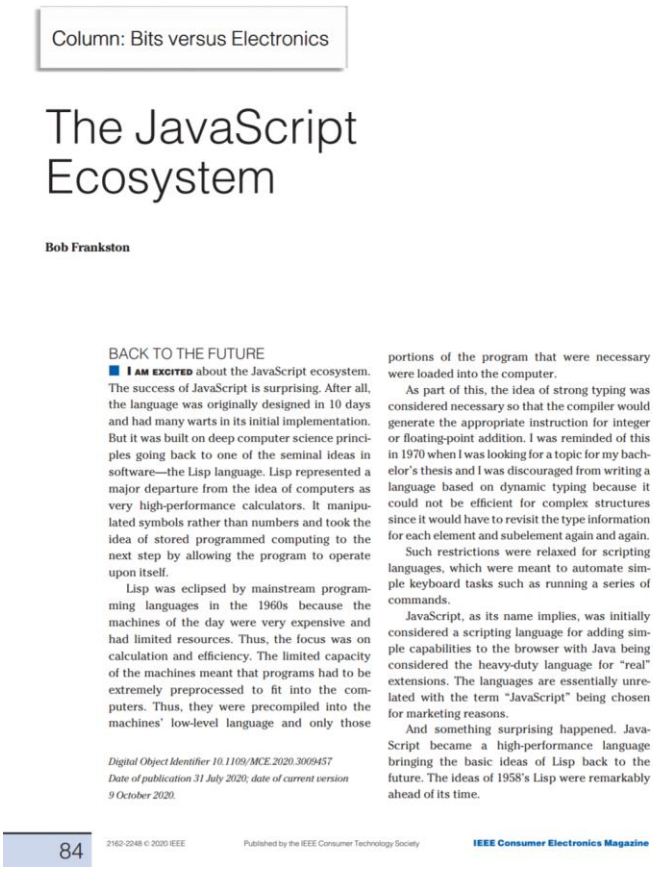
The JavaScript Ecosystem



Column: Bits versus Electronics

The JavaScript Ecosystem

Bob Frankston

BACK TO THE FUTURE

I AM EXCITED about the JavaScript ecosystem. The success of JavaScript is surprising. After all, the language was originally designed in 10 days and had many warts in its initial implementation. But it was built on deep computer science principles going back to one of the seminal ideas in software—the Lisp language. Lisp represented a major departure from the idea of computers as very high-performance calculators. It manipulated symbols rather than numbers and took the idea of stored programmed computing to the next step by allowing the program to operate upon itself.

Lisp was eclipsed by mainstream programming languages in the 1960s because the machines of the day were very expensive and had limited resources. Thus, the focus was on calculation and efficiency. The limited capacity of the machines meant that programs had to be extremely preprocessed to fit into the computers. Thus, they were precompiled into the machines' low-level language and only those portions of the program that were necessary were loaded into the computer.

As part of this, the idea of strong typing was considered necessary so that the compiler would generate the appropriate instruction for integer or floating-point addition. I was reminded of this in 1970 when I was looking for a topic for my bachelor's thesis and I was discouraged from writing a language based on dynamic typing because it could not be efficient for complex structures since it would have to revisit the type information for each element and subelement again and again.

Such restrictions were relaxed for scripting languages, which were meant to automate simple keyboard tasks such as running a series of commands.

JavaScript, as its name implies, was initially considered a scripting language for adding simple capabilities to the browser with Java being considered the heavy-duty language for "real" extensions. The languages are essentially unrelated with the term "JavaScript" being chosen for marketing reasons.

And something surprising happened. JavaScript became a high-performance language bringing the basic ideas of Lisp back to the future. The ideas of 1958's Lisp were remarkably ahead of its time.

Digital Object Identifier 10.1109/MCE.2020.3009457
Date of publication 31 July 2020; date of current version 9 October 2020.

84 2162-2248 © 2020 IEEE Published by the IEEE Consumer Technology Society IEEE Consumer Electronics Magazine

Back to the Future

I'm excited about the JavaScript ecosystem. The success of JavaScript is surprising. After all, the language was originally designed in 10 days and had many warts in its initial implementation. But it was built on deep computer science principles going back to one of the seminal ideas in software – the Lisp language. Lisp represented a major departure from the idea of computers as very high-performance calculators. It manipulated symbols rather than numbers and took the idea of stored programmed computing to the next step by allowing the program to operate upon itself.

Lisp was eclipsed by mainstream programming languages in the 1960ords because the machines of the day were very expensive and had limited resources. Thus, the focus was on calculation and efficiency. The limited capacity of the machines meant that programs had to be extremely preprocessed to fit into the computers. Thus, they were precompiled into the machines' low-level language and only those portions of the program that were necessary were loaded into the computer.

As part of this, the idea of strong typing was considered necessary so that the compiler would generate the appropriate instruction for integer or floating-point addition. I was reminded of this in 1970 when I was looking for a topic for my bachelor's thesis and I was discouraged from writing a language based on dynamic typing because it couldn't be efficient for complex structures since it would have to revisit the type information for each element and sub-element again and again.

Such restrictions were relaxed for scripting languages which were meant to automate simple keyboard tasks such as running a series of commands.

JavaScript, as its name implies, was initially considered a scripting language for adding simple capabilities to the browser with Java being considered the heavy-duty language for "real" extensions. The languages are essentially unrelated with the term "JavaScript" being chosen for marketing reasons.

And something surprising happened. JavaScript became a high-performance language bringing the basic ideas of Lisp back to the future. The ideas of 1958's Lisp were remarkably ahead of its time.

One nice thing about the heritage of Lisp is the orthogonality – the rich set of mechanisms are built on a simple base. While there is an emphasis on performance on high end browsers, it also allows for compact implementations where appropriate.

How and Why JavaScript

In 2014, I wrote about the growing importance of HTML5[i]. Writing applets in Java was problematic because the apps had full access to the host computing environment. Confining JavaScript to the browser meant the code was portable and sandboxed.

That also meant that the code would be sloppy, at least at first, because the damage was limited to an errant web page. The applications ran outside the security perimeter since the servers could trust any code on the client side (AKA, in the browser).

Google has a huge financial interest in browser performance since a few tenths of a second in browser response time could translate into billions of dollars in revenue. Thus, the great incentive to improve the performance of JavaScript and it worked far better than I expected. I had been using JavaScript for simple projects[ii] when my friend Dan Bricklin commented that JavaScript had become a high-performance language with Google's V8 engine playing a key role.

One of the techniques used is JIT or Just-in-Time compilation. As computer performance increased, the percentage of time spent translating machine code relative to the total execution time become very small thus eliminating the need for precompilation. This meant that the JIT compiler could take advantage of the on-the-ground knowledge such as exactly which processor was being used and what special feature it may have.

Part of this was using the knowledge of the types of the parameters. Thus, if a piece of code is being called with an integer parameter that method could be compiled for just that case. The compiled code would do a quick check for the parameters to make sure that, at runtime, indeed, the parameters were integers. For a simple operation like addition that might be high overhead but such checking gets pushed back into the containing method and a whole set of operations can be made safe and efficient because the type checking could cover a variety of methods.

All this while preserving the ability to edge cases on the fly rather than having to check for them every time thus allowing more streamlining that precompiled code.

Another technique is memoization that can short circuit the entire computation by recognizing that the parameters are the same as a previous invocation and that the code has no side-effects.

These techniques work well when the code is regular but that is a very reasonable assumption akin to the locality assumption that allows both modern operating systems and machine learning to work so well. Oddball cases can cause performance problems but aren't fatal.

Declaring types in programming languages isn't merely about performance, it is also about safety. Having strong typing means that bugs are discovered at compilation time rather than lurking until the program runs. It turns out that using types for safety and types for runtime performance would be decoupled. I see Microsoft's **TypeScript**[iii] as a way to have a conversation with the IDE – Integrated Development Environment rather than as a necessity for compilation.

The IDE can help me identify problems and, just as important, refactor code, by following the implications of my declarations. If I try to add 12 to a string "12" raw JavaScript would simply convert the number to a string and give me the result "1212" which may or may not be what I want. If I use TypeScript the IDE would flag it thus forcing me to be explicit about what I want. Or not – I could just tell it to ignore the type information by using the "any" type. I get a chance to decide what is appropriate for the code I'm writing.

TypeScript is an important part of scaling code. I rely on its assistant to manage my own code since I need a housekeeper. It is more important for large teams of programmers. This is why Google has become a key supporter of TypeScript and uses it for projects such as AngularJS.

Visual Studio Code (VSC) is part of this ecology – an IDE well suited for TypeScript and written in TypeScript! It is built on Electron which is, in turn, built on the Google's Chromium engine that is behind both Chrome and (now) Microsoft's Edge browsers.

Much of this including the TypeScript compiler and VSC are open source with any contributors with GitHub (now owned by Microsoft) being the shared repository.

Note that both VSC and GitHub support a wide variety of languages, but I see JavaScript (any mention of TypeScript includes TypeScript) as having a special place while coexisting with other languages.

JavaScript doesn't have to be everything for everyone. It doesn't have to support complex asynchronous code and instead can focus on a single threaded model that eliminates much of the complexity of interlocking code while still supporting asynchronous code.

An operation such as fetching data from a server is done asynchronously. Traditionally this has been handled with callbacks that are invoked when the operation is complete. That pattern can become complicated very quickly. The async/await keywords greatly simply the code. For those used to more traditional languages you need to wrap your head around the idea that the code is really a callback behind the scenes but you can read it as linear code – you just need to choose whether you want to read the A path or the B path.

In a sense it's like my view of removing the GOTO from code. It allows you to use your understanding of the static code as a way to understand the runtime or dynamic behavior.

JavaScript does concepts such as worker threads that allows for spinning of truly parallel operations but with strong constraints on the interactions with the main code thus preserving the simplicity.

The lack of static typing means that classes in JavaScript are very different from classes in languages such as C++ or Java. Though the class keyword has been introduced into the languages the JavaScript classes are prototypes that can be copied and shared and manipulated at runtime. Coupled with the creative use of lambda expressions one can implement various styles of programming such as applicative programming. Though it is important to recognize how the JavaScript prototypes differ from static classes.

Together these innovations have given high performance production language that allows the developer to choose the style of programming and flexibility that is appropriate for a project.

JavaScript is still a work in progress. Future developments are needed to improve the safety of encapsulating code and increase the ability to operate on the code itself. There is a danger of adding too much and maybe we'll need a reboot but, for now, it's my language of choice.

Oh, there is an optional escape hatch – WebAssembly. It is the equivalent of low-level machine code but that's a topic in its own right.

Environments

It is useful to distinguish between JavaScript engines[iv] and platforms. Engine is a term of art that is used because neither interpreter nor JIT (Just in Time) compilation captures the essence of how JavaScript is implemented.

We also have platforms or environments which wrap JavaScript and support applications.

NodeJS

The language itself is only part of the story. Without the need for precompilation the code itself is portable. Hence the ability to import code into the browser from disparate sources on the fly and composite them. We're no longer confined to the browser.

While Ryan Dahl's[v] NodeJS[vi] was not the first server-side JavaScript environment it has become the standard. It takes full advantage of V8 (though it can use other engines) to provide a high-performance server environment with all the benefits of JavaScript. It also played a role in disentangling JavaScript from the browser DOM (Document Object Model).

Deprecated functions such as **`bold`**() (which returns a string surrounded by "`<bold></bold>`") are part of that legacy. Similarly, the implicit browser namespace has evolved to explicit namespaces for various environments.

Part of what makes NodeJS so powerful is the full support of the asynchronous capabilities of JavaScript. It also shared the powerful debugging tools with the browser making them available via VSC.

For very high-performance servers it's easy to spin up additional instances of the server to run in parallel. As I wrote[vii], my own site is written in JavaScript (OK, TypeScript) and runs very well. I do spin up additional instances as beta sites because it takes only a few minutes to startup a Digital Ocean droplet for my purpose.

For more complicated sites there is a lot of tooling but that is another topic.

NPM is the Node Package Manager that makes it easy to tap into a rich library of open source (and proprietary) packages typically stored on GitHub. Modules can be very simple or be very sophisticated as in React for implementing sophisticated websites.

Deno

Ryan Dahl looked at what he accomplished with Node and, as with any great developer, looked over his work and saw what a mess it was. It is a complicated piece of C++ code that requires Python to put it all together. The module capability had gotten bloated.

His new project, Deno, is very promising. It supports TypeScript natively and replaces the modules with direct use of libraries. While it's hard to displace Node, I'm excited by Deno and see it as complementing Node as a platform of choice, but it may take a while.

Deno is built using Rust[viii] which was developed by Mozilla as an alternative to C++. Rust has also been embraced by Microsoft.

As an aside, VSC was written by one of the developers of the Eclipse IDE. We're getting the benefits of a lot of experience contributing to the JavaScript Ecosystem.

ECMA TC53

Peter Hoddie[ix], best known for writing Apple's QuickTime Graphics library, initiated ECMA TC53 – JavaScript modules for embedded systems. It reminds me of the early days of microprocessors when the chip people saw processors simply as a way to save a few chips in a circuit. Us software types saw it very differently – very inexpensive computers we can own. That was the genesis of Steve Wozniak's original Apple computer which used the low-priced 6502 processor. I could then take that "toy" computer and turn it into a powerful tool for financial analysis.

ECMA TC53 allows us to rethink how we build devices and hardware in general. We can now focus on the capabilities and well-written software rather than the accidental properties of particular hardware and worrying about the pitfalls of C++ programming. Moddable's JavaScript engine is called XS.

I've described how high-performance processors have allowed us to embrace the programming techniques to make the V8 engine so powerful. Conversely, we can now take our understanding to very small systems and gain the benefit of JavaScript even if the performance isn't as high. The intellectual synergy is what is important.

JavaScript Inside

There are a number of other engines and platforms including Johnny-Five, JerryScript, Espruino, WebOS etc. which implemented JavaScript (or, sometimes, subsets).

Each one is worthy of a discussion in its own right. What is important is that these engines have very small footprints. One point that the Espruino people make is that sending source (or tokenized) JavaScript can be more compact than using compiled code.

Users can add widgets and capabilities to the platforms without threatening the integrity unless the platform providers choose to expose features that might be risky. This is similar to running apps on a Smartphone where you need to approve access to capabilities.

Browser Environments

The browser is more than just a JavaScript engine. It's a virtual and portable operating system. It has become a first-class application environment with great support for mobile (AKA, small portable) devices.

Progress Web Apps (PWAs) have a number of advantages over so-called native applications including taking advantage of the full tooling of the JavaScript ecosystem. PWAs don't need to be "installed". They are web pages (more properly, applications) that can be available offline. They are indistinguishable from native applications but as convenient as visiting a web site.

The term "Progressive" is used because applications can progressively take advantage of features available at runtime – another reminder of taking advantage of the on-the-ground conditions.

It is up to the underlying platform to choose which native capabilities are made available and the user can decide how much to trust the app.

The New Set Top Box

One area where I expect browser engines to become dominant is as a TV viewing platform as the new Set Top Box (not that there is room on an LCD). The growing variety of boxes -- Roku, Amazon Fire, Xfinity Flex, LG's WebOS, 小米 etc. – recalls the days of a variety of incompatible PCs that limited the ability to write portable software and to reach a wide market. The browser already supports TV viewing and I find the experience better than the native ones in those boxes.

There are some user-experience (UX) elements that are missing. One is relatively simple – standardizing the use of the keys on a handheld controller since the keyboard is often too clunky to hold while lounging. Another possibility is to have an API that makes it easy to write control applications so that users can create their own controls using smartphones. Instead of picture-in-picture, perhaps, previews can be on the local device for one person while the family can continue to watch the large screen. Users can then implement their own voice navigation and more.

Another reason for using the browser as the environment is that the nature of TV is changing. While a tablet might be used to control a large TV, it is also a first-class viewing device in its own right. We can have multiple viewing services and controllers that can work individually or in concert (pun partially intended). We need tools that allow us to create apps that work across and within each of the devices and environments.

Designing such an environment would be a good project for the CT Society as a vendor-neutral organization with connections to the TV broadcast industry. We already see

elements of such an environment in Chromebook and LG's WebOS.

Beyond the REST

JSON based APIs have been associated with RESTful interfaces but those are a subset. The use of HTTP/JSON is a very common pattern and makes it easy for JavaScript programs to provide and use services. Of course, the interface isn't restricted to JavaScript, but I considered it part of the ecosystem because of the use of JSON as the data format.

The stark simplicity of JSON is in sharp contrast with earlier protocols that tended to use binary data structure so were inscrutable. We also had XML interfaces on the path to JSON but protocols like SOAP tended to be complicated having inherited paradigms from the static typed languages. In addition to its simplicity JSON payloads are easy to extend. It doesn't matter if a number is passed in a string field as long as it can be converted to a number.

Using JSON means that it's easy to evolve the interface and add new bundles of information that can be passed on without interpretation. This nicely matches the design point of the internet – intermediate applications can transfer data without having to understand all of it. The information is not fully trusted so that it's safe. In fact, this lack of trust leads to resilience because it means verifying the contents where it is actually used in context.

Systems Thinking

Thinking outside the Box

Literally.

We can think of meta-devices that span multiple pieces of hardware which is the inverse of multiple applications sharing single device.

A weather reporting system can have monitoring software running on multiple computers, each contributing to the whole even though no particular system is critical as long as we can create an overall map of the weather. JavaScript facilitates this making code portable without artifacts of compilation. Even better using Deno so that the libraries used are runtime resources.

This is a way to think about an Internet of Things – an IoT – not as a carefully engineered systems with well-defined components but capabilities that can be composited dynamically using best-efforts connectivity. The devices can be general purpose platforms such as a Raspberry Pi with weather monitoring being one application on either a dedicated or shared device. Rather than carefully engineering each connection we can take a resilient approach. In the weather monitoring examples, if a few systems go down that's not a problem since the information comes from the collection and is not critically dependent upon any single node. If there is a connectivity failure the data can be stored and made available later. There is no need to distinguish between a failure of a link and a failure of a device. It's about dynamic relationships among communities of things[x]. This approach of finding value in the whole has many advantages in allowing innovation without relying on a provider's guarantee of reliable delivery. Instead we can take advantage of any opportunity to connect.

Cloud-hosted Resources

The cloud grew up as a way to host massive computing resources. In an early generation we used the term timesharing. We're now evolving a more decentralized approach that includes local computing and using these shared facilities (AKA the cloud) as a resource. Many applications that were cloud-based such as machine learning now run locally. At least in part.

The JSON APIs are part of the ecosystem and so is portable code. You can think of an AWS lambda as a snippet of code that can be invoked independent of the particular hardware and can run locally or remotely. Since JavaScript can be transferred as easily across systems, we can make creative use of resources that match our needs by being able to take advantage of the computing power, locality, storage, and other characteristics dynamically. This is a reminder that it is about the application and not any particular computer hardware.

Beyond JavaScript

While it is convenient to see everything in terms of JavaScript, we aren't limited to one language. It coexists with the Python community that has some similar characteristics and rich libraries for the data analysis and other communities. There are also more traditional languages like C# and Rust for situations where we need more intense parallelism and low-level access.

The power of the JavaScript ecosystem is making it easy to write applications that add capabilities incrementally. It facilitates portability both because the code itself is portable and because using simple JSON APIs facilitates cooperation across systems. The V8 engine is one example of the power of a large ecosystem that can invest in improving performance beyond what is supposed to be possible.

The same ecosystem also gives us small engines that extend the reach.

My website is, of course, one example of using JavaScript and how it facilitates development. I recently added a small number of lines of JavaScript to add video streaming capabilities. It took that little because of the synergy with the larger ecosystems. That synergy is what makes this ecosystem so exciting.

The Future of Consumer Technology

I see JavaScript as part of the larger trend of using software to define products and empowering people to be contributors and not just consumers though that word "consumer" won't go away any more than "dial" does for phone calls and "film" goes for recording MP4 video.

The IEEE Consumer Electronics Society was originally part of the Broadcasting society in the days when you'd have a new industry for each technology. Today a television is just one more app. JavaScript is at the nexus of this transformation.

---

[i] https://rmf.vc/IEEEHtml5
[ii] https://rmf.vc/IEEEBlinky
[iii] https://en.wikipedia.org/wiki/TypeScript
[iv] https://en.wikipedia.org/wiki/JavaScript_engine
[v] https://en.wikipedia.org/wiki/Ryan_Dahl
[vi] https://en.wikipedia.org/wiki/Node.js
[vii] http://rmf.vc/IEEEInsites
[viii] https://www.rust-lang.org/
[ix] https://www.moddable.com/peter-hoddie
[x] https://rmf.vc/ieeecommunitiesofthings