially to enabling connected things. The reality isn't quite there because the focus has been on building support for high value applications services rather than architecture for connected things that aren't tied to a particular business offering or solution.

This further contributes to the narrative that the Internet is another telecommunications service which is why we accept having have a separate account per device for our smart phones, needing to "login" to the Internet and pressing "agree" to connect.

When I'm building and connecting my own things I'm back to the origins of the Internet when we were interconnecting our own computers rather than accessing services. There were no security perimeters because we just using our own wires and radios. Each program took responsibility for its own safety and systems like Multics were designed from scratch to provide us the tools for manage trust.

I'm striving to preserve fundamental peer architecture in my current approach. I can connect to my device locally by using its local name and a port number. Because the local network is behind a NAT the device would not normally be accessible outside the house so I had to map an external port to an internal port. I then use http://dyn.com/dns to automatically update my DNS name so that my site is reachable without needing a static IP address.

Connecting our own things forces us to confront the access narrative and is starts on the path to rediscovering connectivity.

Roots, Back to

Part of the fun is returning to my roots. The very first computer I used in 1963 was an IBM 1620 that was orders of magnitude slower but I had hands on access. I then moved onto more powerful machines as we worked on creating what was the called the information utility. But when I got a chance to play with my own 6502 based computer I couldn't resist and that practice served me well in when it came to implementing VisiCalc.

When we developed VisiCalc it took Dan Bricklin a few days to create a very limited mockup in Basic. It took me almost two months to implement enough of VisiCalc for Dan to use it in a class. I hadn't quite gotten division working yet but it was still useful. It then took us the better part of a year to go from working prototype to shipping prototype.

I say shipping prototype because you learn a lot more from a prototype that you really use than from a mockup. In fact, my style tends to implement a working prototype and then flesh out those pieces as needed.

This isn't necessarily feasible in a large project but, then again, isn't that the basis for agile programming? The more functional the intermediate version the more you know you're on track for shipping.

But, for the most part, I stick with that first 20% of the effort that gets me 80% of the value because there are so many other projects I need to do. Need? Well, feel compelled to do even if there aren't a myriad of deadlines.

The key is to not only complete the project but add to the future value by learning and creating building blocks I can share. That is one of the virtues of open software and sites like GitHub.

Learned & Learning by Doing & Using

While blinking lights are a fun home control application that forces me to think outside the one box and understand more about interconnected devices. That experience first made me understand the limitations of the current Internet protocols and why automation is the wrong meme. But that's another column ...

If you're curious you can find a version of the application here. It's not the most polished of code ... only about 20% of what it could be.