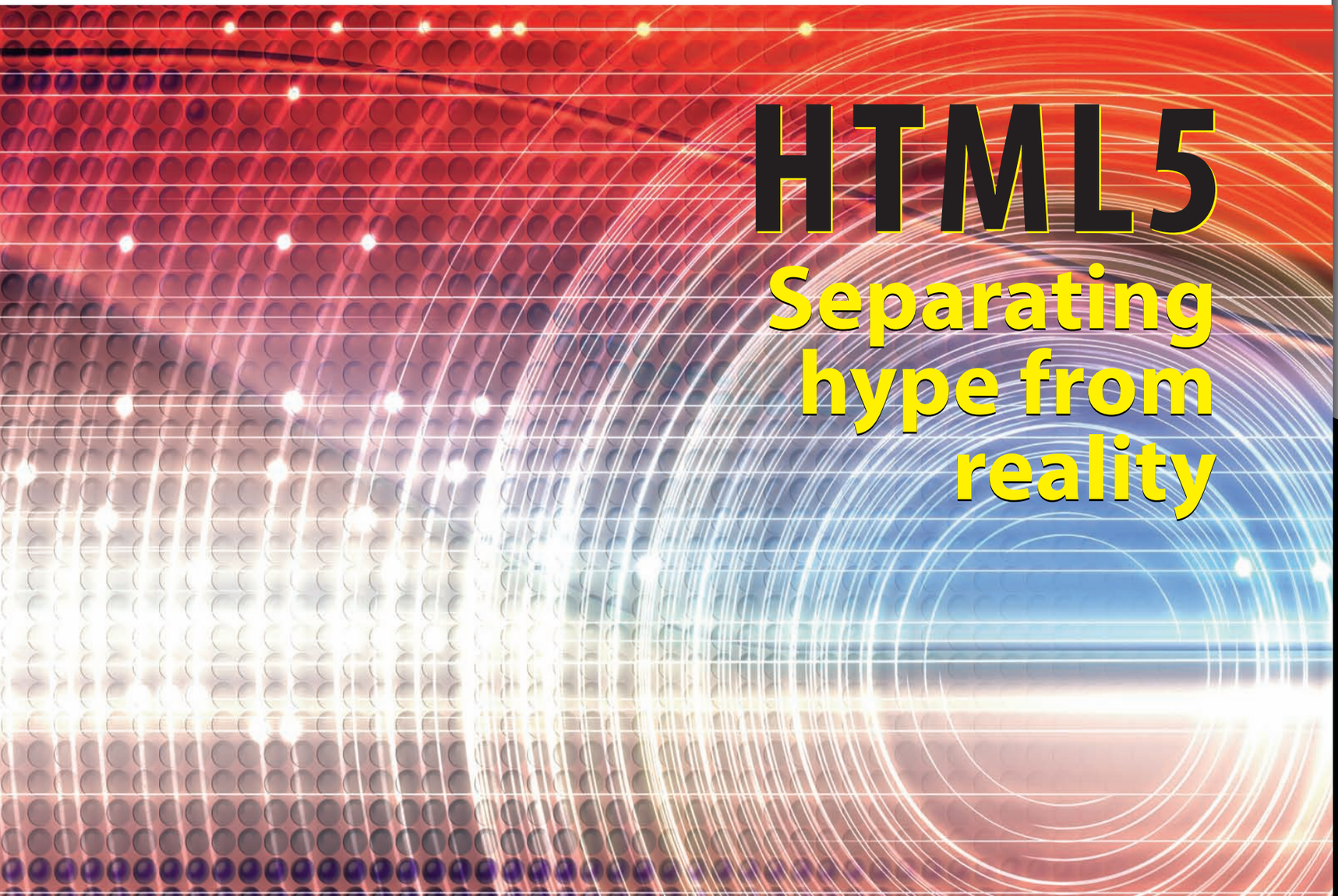
WEB DEVELOPMENT ISSUE

Dr. Dobb's Journal

NOVEMBER 2011



HTML5

Separating hype from reality

Next

ALSO INSIDE

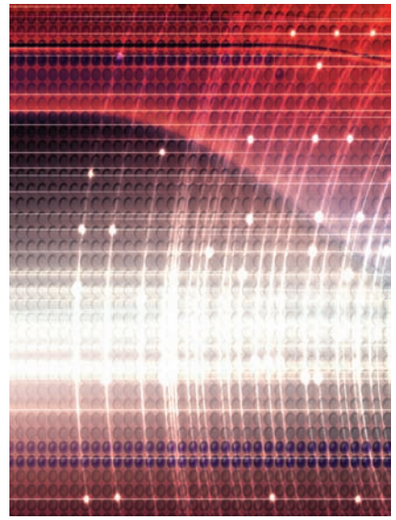
UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

November 2011



COVER STORY

More on DrDobbs.com

Dennis Ritchie, in Memoriam

The inventor of C, designer of a universal language syntax, and a major contributor to UNIX died recently at the age of 70.

http://drdobbs.com/cpp/231900742

Java Reloaded

After several embarrassing bumbles, Oracle is finally getting Java on track.

http://drdobbs.com/java/231900536

Sharding, Replication, Caches, and In-Memory Databases

Caching, replication, and sharding have proven to be important tools in the modern database architect's toolbox. To meet the requirements of high-throughput websites, system architects are turning to distributed caches and in-memory data grids to reduce I/O operations against disk-based databases.

http://drdobbs.com/blogs/high-performance-computing/231000846

Currying and Partial Functions in JavaScript

One of the most handy and unusual techniques in JavaScript is applying functions, both in whole and in part, as well as transforming them on the fly

http://drdobbs.com/open-source/231001821

Elegance or Trickery: How To Free Memory

One way to free memory is to create a temporary, empty `vector`, and swap that `vector`'s contents with `v`'s elements.

http://drdobbs.com/blogs/cpp/231900605

Mailbag

Portability, Gary Killdall, and Metrics

What Portable Apps?

In a recent editorial (http://drdobbs.com/architecture-and-design/231901048), editor Andrew Binstock argues that many apps today, especially those written in native languages, tend to have poor portability; and if the trend should continue that the so-called "C++ renaissance" might be a short-lived affair. His comments received several passionate rejoinders:

How inspiring to find that there are still those willing to step forward and once again point out that the Emperor is wearing no clothes!

When the hype cycle fades, as it inevitably will, perhaps the wiser among us will finally come to realize that despite all of the lip-service major vendors are paying to "write once, deploy everywhere" and HTML5/JavaScript, none of the platform providers have any real interest in such efforts [succeeding]. As Microsoft indicated at MIX11, they still believe that native applications are best — the very reason they're simultaneously paying homage to HTML5 while ensuring IE provides capabilities unavailable via other browsers. Google's doing the same with Chrome. Apple's built a wall around its empire for exactly the same reason, specifically banning Flash not for the irrational reasons given, but because Flash, and to a lesser degree, Silverlight, are still unmatched with regard to the creation of immersive user experiences — UNLESS one goes native.

— posted by user "Chance"

The reality remains that each vendor has a justifiably vested interest in developers fully exploiting that which is native to their platform — their future success is extremely reliant on maintaining such levels of differentiation, for no one wins upon a level playing field.

Fortran is the most portable language out there. It's the most portable because it doesn't interact with the hardware. If you have Linux, you have Fortran. 30-year-old dusty decks can usually be recompiled to run on anything (via GCC, and that's a lot of stuff). Fortran is a fully modern language with an active standards committee. It became unpopular because the heavyweights (Microsoft) couldn't manipulate it for profit.

— Rob Hickey

Gary Kildall

Our 1997 article on Gary Kildall's life (http://drdobbs.com/architecture-and-design/184410428) elicited the following comment:

Gary owned the Pebble Beach house before the Novell merger and he was already wealthy and had his stunt planes, four Lamborghinis and other assorted collectibles, and the CD project was well underway. He had sold out of Digital Research before that company ran into financial troubles (due to the huge integrated languages project at a time when income was dropping). Gary bought back into DRI to try and salvage it, he brought in Dick Williams from IBM to CEO the company and started getting interested in GEM again. He had some ideas he wanted to pursue with it, which was before Microsoft built Windows, which was a Mac/GEM imitation that Microsoft gave away for free. It's hard to believe that clunky thing ever became stable enough to own the market. I was the GEM programmer support engineer at the time and Gary and I were friends who loved exotics, I had a De Tomaso Pantera, he later bought a De Tomaso Mangusta just before his death. It has to be said that no one saw the small computer revolution coming in the abundance that it did. It was the Pacific Rim manufacturers that realized they could build an IBM-compatible computer, run all of that great software IBM built or sponsored for half price, which made it available to the common person's budget, and all they had to do was run that operating system.

—posted by user "BTOMINGAS000"

The Dismal Science of Code Metrics

In response to a discussion of how little use many code metrics are today, we received this observation from Capers Jones.

The lines of code (LOC) metric has such serious economic flaws that I classify use of that metric as professional malpractice. Your column was interesting but did not discuss the main failings of LOC metrics:

- Requirements and design bugs outnumber code bugs and are invisible using LOC
- LOC metrics penalize high-level languages in direct proportion to the power of the language (your readers should know this)
- There are no international LOC counting standards
- For more than 50 languages out of 2,500 there are no effective counting rules at all because of the use of pull-down menus and buttons à la Visual Basic.

We recently did a study of 10 languages used for the same application. The productivity and quality were measured using both LOC metrics and function points. The paper was derived from a consulting study where developers were stopped from using object-oriented languages because managers thought the procedural language in use had higher LOC per month. The managers' analysis, we demonstrated, was seriously in error.

— Capers Jones

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

The New Intermediate Language of Choice

As more languages target the browser, they're using JavaScript as their output format

Earlier this month, Google unveiled Dart, a new language it developed for internal use that provides a variety of useful features and generates, not C, not native binaries, but JavaScript. The initial reception has been mixed: some developers liking the idea, others wondering about the need for another Google-sponsored language, and a third group complaining about yet another programming language, regardless of who sponsored it. I don't share the viewpoint of the third group at all. One of the most fertile areas of innovation in computing is in language design. And today, we're seeing a bountiful harvest of new languages bringing all kinds of interesting features, many of which are being picked up by established mainstream languages. It's precisely because of the richness of this innovation that we at *Dr. Dobb's* run a regular "Language of the Month" feature. Our biggest challenge with the column is getting to all the interesting languages fast enough.

A theme that is emerging in new languages such as Dart is the use of JavaScript as a universal intermediate language. In addition to Dart, Fantom (our "Language of the Month" in February this year http://drdobbs.com/tools/229218754) and CoffeeScript do this. Moreover, some Web frameworks, such as Google Web Toolkit (GWT) do something similar, namely translating other languages (in the case of GWT, Java) into JavaScript.

The choice of JavaScript in many ways parallels the choice of C for this role in years past. The prime reason for its adoption is portability. Every browser supports JavaScript and supports it well. There is active competition today between the major execution engine (Google's V8, Mozilla's xMonkey, and the recently announced Chakra from Microsoft), so performance is another reason for adoption. Finally, like C, it's comparatively easy to drop down to the low level of the emitted language and program some special functionality, if needed.

A key part of the first two benefits derives from the fact that JavaScript is the province of no single vendor. In this regard, push-

ing JavaScript through the standardization process years ago as ECMAscript was an event of greater importance than it appeared at the time. In so doing, JavaScript was moved out of the shadow of Netscape and Mozilla and became a universal language, not quite native, and not quite high-level enough to impose heavy start-up costs. In other words, it's a language suited to being an intermediate language.

What JavaScript does not share with C, though, is the latter language's elegance. As a leaked memo from Google states (https://gist.github.com/1208618), part of the motivation for Dart (then called "Dash") is to get around the fact that "Javascript has fundamental flaws that cannot be fixed merely by evolving the language." Google believes that there is little likelihood that going the standardization route will be an effective remedy for these issues, as changes to the standard "will take years and will be limited by fundamental problems in the language (like the existence of a single Number primitive)."

The memo, being internally directed, does not give reasons why a user should prefer Dart over CoffeeScript or Fantom. And, at this point, it's far too early to compare the languages and provide any recommendation. All three languages are new: Fantom began generation of JavaScript within the last year, CoffeeScript was started in 2009 and reached v. 1.0 in 2010, and Dart is very much in beta. We will, of course, track the languages at *Dr. Dobb's* as they mature, but I'm betting that before any of them reaches a tipping point, they'll be joined by new languages that also use JavaScript as their output.

— Andrew Binstock is Editor in Chief for **Dr. Dobb's** *and can be contacted at alb@drdobbs.com.*

Comment

HTML5 — The Hype And the Reality Behind It

As HTML5 emerges in browsers and on mobile devices, it's important to know what it can and cannot do — especially since the draft standard itself is still evolving

By Dino Esposito

HTML has been around for about twenty years now and has gone through two main ages — static rendering and dynamic rendering. Static rendering was the infancy of the Web when HTML was used to display the content of documents and links between them. Dynamic rendering came a few years later when browser vendors made the representation of the displayed document modifiable via JavaScript. That was the youth of the Web bringing a new vision of the markup. HTML was then perceived as a means to provide dynamic representation of the content of a Web page; more of an application delivery format than a plain document format.

HTML5 marks the beginning of the third age of the Web in which HTML advances at a brisk pace towards becoming a true application delivery format. HTML5 is not limited to presentation, but also provides a slew of new functionalities for Web and other types of applications.

These days, HTML5 is being discussed a lot and pushed hard as the next big thing for whatever kind of software applications you're involved with. In this article, I'll try to provide an overview of HTML5 and focus on dispelling the hype that surrounds HTML5.

The Hype

First and foremost, HTML5 is not here yet and the World Wide Web Consortium (W3C) is not expected to release any recommendation (http://www.w3.org/TR/html5/) for at least three more years. Until recently, mindful of the browsers war and how hard you worked to

provide users with the same experience regardless of the browser, you would likely have dismissed the argument about an announced but yet-to-come version of the HTML language.

So why are we talking HTML5 today, a good three years ahead of its official release? We talk about HTML5 today because most browsers are already incorporating HTML5 capabilities in their latest versions.

For certain, HTML5 will play the same prominent place in Web development HTML has played so far. In addition, the new features in HTML5 make it possible to build other types of applications. For example, mobile native applications for platforms like the iPhone and Android. I don't know — and anybody really knows — whether HTML5 and JavaScript will form the primary development platform for the years to come, but regardless of that role HTML5 is worth a closer technical look.

HTML5 is not simply markup. It comes tightly bound to CSS3 and ECMAScript5 (the latest JavaScript). In the end, the emphasis on HTML5 is due to the programming power that results from the combined effect of markup, style and script enhancements, plus a bunch of new APIs that browsers implement in a unified manner.

The Good

HTML5 is a rich markup language that incorporates in its standard syntax many common features that developers and designers previously coded by hand in thousands of Websites. Let's delve deeper in some of these.

Semantic Markup

It's common to have a header and footer in pages as well as a navigation bar on the left of the page. More often than not, these features are achieved by using DIV elements styled to align to the left or the right. Most pages today end up with the following template:

```
<div id="page">
   <div id="header">
   </div>
   <div id="navbar">
   </div>

   <div id="container">
      <div id="left-sidebar">
      </div>
      <div id="content">
      </div>
      <div id="right-sidebar">
      </div>
   </div>
   <div id="footer">
   </div>
</div>
```

The template includes header, navigation bar, footer and a three column layout in between. The markup alone, however, doesn't produce the expected result. For that, you need to add ad hoc CSS styles to individual DIV elements.

With HTML5, you still need to use the same bit of CSS to make the page look compelling and place segments where they belong. However, you now can describe the page in a much easier way, replacing generic DIV elements with more semantically meaningful elements such as HEADER, FOOTER, and ARTICLE. Here's how you can rewrite the preceding template with newest HTML tags.

```
<header> ... </header>
<nav> ... </nav>

<article>
   <aside>
      ...
   </aside>
   <section> ... </section>
   <section> ... </section>
   <section> ... </section>
   <aside>
      ...
   </aside>
</article>

<footer> ... </footer>
```

The NAV element logically groups links that would go in a navigation bar. The ARTICLE element represents the container of any content for the page and incorporates ASIDE elements and SECTION elements. All these are block elements and must be styled properly to form a presentable page. Other new elements complete the list of enhancements such as FIGURE and DETAILS. The FIGURE element is designed to include figures with caption whereas DETAILS replaces the canonical hidden DIV that developers use to hide optional content and display it via JavaScript. The DETAILS element just offers a clear semantic and an appropriate collection of attributes.

Input Forms

The current HTML allows users to enter only plain text, leaving developers the pleasure of validating actual content against numbers, emails, dates or phone numbers. HTML5 comes with several new values for the attribute type of the INPUT element. Here are a few examples:

```
<input type="date" />
<input type="time" />
<input type="range" />
<input type="number" required />
<input type="email" />
```

What's the real effect of these values? The intended effect — though not completely standardized yet — is that browsers provide an ad hoc user interface so that users can comfortably enter a date, time or number. As of today, Opera 11 is the only browser that takes new values for the type attribute seriously. Opera 11 automatically pops up a calendar for dates and a slider for a range type. As Figure 1 shows, Opera provides also automatic validation for fields. The input field in figure is marked as an email field but it doesn't contain a valid email address and Opera complains! Finally, the `required` attribute marks an input field as mandatory whereas the `placeholder` attribute allows displaying a hint in a text box wherever

possible. (You won't find a hint in a range input field that is rendered using a slider.)

Note that the level of support you get from current browsers for the input types is limited. In the end, developers have still a bit of work to do with JavaScript to ensure that correct data is posted to the server. The primary benefit is that the underlying markup is more readable.

Audio and Video

One of the biggest gains of HTML5 is saying farewell to external plugins such as Flash and Silverlight for playing audio and video. You have two new elements that point to a URL and play any content. The browser implementation of these tags is expected to provide a control bar for the user to pause and resume the playback.

The sore point of multimedia elements is the format (both file format and codecs) of audio and video files. The HTML5 won't make an official call, so deciding about the format to support will remain up to the vendors. From a developer's perspective this is not exactly great news as it represents a breaking point — different browsers support different formats and you should detect the browser or provide multiple files for the browser to choose. Here's the syntax to indicate a selection of formats.

```
<video poster="init.png" controls>
   <source src="..." type="video/mp4" codecs="..." />
   <source src="..." type="video/webm" codecs="..." />
</video>
```

Note that you use the `controls` attribute to display the control bar and the `poster` attribute to specify an image to use as a splash-screen until the media is ready to play.

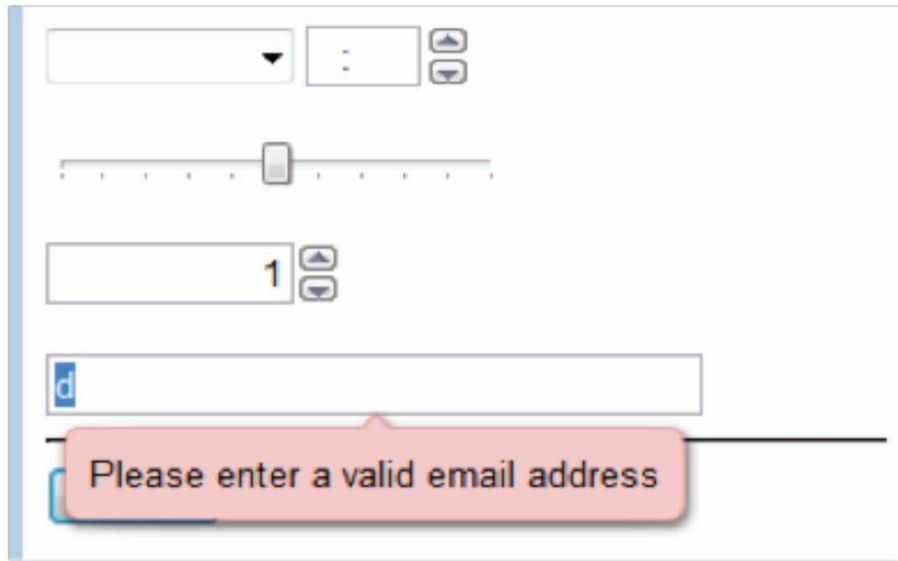


Figure 1. HTML5 input in Opera 11.

Data Storage and Offline Working

We all know that Websites need an active Internet connection to work. Whether Ajax or browser-led, any requests the user generates by interacting with the site are to be delivered to some remote endpoint to obtain a response — typically, a Web page or auxiliary resources like images, scripts and styles. All browsers have some caching mechanism specifically introduced to save a few HTTP requests by resolving static resources from the local computer. However, this mechanism has never been standardized and any browsers use it more as a form of optimization than as a way to enable users to work on the site while offline.

HTML5 comes with several separate specifications for Data Storage and Offline Applications. Altogether, these APIs provide a framework

for developers to build Web applications that not only force the browser to cache resources according to a public manifest, but also enable the local code to save application data in a custom format according to application-specific rules. One could say that data storage is a very enhanced version of cookies, whereas offline working is a very enhanced version of the classic browser cache.

You access the local storage through the `localStorage` property exposed by the browser's window object. This property offers a dictionary-based programming interface similar to that of cookies. There are methods to add and remove items, to count the number of items in the store, to get the value of a particular item and to empty the store. You use the manifest cache special resource, instead, to let the browser know about which resources to cache and which to always request to the network. You reference a manifest this way:

```
<html manifest="manifest.cache">
```

Note that the URL can be a dynamic page (for example, ASPX, PHP) and the resource must be typed as text/cache-manifest. The content of a manifest file is a plain list of resource names with a bit of syntax rules:

```
CACHE:
default.html
styles.css
logo.png

NETWORK:
login.aspx
/public/api

FALLBACK:
login.aspx nologin.png
```

The CACHE section indicates resources to always cache; the NETWORK section lists resources to always request from the server; the FALLBACK section indicates alternate resources to use when network is not available.

The Weird

So is HTML5 ready for prime time? HTML5 is definitely receiving a very warm welcome and its features — although not standardized yet — are gradually finding support in recent browsers. Some flavors of HTML5 are available today in Opera, Chrome, Firefox, and Internet Explorer 9. They are, however, flavors. Browser incompatibilities do exist and will likely continue to exist at least until the specification is finalized around 2014. What should you do today?

As far as Web applications are concerned, either you exercise control over the browser or you provide a powerful fallback mechanism. Input elements can be coded as HTML5 and they will render gracefully on older browsers; semantic elements such as ARTICLE should be styled appropriately to render gracefully on older browsers.

You also need to provide fallback for media elements. In the end, it's not a mission-impossible task, but it is an extra amount of work that, as I see things, doesn't pay off currently. What's the point of serving HTML5 markup instead of plain old markup?

It's a totally different story if you intend to leverage companion features of HTML5 such as offline or data storage. But, again, what if the browser doesn't support HTML5 or, supports it partially? You have to code two versions of the site. A library like Modernizr (http://www.modernizr.com) helps in that it performs feature detection.

Summary

The bottom line is that I see HTML5 ramping up especially outside the classic realm of Web applications. I expect the full set of HTML5 capabilities to form a compelling framework for mobile and tablet applications and even for developing applications for delivery over a variety of software platforms. A tool like Adobe's PhoneGap allows writing native applications for iOS and Android by using HTML5, CSS3 and recent JavaScript enhancements. With it, you can leverage the full power of HTML5 without being concerned about the browser's capabilities. Surprisingly enough, all browsers on recent smartphone OS are HTML5 ready. Similar full support for HTML5 development is provided in Adobe AIR and was announced by Microsoft for Windows 8. HTML5 is ready today for good development but I see it as being beneficial mostly in closed environments.

— Dino Esposito is the author of several programming books, including the recent **Programming Microsoft ASP.NET 4** *(http://is.gd/HlXFdm) from Microsoft Press.*

Comment

Language of the Month: Opa

A single language to write all three tiers of a Web application and provide intelligent scalable deployment

By Adam Koprowski

Opa is a new Web programming language. It delivers innovation to Web programming in three main areas:

- Web programming is typically very fragmented. The database, client code, and server code are all driven by different technologies, which introduces complexity and the need for communication among different elements of the application. Opa eliminates these difficulties by providing a uniform and coherent platform for every part of the application.
- Opa provides a common runtime for all the components of the application. The Web server, the database, your application, and its resources all live together in a single executable. Deployment is as simple as executing the file. Cloud deployment is as simple as executing it on many machines at once.
- Opa brings increased safety to Web development thanks to its use of static analysis and static typing techniques.

The ideas behind Opa go back 10 years or more, but development gained real speed four years ago with the founding of MLstate in Paris. Opa was its principal product.

Work on the new language was triggered by frustration with existing tools for Web development. Even relatively simple apps were composed of a bunch of separate components that had to be harnessed to work together, sometimes against their will.

Want to develop a desktop application? Choose your language and you are on your way. Want to write a Web app? You'll have to configure a Web server and a DBMS, master a database query language, select a client-side language, and choose a server-side language, plus probably a Web framework on top of all that. Then you'll have to set up lots of boring boiler-plate for client-server communication with JSON or XML.

Developers seem to accept this multitude of technologies as inherent to Web programming. But it doesn't have to be. What if you could create all the components of your application with one coherent package?

That's what Opa is all about. Opa stands for "One Pot Application." It flattens the technology stack needed to develop for the Web.

Hello Opa

This is probably the simplest application that you can write in Opa:

```
server = Server.one_page_server("Hello", ( -> <>Hello Web</>))
```

This program is the Web equivalent of the traditional "Hello world" program. It creates a Web service with a single, static Web-page titled "Hello" and with "Hello Web" as its content. Let's take a look at the code in more detail.

`server` is a keyword that marks the entry point of the application — an HTTP server, as Opa programs are Web applications. It's like `main` in Java or C, only an Opa program may consist of more than one `server` — if it is listening on different ports, for instance.

`Server.one_page_server` is a function that constructs a particular, simple type of server, which consists of a single page. It takes two arguments: the title of the page and its body. The first argument is a single string. The second is more interesting. It's a function (with no arguments) that returns an `xhtml`, which is Opa's type for, well, XHTML. Indeed, if we take a look at Opa's API documentation we see:

```
Server.one_page_server : string, ( -> xhtml) -> service
```

In our call to this function, we pass `-> <>Hello Web</>` for the argument. There are two things to notice here.

First, the arrow is used to construct anonymous functions, the general syntax being `arg1, arg2, ... -> exp`. In our example, there are no arguments; hence, nothing before the arrow.

The second is the syntax for XHTML. The empty tags `<>...</>` serve as delimiters for the markup. (We could have written something like: `<span>Hello Web</>`; the name in the closing tag is optional.)

I mentioned that the goal of Opa was to replace a complex technology stack with a single language — so what is XHTML doing here? It turns out that Opa does not replace (X)HTML or CSS. Instead, it incorporates them into the language. They are regular datatypes that can be manipulated easily within Opa.

It would be possible to build an abstraction layer on top of those presentation technologies, and in a sense, Opa provides such a layer in the form of numerous widgets. But in the end, Web pages are built using (X)HTML and CSS. A Web language should make it possible to work with them easily.

Let's continue with our "Hello Web" example. After saving the code to a file, hello_web.opa in this case, we compile it with:

```
opa hello_web.opa
```

Compilation results in a single executable: hello_web.exe. This executable contains:

- All the resources used by the application, including images, external stylesheets, and so on
- JavaScript for client-side code (automatically generated by Opa)
- Code for all the client-server communication (Ajax & Comet, generated by the compiler)
- Database queries (automatically generated by Opa)
- An HTTP server
- A DBMS

In other words, the executable contains all that is needed to run the application. Copy it to the server, execute it, and your Web application is up and running. Want to deploy the application in the cloud? Use the opa-cloud tool.

Opa's Type System

Opa incorporates a strong static type system. It is based on the notion of a record (product types), and on sums (union types). Together, these types form the basis for algebraic types.

Almost every type in Opa is built using algebraic types, except for primitives such as `string`, `int`, and `float`. Even `css` and `xhtml` are constructed using algebraic types. For instance, Opa's boolean type is a choice between true or false:

```
type bool = {true : void} / {false : void}
```

The type declaration can be abbreviated as:

```
type bool = {true} / {false}
```

This is roughly equivalent to an enumeration type, but we can add more complex records and recursive types. Here is the concise definition of a binary tree:

```
type bin_tree('a) =
  / {leaf}
  / {left:bin_tree('a)  value:'a  right:bin_tree('a) }
```

The `'a` in this code is a type-parameter that can be instantiated with any other type, resulting in a binary tree over elements of that type.

Opa is a higher-order language, so the types of record labels can be functions. As a result, records can also be used as interfaces. Consider

this (simplified) signature for a function that creates a button. The arguments are its content and the function that should be executed when the button is clicked. The function returns a button; that is, a record allowing interaction with the button (in this case, changing its title and disabling it).

```
mk_button(title : xhtml, on_click : -> void) :
  { set_title : xhtml -> void; disable : -> void }
```

A common criticism of statically typed languages is the overhead required by writing types. Type inference ensures that this is almost never required in Opa.

Database Integration

I said earlier that Opa eliminates the need for a multitude of technologies. SQL is one of them. Declaring persistent data is as simple as declaring a variable:

```
db /revenue : intmap(float) // mapping year -> revenue
```

The only difference is that in such declarations, stating the type explicitly is required.

This is how we perform reads:

```
this_year_revenue = /revenue[2011]
```

Updates look like this:

```
/revenue[2011] <- /revenue[2011] + 999.99
```

In this example, `/revenue` is a path, which is used to organize data in a hierarchical way — much like files in a file system. The leaves in this database tree can be arbitrary Opa types (with the exception of functional types), giving rise to further levels in the hierarchy. Think of records having other records as their fields, and so on.

At this time, all of this works with Opa's internal database. A future release is expected to offer interoperability with MongoDB and CouchDB.

DOM Manipulation and MVC

Opa offers tight integration with XHTML for the presentation layer. This code is a simple but complete application that presents a button. Upon clicking the button, the present date is displayed or updated.

```
Page() =
  date_id = Dom.fresh_id()
  show_date(_) =
    Dom.transform([#{date_id}
       <- <>{Date.to_string(Date.now())}</>])
  <button onclick={show_date}>Click for date</>
  <span id=#{date_id} />
server = one_page_server("Date", page)
```

The last two lines of the `Page` function are its result: the XHTML representation of the page. The first tag is a `<button>`. Its `onclick` attribute contains Opa code to handle the event. In this case, it's a `show_date` function that uses the `Dom.transform` function to perform a list of transformations on the page DOM. Here, it simply replaces the content of the element with `date_id` with an XHTML representation of the current date.

This example conforms to the MVC architecture. Opa does not enforce MVC structure, but MVC is supported. It's perfectly possible to separate different layers of Opa programs, with the database serving as a model, the controller being the logic in the code, and the view embodied by XHTML, possibly factored out of the program using Opa's templating library.

Mutability and State

It is well known that mutation-free code is much easier to reason about. It is equally well known that every non-trivial program will have mutable-state. Opa tries to find a balance here by being mostly pure, with the exception of database operations and sessions.

Opa uses sessions to capture and handle mutable state. A new session is created like this:

```
Session.NonBlocking.make :
   'state, ('msg, Session.NonBlocking.handler('state) -> void) ->
    channel('msg)
```

This function takes two arguments: the initial state of the session and a callback function that is called every time a message is sent to the session. The callback function has two arguments: the message and a handler. The `make` function returns a channel that can be used to send messages to the session:

```
send : channel('message), 'message -> void
```

The handler in the callback function can be used for retrieving and updating the state:

```
Session.NonBlocking.get :
  Session.NonBlocking.handler('state)->  'state
Session.NonBlocking.update :
  ('state -> 'state), Session.NonBlocking.handler('state) -> void
```

`Session.NonBlocking.get` returns the current state associated with a session. `Session.NonBlocking.update` updates this state.

This may look simple, but make no mistake: Sessions are a powerful concept in Opa, especially in combination with location transparency, which ensures that session communication works regardless of the location of the sender and the session.

Opa further builds on the concept of sessions with cells (which are essentially sessions that return results to a received message) and with networks (which offer infrastructure for broadcasting information to observers). Together, these notions serve as a high-level backbone for distributed communication in the Opa ecosystem.

Client-Server Separation

The biggest source of complexity in Web development is the distribution of functionality between the client and the server. (And of course, it is common for many clients and multiple servers to be involved.) Opa transparently takes care of dividing the application between the client and the server, and it takes care of all the communication between the two.

The Opa compiler decides where to put code on a case-by-case basis, marking each function as client-only, server-only, or both. The functions for implementing the user interface (manipulating the DOM) are clear

candidates for the client side, of course, while functions that access the database will need to be kept on the server. The compiler also attempts to minimize communication overhead.

A call to a function residing on the "other" side is changed into a remote procedure call, with all the serialization, deserialization, and communication handled transparently. The fact that this is done automatically saves developers lots of trouble and makes client-server communication more secure.

This automatic process works well most of the time, but sometimes performance considerations require tweaking with the `@client`, `@server`, and `@both` directives. Similarly, safety precautions may require use of visibility directives such as `@private` and `@publish`.

Showcase

The canonical example of a Web application is chat. Let's see how a simple chat program can be coded in Opa.

```
type message = { author : string ; text : string }

@publish room = Network.cloud("room") : Network.network(message)

user_update(x : message) =
  line = <div class="line">
           <div class="user">{x.author}:</>
           <div class="message">{x.text}</>
         </>
  do Dom.transform([#conversation +<- line ])
  Dom.scroll_to_bottom(#conversation)

broadcast(author) =
  do Network.broadcast({~author text=Dom.get_value(#entry)}, room)
   Dom.clear_value(#entry)

start() =
   author = Random.string(8)
   <div id=#header><div id=#logo></></>
   <div id=#conversation onready={_ ->
      Network.add_callback(user_update, room)}></>
   <div id=#footer>
      <input id=#entry onnewline={_ -> broadcast(author)} />
      <div class="button" onclick={_ ->
         broadcast(author)}>Post</>
   </>

server = Server.one_page_bundle("Chat",
   [@static_resource_directory("resources")],
      ["resources/css.css"], start)
```

At fewer than 25 lines of code (not including CSS), this app is concise. You can see this application in action at http://chat.opalang.org/ (open the app in two browser windows if there is no one else around). The complete source code (including resources, CSS, and comments) is available at https://github.com/Yoric/OpaChat.

...And More

Clearly, one can't cover all the features of a new language in such a short article. If you want to learn more, go to http://opalang.org, where you'll find a manual, sample apps, and a blog with tutorials and discussions. If you want to peek under the hood and see the implementation of Opa, it is open source and you can download it from GitHub.

—Adam Koprowski is a tech evangelist at MLstate.

Extend MVC Apps Easily Using NuGet Packages

Get started with NuGet – a Visual Studio extension that makes it easy to pull libraries, components, and their configuration data into your project

By Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and Jeremy Skinner

The ASP.NET MVC Framework provides a lot of control over rendering HTML out of the box, but that comes at a cost. The HTML helpers are basic and provide only simple user-interface elements, leaving it up to you to hand-craft UIs using HTML and CSS. Although that's a great option for an experienced Web designer, most developers find third-party components a boost to productivity. They let you develop your application rather than spend lots of time on UI infrastructure.

That's what the NuGet Package Manager is all about. This article teaches you a little bit about NuGet and shows you how to use it to add functionality to a project.

About NuGet

NuGet is a Visual Studio extension that makes it easy to pull libraries, components, and, most importantly, their configuration into your Visual Studio project. This tool is installed with MVC 3 and it is used to bring in various components to make developing with MVC easier. These components are called NuGet packages, and they can include .NET assemblies, JavaScript files, HTML/Razor files, CSS files, images, and even files that can add configuration data to your project's web.config. The goal of NuGet is to make it super-easy to load or update a component.

When you create an MVC3 project in Visual Studio, some NuGet packages are automatically installed. These are jQuery, jQuery UI, Modernizr, and Entity Framework. The reason this is a big deal for you and me is this: jQuery and Modernizr are open-source projects that have frequent releases, much more frequent than the release schedule of ASP.NET or MVC. Because these libraries are included in the default project as NuGet packages, it is insanely easy to update to the latest versions: All it takes is the click of a button. Without NuGet, updating these libraries would have been a manual process of searching for each of the project's Web sites and downloading the files. While this change may seem trivial, it is not. The ability to update and move fast will allow you to begin writing code sooner and spend less time guessing and testing libraries. If you update a library and your tests fail, it is trivial to roll back to the previous version.

NuGet has both a GUI and command-line interface to work with pack-

ages in your projects. First, we will walk through updating a library from the default project template using the GUI.

In the Visual Studio Solution Explorer window, right-click on the Project node and you will see a new context menu: Manage NuGet Packages (Figure 1). Clicking on this menu will bring up the Manage NuGet Packages dialog. The dialog defaults to show packages that are installed in your project and have updates available on the office package source, as seen in Figure 2. The package source is a publicly hosted server that hosts both open-source and closed-source libraries and components.

An Update button shows up for each package that allows you to update the files in your project. Clicking Update for jQuery will make the following actions take place in your project: The old version of jQuery will be removed and the other packages that rely on jQuery will be removed. Then jQuery and the other libraries will be updated. The results of these actions show up in the Results dialog.

The real value that NuGet provides (which has not been described to this point) is how it understands how packages can have dependencies on other packages. The package dependencies could be trivial or complex, but NuGet understands how to deal with that and allows the package authors to specify dependency rules so you don't have to. This is where the real power of NuGet shines through. Before NuGet, this dependency management was communicated through release notes, blog posts, or sometimes never at all. It is these dependencies that can make it painful to use third-party libraries. NuGet replaces this complexity with rules that are implemented by the package authors. The result is a simple experience for developers who would rather write code than debug configuration and dependency issues.

While some of what NuGet does seems like magic, it is a pretty simple process to install and update packages. It is important to understand some basics about NuGet. The most important thing to know is that NuGet will create a folder under your solution file called Packages. Inside this folder, NuGet will download packages and extract some of its

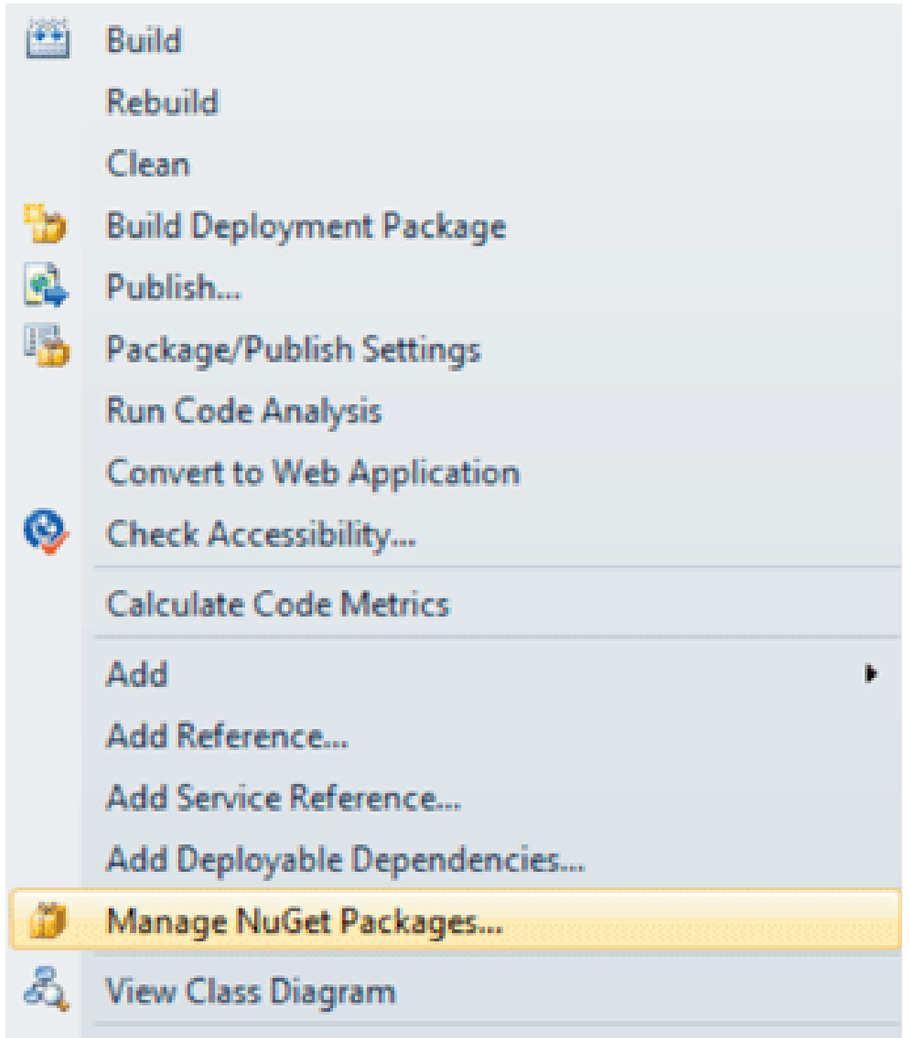


Figure 1: The Manage NuGet Packages menu.

contents into named folders. These folders then get referenced by your projects when the package is installed. The reason this is important is that when using source control, you need to add all the files in the Packages folder into your source control system. Without those files your solution will not compile when a team member pulls down the source code to a different location or machine. Figure 3 shows a listing of the Packages folder created from the default MVC 3 project template.

NuGet brings files into your project in addition to adding them to the Packages folder. NuGet has the ability to add any kind of file to your project.

Now that you have a basic understanding of what NuGet does, we will start using it to add functionality to a project.

Using ASP.Net Webpage Helpers

The ASP.Net team at Microsoft has released a package of helpers that can be used in all ASP.Net applications. These helpers work in MVC but they also work in ASP.Net Web pages as well. The team at Microsoft is able to update and modify these helpers and publish them using NuGet much quicker than they used to when they had to coordinate their release with the entire Visual Studio product. Let's take a look at how to install these helpers using the NuGet console window. Then we will use some of the helpers in a project.

To bring up the NuGet console window, go to the Tools menu, then Library Package Manager, and then Package Manager Console, as seen in Figure 4. This will open a new window in the Visual Studio IDE.

To install a package using the Console, type the command `install-package microsoft-web-helpers`. This will invoke the `install-`

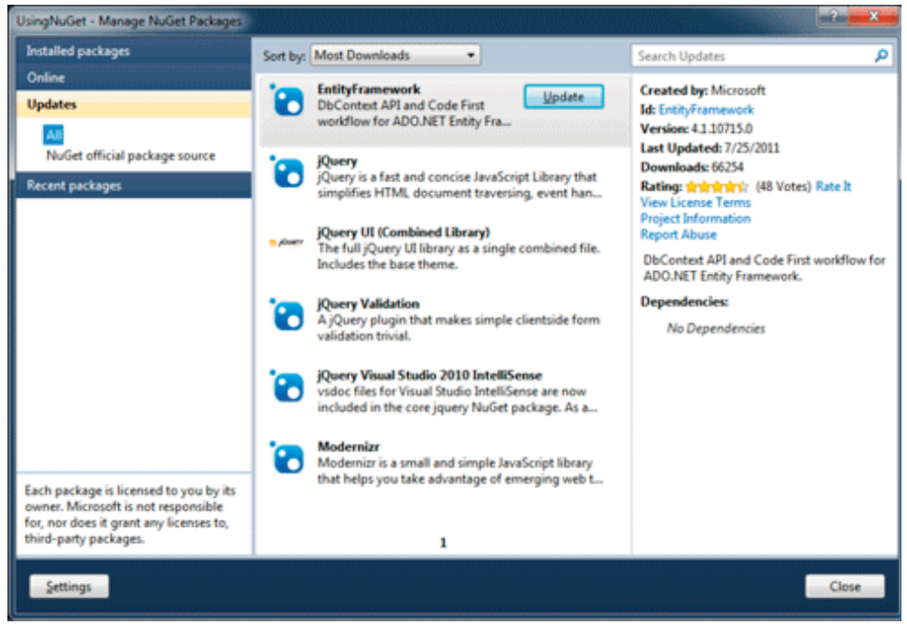


Figure 2: The Manage NuGet Packages dialog.

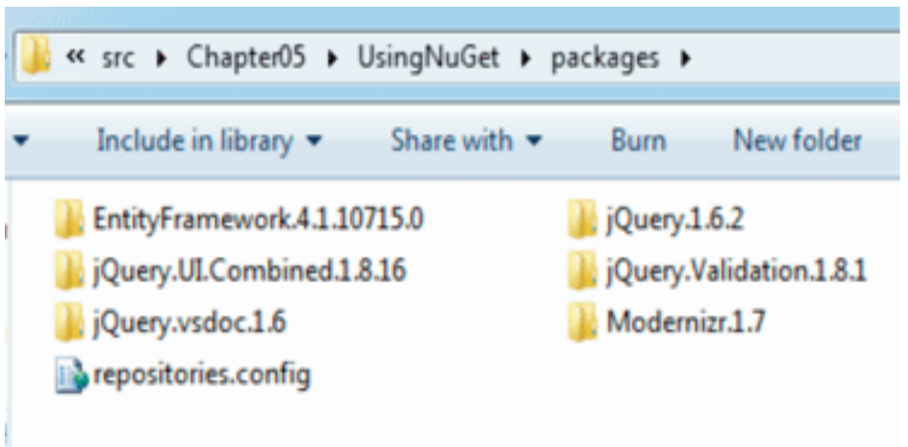


Figure 3: The Packages folder.

`package` command, passing in the package ID `microsoft-web-helpers`. NuGet will download and then reference an assembly in your project. See Figure 5 for the output of the Console window.

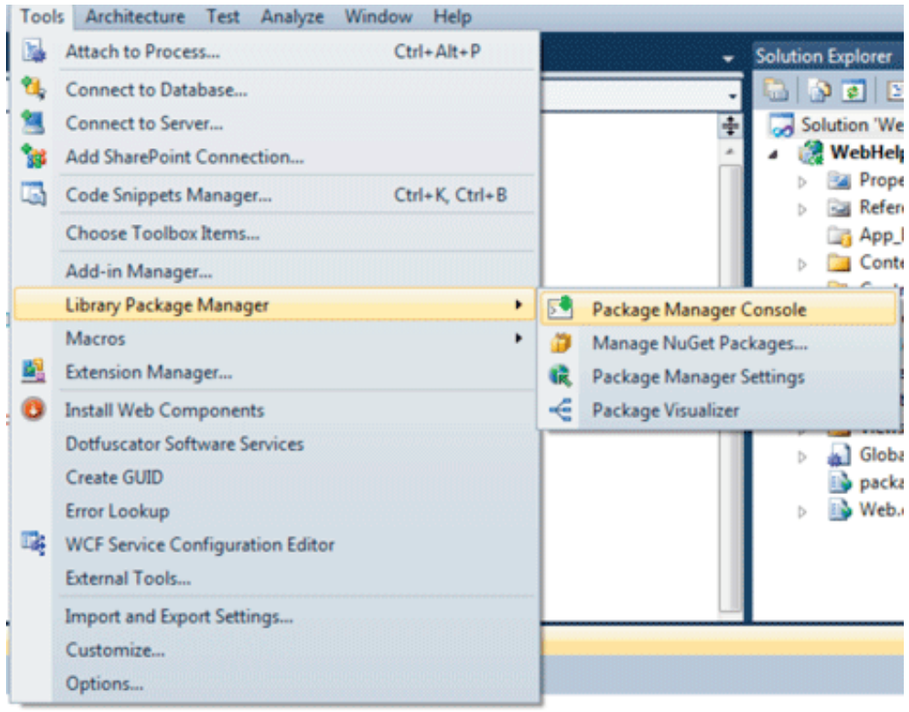


Figure 4: Opening up the NuGet console window.

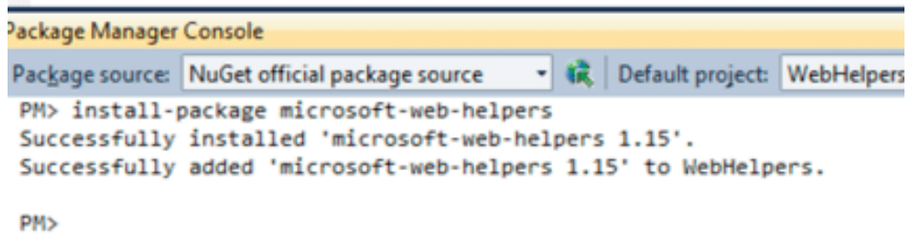


Figure 5: The NuGet Console.

The first example will use the Twitter helper to show a search of Twitter on an MVC View. Create a new View and reference the helpers by adding a `using Microsoft.Web.Helpers` directive. Next, call the Twitter helper, using the `Search` method as shown here:

```
@using Microsoft.Web.Helpers
<h2>@ViewBag.Message</h2>
<p>
@Twitter.Search("MVC3iA",width:800,title:"MVC3 in Action Tweets")
</p>
```

Running this in the browser will show the client-side Twitter widget that queries Twitter for the search term "MVC3iA." See Figure 6 for the screen shot of the page.

This was a really simple way to add some canned functionality into an application with almost no effort. Next, let's look at another helper available in this library. `LinkShare` is a helper that will draw the icons and add links so that a user to your page or site can easily share the URL using the popular social networking sites. You could do this by yourself, but using the helpers lets you do it quickly.

After creating a new Action and View, add the `using` directive to the top of the view code. Use the `LinkShare` helper to create a helper on the View:

```
<h2>LinkShare</h2>
@LinkShare.GetHtml("MVC 3 in Action")
```

Summary

There it is — a quick widget that gives your app social network sharing with a simple helper. Using the code is simple, but the enabler for this is really the power of NuGet and how it makes finding and adding libraries to your project frictionless.

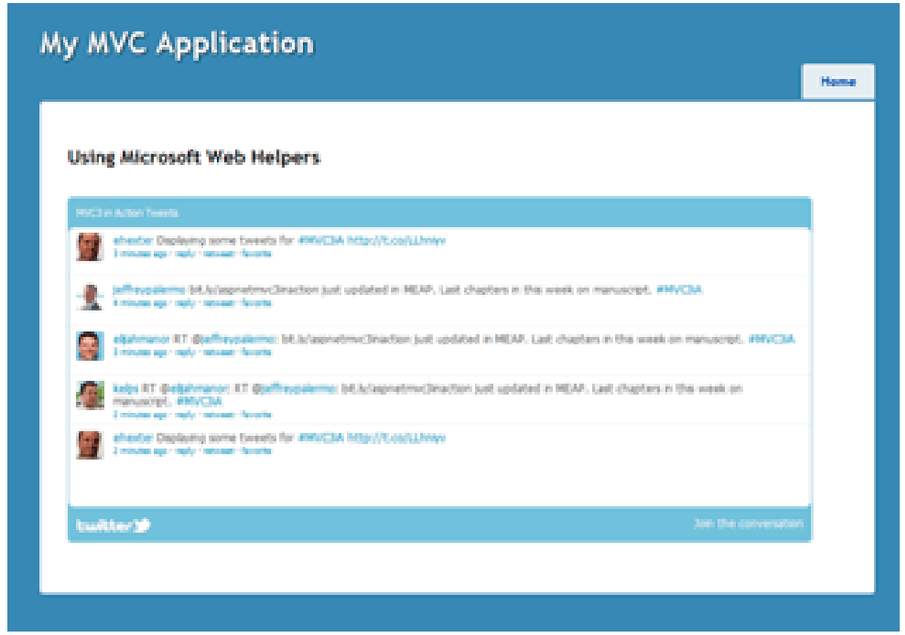


Figure 6: The Twitter search.

Components like Microsoft Web Helpers let us add new functionality to applications and Web pages quickly and easily. They're even easier to use with NuGet, which turns hours of downloading, reading "getting started" docs, and debugging through configuration into a few seconds of automation.

— Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and Jeremy Skinner are the authors of ASP.NET MVC 3 in Action *(http://is.gd/oP4Jo0), published by Manning. This article is based on Chapter 5 of the book.*

Comment

From the Vault

m1: A Mini Macro Processor

Writing for *Dr. Dobb's Journal* back in 2007, noted computer scientist Jon Bentley offered up a simple, Awk-based macro processor that is a surprisingly useful tool for manipulating text files.

— *DDJ*

By Jon Bentley

The UNIX system provides several macro processors. The shell contains powerful mechanisms for text manipulation; the C language has a macro preprocessor; document preparation tools like Troff, Pic, and Eqn all have macros; and the m4 macro language is a general-purpose tool useful in many contexts. m1, a basic macro language, is at least three notches below m4 (it may well be six below, but m2 was too hard to type).

m1's implementation grew from a dozen-line Awk program that provides rudimentary services to a limited but useful two-page program. But why should programmers study a kind of program built in the 1950s? A few reasons I find convincing:

- Macro processors provide a fine playground for learning about programming techniques. (B.W. Kerrnighan and P.J. Plauger devote the final chapter of *Software Tools in Pascal* (http://is.gd/9t2zB2) to the topic.)
- The implementation described here illustrates a number of devices useful in building Awk programs. (This article assumes familiarity with Awk; for more information, see "The Awk Programming Language" (http://cm.bell-labs.com/cm/cs/awkbook/).
- Investigating the design considerations of this simple macro processor can help you appreciate the macro languages you use. (Studies indicate that programmers spend 1.7 percent of their time cursing unexpected side effects of macros.) If those reasons aren't good enough, here's the most convincing argument: programmers have studied macro processors since the beginning of time, so you have to, too. It's a rite of passage.

The Problem

Given that the UNIX system on which I live already has so many macro processors, why did I even consider building one more? Most macro languages assume that the input is divided into tokens by the rules of an underlying language. I recently faced a problem that didn't come in such a neatly wrapped package. I needed to make substitutions in the middle of strings, as in:

```
@define Condition under
  ...
You are clearly @Condition@worked.
```

The first line defines the string `Condition`, and in the second line that string (surrounded by the special `@` characters) is replaced by the text under. Definitions must start on a new line with the string `@define`; the name is the next field separated by whitespace (blanks and tabs), and the replacement text is the rest of the line. Replacements are insensitive to context: the string `@Condition@` is always replaced, even if it is inside quotes or not set apart by whitespace.

A simple macro language like this was sufficient to solve my immediate problem. But once I had it, more applications of the macro processor started to wander across my terminal screen.

The Program

In "Macro Processors and Techniques for Portable Software" (http://sim.sagepub.com/cgi/content/citation/26/4/131-a), P.J. Brown says:

"When attending computer conferences and the like, I have listened to (and probably delivered) my full share of boring lectures, but there is one class of bore who easily outshines all the others: This is the man who talks in full details about the way his system has been implemented."

I'm going to try to outshine even the bores of Brown's nightmares by not only describing the implementation but also giving the complete code. We'll start with the simple version that supports definition and replacement.

Listing One shows the complete Awk program, which contains two pattern-action pairs.

Listing One

```
awk '
/^@define[ \t]/ { name = $2
         $1 = $2 = ""; sub(/^[ \t]+/, "")
         symtab[name] = $0
         next
        }
        { for (i in symtab)
               gsub("@" i "@", symtab[i])
         print
        }
' $*
```

The first pattern recognizes `@define` lines. Its action stores the name, erases the `@define` and name fields and the white space around them, then stores the remainder of the input line in the symbol table (implemented as an Awk associative array). Execution then proceeds with the next input line. The null second pattern ensures that the action will be executed on all other input lines. The `for` loop iterates over all entries in the symbol table, and the `gsub` globally substitutes replacement values for their names. The `print` statement writes the transformed input line.

In the next version of the program we will add a simple include facility. The input line `@include` filename is replaced by the contents of filename. We will restructure the program around a recursive routine to read files and add functions to make it easier to extend.

Listing Two shows the resulting code. If the program is invoked with a single argument, the BEGIN block takes that as the name of the input file; otherwise it processes the standard input. The function `dofile` processes a file, `dodef` processes a definition, and `dosubs` applies the substitutions in the symbol table to its input string. The `dodef` function uses a complex regular expression in a `sub` command to remove the first two fields (because setting them to blanks — as in the first version — causes Awk to replace all field separators with a single blank).

Listing Two

```
awk '
function dofile(fname) {
        while (getline &lt; fname &gt; 0) {
                if (/^@define[ \t]/)
                        dodef()
                else if (/^@include[ \t]/)
                        dofile(dosubs($2))
                else
                        print dosubs($0)
         }
         close(fname)
}

function dodef( name) {
        name = $2
        sub(/^[ \t]*[^ \t]+[ \t]+[^ \t]+[ \t]+/, "")
        symtab[name] = $0
}

function dosubs(s, i) {
        for (i in symtab)
                gsub("@" i "@", symtab[i], s)
        return s
}

BEGIN {    if (ARGC == 2) dofile(ARGV[1])
           else dofile("/dev/stdin")
      }
' $*
```

So far we have assumed that macro definitions expand into unadorned text. But look what happens when the replacement text contains further macro calls, as in:

```
@define DIR/usr/jlb/macro.paper
@define PROBSECFILE @DIR@/sec2.in.
```

After these definitions, the string `@PROBSECFILE@` should be expanded into /usr/jlb/macro .paper/sec2.in. The previous implementation may or may not handle this correctly (details are left as an exercise for Awkophiles). The implementation of `dosubs` in Listing Three handles nested macros by repeatedly expanding the string until no more expansions are made.

Listing Three

```
function dosubs(s,  changes, i) {
        do {
                changes = 0
                for (i in symtab)
                        changes += gsub("@" i "@", symtab[i], s)
        } while (changes)
        return s
}
```

That version is correct but slow; we can speed it up with a guard to check for the common case of no remaining `@` characters:

```
   ...
changes = 0
if (s ~ /@.*@/)
      for (i in symtab)
          ...
```

Without the guard, the program takes 5.4 seconds to process one large file; with the guard, the time drops to 2.3 seconds. The faster version of `dosubs` described in "A Substitution Function" takes just 0.8 seconds on the same file.

A Substitution Function

Several versions of the `dosubs` function perform macro substitution. The final version of the program (Listing Four) uses an even faster version of the function.

The idea is to process the string from left to right, searching for the first substitution to be made. We then make the substitution and rescan the string starting at the fresh text. We implement this idea by keeping two strings: the text processed so far is in `L` (for left), and unprocessed text is in `R` (for right). Here is the pseudocode for `dosubs` (the final version will be shown in Listing Four).

```
L = Empty
R = Input String
while R constains an "@" sign do
        let R = A @ B; set L = L A and R = B
        if R contains no "@" then
              L = L "@"
           break
        let R = A @ B; set M = A and R = B
        if M is in Symtab then
              R = SymTab[M] R
        else
              L = L "@" M
              R = "@" R
return L R
```

Sometimes you want to make a file you can conditionally change. Consider the arduous task of writing a Ph.D. thesis, which can strain even the best professor-student relationship. A friend of mine organized his thesis so that by setting a given flag, he could remove all reference to his thesis advisor. The version he showed his advisor (whom we'll call "Professor Newton" to protect the innocent) was compiled from a file like this:

```
@define WANTNEWT 1
   ...
@if WANTNEWT
This area was profoundly influenced by the groundbreaking work of
Professor Newton.
@fi
```

For his private amusement, the poor student could recompile the document after setting WANTNEWT to zero. The semantics of the `@if` statement are that the text up to the next `@fi` statement is included if the variable is defined and not equal to zero. To implement the statement, we need this function to discard text:

```
function gobble (fname) {
        while (getline &lt; fname &gt; 0)
              if (/^@fi/)
                    break
}
```

We then add these lines to the chain of `if` statements in `dofile`:

```
} else if (/^@if[ \t]/) {
       if (!($2 in symtab) ||
                     symtab[$2] == 0)
                   gobble(fname)
}...
```

The complete m1 program has a couple of additions to this simple conditional. Text may contain nested `if` statements; `gobble` is modified to keep a counter of the current `if/fi` nesting. The `@unless` statement is the complement of `@if`— it includes the subsequent text (up to the same `@fi` delimiter) if the variable is undefined or defined to be zero.

The final version of m1 also supports multiline `@define`s. If a `@define` line ends with a backslash (\), the text is continued on the next line (discarding whitespace before the first text character). To implement long defines, we make the minor change to `dodef` to continue reading text as long as lines end with a backslash. We must also make a major change to the I/O structure of the entire program because macro expansion can generate lines that need to be read by the `dofile` function. The new `readline` function reads a line from the text buffer if it is not empty; otherwise, it reads from the current file. The string `s` can be pushed back onto the input stream by concatenating it on the front (left) of buffer by the idiom `buffer = s buffer`.

The complete program is adorned with several other bells and whistles. Here are the most interesting and important:

- Comments. It is immoral to design a language without comments. Lines that begin with `@comment` are therefore ignored.
- Error checking. The final Awk program has a number of if statements that check for weird conditions, which are reported by the error function.
- Defaults. The `@default` statement is a `@define` that takes effect only if the variable was not previously defined; we'll see its use shortly. We could get the same effect with an `@unless` around a `@define`, but the`@default` is used frequently enough to merit its own command.
- Performance. When `dofile` reads a line of text unadorned with `@` characters, it performs several tests and function calls. The final version adds a new `if` statement to print the line immediately.

`@comment Any text`	`'.`
`@define name value`	
`@default name value`	`Set if name undefined`
`@include filename`	
`@if varname`	`Include subsequent text if varname!=0`
`@fi`	`Terminate @if or unless`
`@unless varnmae`	`Include subsequent text if varname!==0`
`Anywhere in line @name@`	

Figure 1: A summary of the m1 language.

The m1 program could be extended in many ways. Here are some of the biggest temptations to "feeping creaturism":

- A long definition with a trail of backslashes might be more graciously expressed by a `@longdefine` statement terminated by a `@longend`.
- An `@undefine` statement would remove a definition from the symbol table.
- I've been tempted to add parameters to macros, but so far I have gotten around the problem by using an idiom described in the next section.
- It would be easy to add stackbased arithmetic and strings to the language through `@push` and `@pop` commands that read and write variables.
- As soon as you try to write interesting macros, you need to have mechanisms for quoting strings (to postpone evaluation) and forcing immediate evaluation.

Listing Four contains the complete implementation of m1 in about 100 lines of Awk, which is significantly shorter than other macro processors.

The program uses several techniques that can be applied in many Awk programs:

- Symbol tables are easy to implement with Awk's associative arrays.
- The program makes extensive use of Awk's string-handling facilities: regular expressions, string concatenation, gsub, index, and substr.
- Awk's file handling makes the dofile procedure straightforward.
- The readline function and pushback mechanism associated with buffer are of general utility.

Listing Four

```
awk '
function error(s) {
        print "m1 error: " s | "cat 1&gt; &2"; exit 1
}

function dofile(fname, savefile, savebuffer, newstring) {
        if (fname in activefiles)
                error("recursively reading file:  " fname)
        activefiles[fname] = 1
        savefile = file; file = fname
        savebuffer = buffer; buffer = ""
        while (readline()  ! = EOF)  {
                if (index($0, "@") == 0) {
                        print $0
                } else if (/^ @define[ \t]/) {
                        dodef()
                } else if (/^ @default[ \t]/) {
                        if (!($2 in symtab))
                                dodef()
                } else if (/^ @include[ \t]/) {
                        if (NF != 2) error("bad include line")
                        dofile(dosubs($2))
                } else if (/^ @if[ \t]/) {
                        if (NF != 2) error("bad if line")
                        if (!($2 in symtab) || symtab[$2] == 0)
                                gobble()
                } else if (/^ @unless[ \t]/) {
                        if (NF != 2) error("bad unless line")
                        if (($2 in symtab) && symtab[$2] != 0)
                                gobble()
                } else if (/^ @fi[ \t]?/) { # Could do error checking
                } else if (/^ @comment[ \t]?/) {
                } else {
                        newstring = dosubs($0)
                        if ($0 == newstring || index(newstring, "@") == 0)
                                print newstring
                        else
                                buffer = newstring "\n" buffer
                }
        }
        close(fname)
        delete activefiles[fname]
        file = savefile
        buffer = savebuffer

}
```

```
function readline( i, status) {
        status = ""
        if (buffer != "") {
                i = index(buffer, "\n")
                $0 = substr(buffer, 1, i-1)
                buffer = substr(buffer, i+1)
        } else {
                if (getline &lt;file &lt;= 0)
                        status = EOF
        }
        return status
}

function gobble( ifdepth) {
        ifdepth = 1
        while (readline()) {
                if (/^ @(if|unless)[ \t]/)
                        ifdepth++
                if (/^ @fi[ \t]?/ && --ifdepth &lt;=0)
                        break
        }
}

function dosubs(s, l, r, i, m) {
        if (index(s, "@") == 0)
                return s
        l = "" # Left of current pos; ready for output
        r = s  # Right of current; unexamined at this time
        while ((i = index(r, "@")) != 0) {
                l = l substr(r, 1, i-1)
                r = substr(r, i+1)     # Currently scanning @
                i = index(r, "@")
                if (i == 0) {
                        l = l "@"
                        break
                }
                m = substr(r, 1, i-1)
                r = substr(r, i+1)
                if (m in symtab) {
                        r = symtab[m] r
                } else {
                        l = l "@" m
                        r = "@" r
                }
        }
        return l r
}
```

```
function dodef(fname, str) {
        name = $2
        sub(/^ [ \t]*[^ \ t]+[ \t]+[^ \ t]+[ \t]+/, "")
        str = $0
        while (str ^\$/) {
                if (readline() == EOF)
                        error("EOF inside definition")
                sub(/^[ \t]+/, "")
                sub(^\$/, "\n" $0, str)
        }
        symtab[name] = str
}

BEGIN { EOF = "EOF"
        if (ARGC == 1) dofile("/dev/stdin")
      else if (ARGC == 2) dofile(ARGV[1])
      else error("usage: m1 fname")
       }
' $*
```

Applications

According to Kernighan, "Macroprocessors are appealing. They're simple to implement, and you can get all kinds of wondrous effects. Unfortunately, there's no way to tell what those effects are going to be."

A toy example illustrates some simple uses of m1. Here's a form letter I've often been tempted to use:

```
@default MYNAME Jon Bentley
@default TASK respond to your\
  special offer
@default EXCUSE the dog ate my \
  homework
Dear @NAME@:
Although I would dearly love to
@TASK@, I am afraid that I am unable to
do so because @EXCUSE@.  I am sure that
you have been in this situation many
times yourself.
  Sincerely,
  @MYNAME@
```

If that file is named sayno.mac, it might be invoked with this text:

```
@define NAME Mr. Smith
@define TASK subscribe to your \
  magazine
@define EXCUSE I suddenly forgot how \
  to read
@include sayno.mac
```

Recall that a `@default` takes effect only if its variable was not previously `@defined`.

I've found m1 to be a handy Troff preprocessor. Many of my text files (including this one) start with m1 definitions like:

```
@define ArrayFig @StructureSec@.2
&@define HashTabFig @StructureSec@.3
@define TreeFig @StructureSec@.4
@define ProblemSize 100
```

Even a simple form of arithmetic would be useful in numeric sequences of definitions. The longer m1 variables get around Troffs dreadful two-character limit on string names. These variables are also available to Troff preprocessors like Pic and Eqn. Various forms of the `@define`, `@if`, and `@include` facilities are present in some of the Troff-family languages (Pic and Troff) but not others (Tbl). m1 provides a consistent mechanism.

I include figures in documents with lines like this:

```
@define FIGNUM @FIGMFMOVIE@
@define FIGTITLE The Multiple \
  Fragment heuristic.
@FIGSTART@
.PS <@THISDIR@/mfmovie.pic
@FIGEND@
```

The two `@defines` supply the two parameters of number and title to the figure. The figure might be set off by horizontal lines or enclosed

in a box, the number and title might be printed at the top or bottom, and the figures might be graphs, pictures, or animations of algorithms. All figures, though, are presented in the consistent format defined by FIGSTART and FIGEND.

I have also used m1 as a preprocessor for Awk programs. The `@include` statement lets you build simple libraries of Awk functions (though some but not all Awk implementations provide this facility by allowing multiple program files). File inclusion was used in an earlier draft of this article to include individual functions in the text and then wrap them all together into the complete m1 program. The conditional statements let you customize a program with macros rather than runtime `if` statements, which can reduce both runtime and compile time.

The most interesting application for which I've used this macro language is unfortunately too complicated to describe in detail. I wrote the original version of m1 to control a set of experiments. The experiments were described in a language with a lexical structure that forced me to make substitutions inside text strings; that was the original reason for bracketing substitutions by @ characters. The experiments are currently controlled by text files that contain descriptions in the experiment language, data extraction programs written in Awk, and graphical displays of data written in Grap. All the programs are tailored by m1 commands.

Most experiments are driven by short files that set a few keys' parameters and then @include a large file with many `@defaults`. Separate files describe the fields of shared databases:

```
@define N       ($1)
@define NODES   ($2)
@define CPU     ($3)
   ...
```

These files are `included` in both the experiment and Troff files that display data from the databases. I had tried to conduct a similar set of experiments before I built m1 and got mired in muck. But the few hours I spent building the tool were paid back handsomely in the first days I used it.

Looking Back

I've found m1 to be a useful tool in several applications. If it is close to what you want but doesn't meet your exact needs, the code is small enough to be tailored to your application.

Building m1 has been a fun exercise in programming. I started with a tiny program and grew it as applications demanded. It uses several useful Awk techniques and even had some interesting performance problems. All in all, m1 provided me with an almost painless way to learn more about macros, one of the grand old problems of computing.

I am grateful for the helpful comments of Brian Kernighan and Doug McIlroy.

— Jon Bentley is the author of **Writing Efficient Programs, Programming Pearls, and More Programming Pearls: Confessions of a Coder,** ***among other books and papers.***

Comment

This Month on DrDobbs.com

Interesting items posted on www.drdobbs.com over the past month that you may have missed

INTERVIEW WITH HERB SUTTER

The lead architect of the Visual C++ design team discusses implementation of C++11 features, why C99 will never be fully implemented, and the emerging "C++ Renaissance."

http://drdobbs.com/cpp/231900562

TESTING THE FINAL SHA-3 HASHING ALGORITHMS

Testing the finalists in the competition for a new SHA-3 standard shows generally fast, secure hashing algorithms with few collisions.

http://drdobbs.com/security/231900574

AUTOMATE SSH LOGINS BY CUSTOMIZING PUTTY

By touching up the putty source code, you can easily automate secure logins.

http://drdobbs.com/cpp/231700268

BENEATH THE PLANET OF THE ITERATIVE QUICKSORT

You first need to know when it is okay for threads to terminate.

http://drdobbs.com/go-parallel/blogs/architecture-and-design/231901058

CYCLIC CYCLES

If that title seems strange to you, keep in mind that the last few posts I've been writing about error detection. One of the most common ways to handle error detection is the cyclic redundancy check or CRC. (And what could be more redundant than a cyclic cycle?)

http://drdobbs.com/blogs/embedded-systems/231900880

JAVAONE: JAVAFX 2, JAVA ON IOS

Overall, Oracle doubled the space for JavaOne, and attendance was twice what it was last year. Here is an overview of what Oracle has announced at JavaOne 2011.

http://drdobbs.com/blogs/java/231900029

JVM LANGUAGE SUMMIT 2011 VIDEOS

The JVM Language Summit is an annual event sponsored by Oracle, attended by implementers of JVM-based languages and programming tools. Slides for each presentation accompany the videos and are available for download on the individual pages.

http://drdobbs.com/java/231002486

IN THIS ISSUE

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

UBM TechWeb

Dr.Dobb's Business Contacts

DR. DOBB'S

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600-3106, sguttman@techweb.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

CMO, Scott Vaughan
(949) 223-3662, svaughan@techweb.com

VP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Group Sales, Kelly Glass
(212) 600-3327, kglass@techweb.com

Publisher's Assistant, Esther Rodriguez
(949) 223-3656, erodriguez@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Inside Sales Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Sales Assistant, Ian Doyle
(415) 947-6105, idoyle@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Shoshana Freisinger
(415) 947-6349, sfreisinger@techweb.com

Sales Assistant, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Jenny Hanna
(516) 562-5116, jhanna@techweb.com

District Manager, Michael Greenhut
(516) 562-5044, mgreenhut@techweb.com

Account Manager, Cori Gordon
(516) 562-5181, cgordon@techweb.com

Inside Sales Manager East, Ray Capitelli
(212) 600-3045, rcapitelli@techweb.com

Sales Assistant, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

Dr. Dobb's

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600 3106, sguttman@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Marketing Manager, Monique Luttrell
(949) 223-3609, mluttrell@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECHWEB

Tony L. Uphoff CEO

John Dennehy CFO

David Michael CIO

Bob Evans Sr. VP and Global CIO Director

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz VP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing