Web Presentation Layer Bootstrapping for Accessibility and Performance

Clint Andrew Hall
Cerner Corporation
2800 Rockcreek Parkway
Kansas City, MO 64011
011 (816) 201-5045
clint.hall@cerner.com

ABSTRACT

In websites today, most browser incompatibilities are overcome using detection by available client features or the user-agent. This logic is often baked into JavaScript libraries client-side to limit functionality, or clients are filtered server-side to redirect to alternate versions of the site. In this paper, I present a technique called the *Web Bootstrapper*, a technique that allows a developer to write a single site while still providing multiple experiences, or "skins," without altering source or running costly client-side code. It is a process by which an accurate collection of only those static resources and metadata necessary for a unique experience be delivered passively, by the most performant means possible. In further contrast to existing methodologies, this approach determines resources based on capability, form factor and platform by targeting and collecting the often-immutable attributes of the client, not specifically its identity or version. Bootstrapping allows for rule-based, externalized, server-side configuration, further promoting progressive enhancement and client performance.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Portability.*

H.5.2 [**Information Systems and Presentation**]: User Interfaces – *Theory and Methods.*

General Terms

Algorithms, Performance, Human Factors, Theory.

Keywords

Web, User Interface, Web Browsers, Performance, Accessibility, Cascading Styleheets, JavaScript

1. INTRODUCTION

The construction of the typical web page has changed significantly (and for the better) over the last ten years. New and more sophisticated user interface technologies and the availability of higher bandwidth speeds have transformed the once rather bland, text-based documents into entire experiences. Web "pages" have now taken on the role of web "solutions," entities that can accept much more complex user input and thus respond just as richly.

Yet this extremely rapid evolution has not come without cost. Accessibility, or the consideration of those users with disabilities, became secondary to the new visual and input requirements. [1] Rapid development combined with intense competition among countless web clients meant that these very fine-tuned pages started to fall at the mercy of varying implementations. Further complicating matters, the emergence of a broad range of alternate form factors, such as mobile phones, has prompted a conditioned user base to demand similar experiences to that of their desktops. Performance, too, has become a concern, as these rich solutions have begun to buckle under their own weight through the request-response transaction model.

A vocal community of web developers began evangelizing a return to semantic web content, combining it with a technique known as *graceful degradation* [2]. In this approach, pages are first designed for "A" grade browsers, or popular web clients with common support of presentation technologies [3]. In this process, more consideration is given to how the code would be interpreted by clients *without* these technologies, providing alternative decoration or behavior where necessary. Thus, older and lesser-capable browsers are allowed degraded experiences that still function.

More recently, developers have begun turning this trend on its head, preferring the philosophy of *progressive enhancement* [4]. Under a progressive enhancement model, the web page is constructed semantically, based on its content and regardless of its visual end-state, resulting in a lowest-common-denominator, extremely portable and accessible representation. Other presentation layer technologies, such as Cascading Style Sheets (CSS) or JavaScript, are then layered onto this structure, enhancing the experience. At the very least, this allows a developer to support accessibility with the base markup, quickly and easily change the style of a page without altering that content, and include corrections for particular browsers. Progressive enhancement is thus a better approach than graceful degradation because it does not take a "white list" means of providing experiences to lesser-capable browsers; the base markup provides an always-available, semantic view of the content.

In its commonly implemented forms, however, this approach, while offering more benefits than graceful degradation, does not go far enough; it does not effectively address performance or the myriad of potential combinations of browsers and form-factors. In fact, the more portable a document attempts to become, the

W4A2009 - Technical, April 20-21, 2009, Madrid, Spain. Co-Located with the 18th International World Wide Web Conference.

more metadata, CSS selectors and scripts unrelated to the desired experience are downloaded and included in the document. Indeed, many CSS and JavaScript frameworks consist a great deal of compatibility code that is ignored unless applicable to that particular browser, (and is thus wasted in those contexts). [5] Older versions of browsers—considered a less-than-significant demographic—are often ignored, allowing sites to fail visually when the clients cannot interpret the CSS selectors and script correctly.

There have been attempts to optimize the potentially excessive or incompatible code. Server-side techniques use unreliable request headers and can only make assumptions about client-side capability. Client-side approaches push detection and inclusion logic to the browser, often using the same unreliable request header and rarely taking the form factor into account. Neither process is configurable in and of itself, and both require changes to source code when a new skin is created.

In computing, *bootstrapping* ("to pull oneself up by one's bootstraps") refers to techniques that allow a simple system to activate a more complicated system [6]. In this paper, I introduce *Web Bootstrapping*, a process by which an accurate collection of only those static resources and metadata necessary for a unique experience be delivered passively, by the most performant means possible. In further contrast to existing methodologies, this approach determines resources based on capability, form factor and platform by targeting and collecting the often-immutable attributes of the client, not specifically its identity or version. Bootstrapping allows for rule-based, externalized, server-side configuration, further promoting progressive enhancement and client performance. Individual presentation collections, or *skins*, can be edited independently of each other, and new collections can be added at run-time without changing any source code of the document. The bootstrapper also supports on-demand resource inclusion, (e.g. Ajax) with identical capability consideration. As an added benefit, by virtue of being a server-side approach, this process is also capable of concatenating static resource content from remote sources, thereby avoiding cross-site scripting and mixed-content warnings.

Using an implementation of this bootstrapping method, we have been able to demonstrate several solutions with different skins based on browser and device. In point of fact, we have yet to find a web-enabled device incapable of displaying these bootstrapped pages. We've also seen significant performance improvements in our web solutions, and we continue to experiment with this technique.

1.1 Background and Related Work

1.1.1 Semantic Markup

One of the most important philosophical shifts in web development came with the return to the use of HTML as an expressive markup language *only* [7]. Before the advent of CSS, most documents had their visual appearance embedded into their very structure. The release of the HTML 4 specification, however, deprecated these visual tags in favor of those with a defining nature; *strong* as opposed to *bold* and *emphasis* opposed to *italics*, for example [8]. Pages built in ways that took advantage of these meaningful tags thus became more meaningful themselves, both to parsing algorithms—search engines, browsers and screen readers—and to users. This also led to greater flexibility using presentation layer technologies such as CSS and JavaScript.

1.1.2 Flexing Presentation Layer Technologies

Cascading Style Sheets (CSS) and JavaScript were created for developers to enhance the user experience of a web page, preferably without altering the semantic content of the document. Once web clients began supporting these technologies, web pages took on a much more layered structure, one where the separation of concerns—content, presentation and behavior—was becoming obvious.

Yet the fact that different manufacturers implemented these technologies in each web client meant there were bound to be inconsistencies between them, (often despite the existence of standards). Furthermore, the eventual demand for web clients on other form factors, such as mobile phones, gaming platforms and handheld devices, meant that the support of these presentation layer technologies could vary greatly depending on conditions such as bandwidth, memory, processing power, size, input sources, haptic feedback, etc. To create a consistent experience in such conditions, developers began to look for ways to flex the presentation layer.

1.1.2.1 User-Agent Detection

An initial obvious answer was the use of the *user-agent string*, a commonly populated header on the HTTP request identifying the browser to the server. Unfortunately, the most common use of this identifier by developers was to blacklist browsers from sites, either because they believed the site would not appear properly, or to promote the use of one client over another. The intense competition between browser manufacturers combined with the lack of any enforcement upon the validity of the user agent string prompted both the inclusion of deceptive strings by vendors, as well as the spoofing of these strings by users. Indeed, some web browsers, such as the Opera Software's Opera Web Browser, allow the user to either spoof, or mask the user-agent string with that of any other browser through easily accessible user configuration. As a consequence of all of these factors, identifying a client by its user agent string for the purposes of flexing its content, presentation, or behavior became an unreliable, discouraged practice [9].

1.1.2.2 CSS Media Typing

Standards bodies like the W3C gathered feedback from manufacturers, developers and users, thus developing several proposals, the most promising being *media typing* for CSS. In this standard, developers could target either portions, or the entire CSS document for a specific display type, such as handheld, print, screen, TV, braille, etc. When implemented properly, within the web client, these media types could omit or include certain portions of CSS, presumably those that would be inappropriate for the device.

Unfortunately, and almost expectedly, the implementation of media type within different web browsers is a mixed. In some cases, mobile browsers honor both handheld and screen media types; some ignore handheld media types altogether. One speculates this is due to a self-enforcing cycle: developers don't use them because devices don't support them, thus devices don't support them because developers aren't using them.

1.1.2.3 JavaScript Frameworks

In further effort to insulate developers from the various inconsistencies between web clients, several JavaScript frameworks have been developed and are used with some prominence. Libraries such as jQuery [10], Dojo [11] and

Prototype [12] can be very useful, as they abstract away the different implementations of common tasks such as object manipulation and event handling, providing a unified set of functions in a number of popular clients.

While helpful in many cases, these libraries have several drawbacks. First, these libraries attempt to level the playing field between the modern, the eccentric and the lesser capable of browsers. Consequently, web clients are forced to download compatibility code for the others, regardless of if it applies or not. Further, these libraries are restrictive in that they only target a finite set of web clients [13]; most don't support mobile due to the devices' limited implementations of JavaScript.

When used thoughtfully, JavaScript frameworks can provide quick, powerful access to most features of the browser. Care must be taken to ensure that bandwidth and processing is not wasted when the site is intended for a broader range of devices and browsers.

1.1.3 Transcoding

Popular "turn-key" approaches for businesses wishing to adapt to the mobile market and accessibility concerns often use transcoding to deliver alternate markup of their existing site.

1.1.3.1 HTML Compression and Adaptation

One approach involves reformatting or compressing existing web content into something more palatable to a smaller or less powerful platform. By creating a set of intelligent rules, large portions of "unnecessary" HTML content can be removed, distilling information as accurately as possible. Several solutions exist [14], both on the server and on the client, but each relies upon complex logic to succeed.

1.1.3.2 SADIe

A more recent approach called "SADIe," short for Structural-Semantics for Accessibility and Device Independence [15], uses the information contained in the CSS class names to develop a structural ontology. Once loaded into an enabled client, the ontology is read and the content transcoded, creating a more accessible view of the page.

The approach relies on knowledge of the page structure and the semantic web to create an appropriate ontology. While much more effective than rule-based output transcoding, this approach still relies on a set of rules to adapt the content, as well as a knowledgeable architect to create the ontology.

1.1.4 Performance

Many web architects have realized the savings, both monetarily and through performance, optimizing the web layer of their solutions. Through the work of architects like Steve Souders [16], a number of effective best practices have been and continue to be identified [17]. Several of these, such as reducing the number of HTTP requests, minifying JavaScript and externalizing resources, can contribute significantly toward faster responses in the web client.

Even the order by which resources are loaded can have a profound effect on page completion times. It has been demonstrated that, in some cases, JavaScript can in fact block entire page execution as the script is interpreted [18]. Understanding the environment in which these resources are downloaded, the effects of their inclusion and the best means by which to incorporate them, will continue to be pivotal in creating performant sites.

1.1.4.1 Cuzillion

Cuzillion is a recently released performance tool written by Steve Souders [19]. This tool can be used to effectively evaluate different resource load configurations in different browsers. Cuzillion has been instrumental in determining the most effective load configurations within the bootstrapper, and I encourage readers to use it to evaluate their own scenarios.

1.2 Contribution of this work

The goals of the Web Bootstrapper project were as follows:

- Promote Progressive Enhancement and Accessibility;
- Target the capabilities of the client, not the identity of the client alone, to deliver a defined experience;
- Support remote configuration;
- Support performance best practices;
- Allow for the collection of precise demographic data.

The remainder of this section will summarize how the bootstrapper meets these goals. Section 2 will discuss the details of the process. Section 3 will discuss the specific implementation of the bootstrapper within our solutions. Section 4 will discuss the real-world results of using this process in our solutions and prototypes.

1.2.1 Expanding the Definition of Accessibility

Traditionally, accessible technology is defined as "[technology that] can be used as effectively by people with disabilities as by those without" [20]. Yet when we consider the portable nature of the web, does this definition go far enough? Should anyone, with or without physical disability, be compelled to have the most up-to-date hardware and software to use a web site? Could such a requirement be considered an economic or educational bias?

Considering this, the traditional definition of accessibility could be expanded: "Technology is considered accessible if it can be used as effectively by people with disabilities as by those without, *as well as by those with less-than-modern means.*" Indeed, progressive enhancement can ensure that the content of a site is portable to any browser capable of negotiating an HTTP request and parsing HTML. Flexing the presentation layer based on the physical characteristics of the device, such as the size of the screen, could certainly assist in delivering a more accessible experience. Reducing the number of static resources downloaded to only those absolutely required could also save bandwidth and processing power, resources that come at a premium on devices like mobile phones. Further, individuals with disabilities that choose to disable JavaScript can avoid downloading resources they do not require.

The bootstrapper enables alternate presentation experiences through its methodology, thus more effectively supporting both the traditional definition of accessibility as well as the expansion offered above.

1.2.2 Web Client Agnostic Content

As demonstrated previously, the number of web clients and the diversity of their features have contributed to a complicated landscape for web developers to conquer. Thankfully, modern desktop browsers vary more in their top-level features than they do in rendering methodologies; some share rendering engines, such as Gecko or WebKit, while most at least attempt to follow standards consistently. When scaling down to mobile devices,

some engines, such as WebKit on the Apple iPhone, have been able to do so effectively; other mobile browsers opt for a subset or a new engine entirely.

When considering the depth of the browser market combined with the breadth of available devices and platforms, User Agent detection can be daunting, expensive and worse still, error-prone. JavaScript and CSS frameworks have taken cross-browser compatibility only so far; they primarily focus on the desktop and do come with a cost.

Rather than interrogate the User Agent string, the bootstrapper applies rules such as, "if the screen size is larger than 800x600, deliver the following resources," or, "if the client supports Java and Adobe Flash, deliver the following script to inject components." Even proprietary means of version detection—such as conditional comments in Internet Explorer—can be exploited for rules processing.

The bootstrapper gives a developer reliable access to a host of client-side information and can thus apply generic, intelligent and performant rules to the delivery of resources.

1.2.3 Configuring Agile Flexibility

In many organizations, an operations team manages the configuration and day-to-day maintenance of sites. As might be expected, the operations team may not be as familiar with the inner workings of the code, nor is it prudent to edit production code directly. Therefore, it is very important that the development team provide a fast, hands-off configuration mechanism before the release. This is often accomplished through individual configuration files.

Server-side configuration options for tasks such as caching, preferences, data sources and the like can be very impressive; yet the presentation layer has very little in terms of options. And, in truth, why should it? The presentation layer of web solutions should be fairly baked and final from release to release.

The trouble arrived as browser vendors, eager to appeal to consumers in a Web 2.0 world, began to release new versions of their software more often than before. With these new releases came fixes for existing bugs, but also new features… often a source for new bugs. Emerging devices with higher bandwidths and better web capabilities began to appear, as did a savvier user base expecting a harmonious, albeit entirely new, experience for our solutions.

As our presentation layer resources are packaged with the other code in the solutions, we began to struggle with the fact that we could not certify and release code quickly enough to meet these browser and device releases. Thankfully, with the separation of presentation layer technologies from the semantic markup, these skins have become much less brittle. At runtime, the bootstrapper allows developers to create, repair or remove both the skins and the rules that provide them without requiring a code release. This gives us greater flexibility toward the browser landscape, as well as agile reaction to any changes within it, all the while coexisting effectively in our release cycles.

1.2.4 Improving Performance

The method by which static resources are included can have a drastic effect on how quickly the page is loaded. The bootstrapper assists by providing a configuration option to the inclusion of static resources. It allows the developer to favor a particular resource delivery preference—*HTML*, *Network* or *Concurrency*—at runtime, thus the bootstrapper can adjust to the changing state of the presentation layer and provide optimal performance.

1.2.5 Data Collection

Several tools exist today, such as Google Analytics, which collect user demographics and provide analysis of visits to a site [21]. Much like the bootstrapper, these tools rely on a JavaScript interrogation of the client followed by a transmission of this data to the server.

Since the bootstrapper can perform this collection to determine static resources, so too can it store and interpret these attributes for the purposes of demographic analysis.

2. WEB BOOTSTRAPPER APPROACH

The following outlines in detail how the bootstrapper is set up and how it executes to effectively deliver static resources and content.

2.1 Setup

The first step in using the bootstrapper is defining the unique experiences to be delivered. These skins will be attributed to entire collections of devices based on their attributes and classified by the bootstrapper ruleset. Table 1 defines a sample set of experiences.

Table 1. Sample set of Experience Definitions

Skin	Description
Plain text	Default skin, devoid of presentation or behavior other than applied by the client.
Mobile	Suitable for small screens; minor subset of CSS, minimal script.
Desktop	CSS 3, Javascript 1.5 compliant skin.

Each of these experiences has their own set of resource files and metadata.

Second, the bootstrapper is provided a set of rules capable of classifying clients based on their attributes. Using Table 1, we can see that the most prominent rule is screen size, followed by discerning the difference between WebKit and other mobile devices. Figure 1 is a pseudo-code representation of a rule file that could support the experiences in Table 1.

```
# PSEUDO-CODE RULE FILE

platformClass = null;

rule "Desktop Platform"
salience 100
no-loop true
    when
        platformClass == null and
        ATTRIBUTES.screenHeight > 600 and
        ATTRIBUTES.screenWidth > 800
    then
        staticFiles.addJavascriptFile("/com/cerner/resources/javascript/desktop.js");
        staticFiles.addCascadingStyleSheet("/com/cerner/resources/css/desktop.css");
        platformClass = "DESKTOP";
end

rule "Mobile Platform"
salience 100
no-loop true
    when
        platformClass == null and
        ATTRIBUTES.screenHeight <=600 and
        ATTRIBUTES.screenWidth <= 800
    then
        staticFiles.addJavascriptFile("/com/cerner/resources/javascript/mobile.js");
        staticFiles.addCascadingStyleSheet("/com/cerner/resources/css/mobile.css");
        platformClass = "MOBILE";
end
```

Figure 1. Pseudo-code Rule File

Third, each page within the solution includes a single JavaScript include that triggers the bootstrapper. Optionally, the source URI of this include can specify a *bundle*, a configuration key that

corresponds to a group of pages that share a set of resources. For example, suppose several pages use the same set of resources because they all deal with a similar topic, such as allergies. These pages could then share the same bundle name, "allergies," which would direct the bootstrapper to deliver only those resources necessary.

Finally, the resource delivery preference for the solution is configured. A bootstrapper delivery preference defines how the resources are loaded; the selection of the method depends on the number and size of the resources as well as the requirements of the HTML.

There are three preferences currently employed by the bootstrapper: *HTML*, *Network* and *Concurrency*.

2.1.1 Favoring HTML

This mode of resource delivery is the least performant, but most closely mimics how the resources are included at run time. In this case, resources are written directly to the HEAD as the page is interpreted. This can be slower, as HTML and other code following these JavaScript elements are usually blocked as the elements complete [22].

This methodology could be chosen in those cases where the HTML contains JavaScript that relies on functionality included before the load of the page is completed.

2.1.2 Favoring the Network

This mode of resource delivery is faster than the HTML method, as it concatenates and delivers all specified JavaScript in one network transfer. In addition, it adds inclusions via Document Object Model (DOM) manipulation; this allows resources to be downloaded concurrently.

This method assumes there does not exist any inline script within the document that must be executed before the document finishes loading. This method would also be preferred when there is very little JavaScript to load, thus favoring the network by not firing off any additional network traffic to load resources.

2.1.3 Favoring Concurrency

In some cases, the most performant method of resource delivery favors concurrency. In this mode, the bootstrapper "phases" resource loading using two passes. In the first pass, CSS and metadata are added to the document, followed by an include requesting the second pass. In the second pass, JavaScript is added to the document. In this way, the presentation of the page is handled first, followed by its behavior.

While involving an additional request to the server, this method defers the processing of JavaScript to run concurrently with the download of other resources. This can be extremely effective when larger amounts of JavaScript are required.

2.2 Page Load Execution

Once the solution has been prepared, the bootstrapper process can execute. Figure 2 provides a visual representation of the bootstrapper methodology. The process executes as follows:

1. An HTML document is served with one static script include:

   ```
   <script type="text/javascript" src="/Bootstrap"
   type="text/javascript"></script>
   ```

   or, optionally:

   ```
   <script type="text/javascript"
   src="/Bootstrap?bundle=[bundleKey]"></script>
   ```

   where `?bundle=[bundleKey]` is an optional grouping identifier.

2. The `/Bootstrap` URL references a server-side process which delivers a configured bootstrapper JavaScript capability detection object.

3. The bootstrapper object attempts to collect a number of attributes from the client.

4. The bootstrapper object appends a SCRIPT tag to the HEAD, resulting the following addition:

   ```
   <script type="text/javascript"
   src="/Bootstrap?r=1[&attribute=value...]>
   </script>
   ```

   where `r=1` informs the server-side process the bootstrapper object has gathered attributes and is ready to receive resources.

5. The `/Bootstrap` URI maps to the server process, which then passes the request parameters and the `user-agent` to a rules engine.

6. A rule set is evaluated, resulting in a list of metadata, CSS and JavaScript file paths to be returned to the client.

7. The process appends, in order, metadata, CSS and JavaScript include script based on the configured favoritism. The bootstrapper object interprets these calls, resulting in additions to the document.

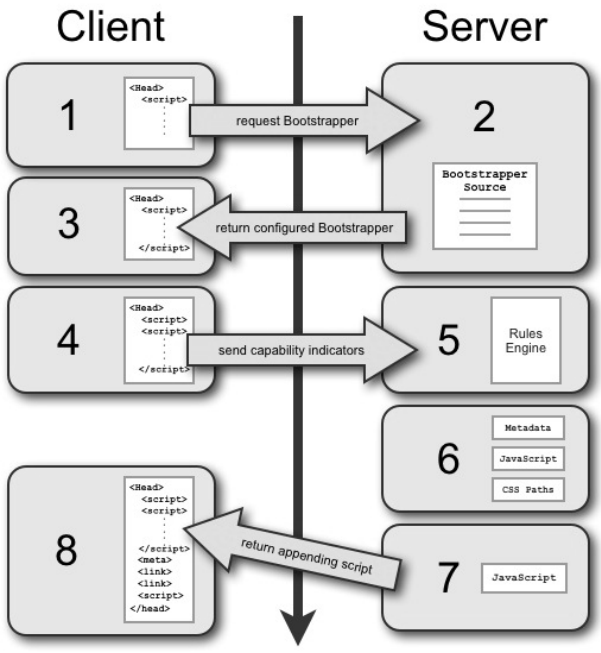

Figure 2. Bootstrapper Execution

2.2.1 The Bootstrapper Payload

The Bootstrapper JavaScript object is responsible for collecting, transmitting and appending resources to the document. The object can be implemented to look for any number of attributes to transmit to the server-side process. The attributes collected from the bootstrapper are output as SCRIPT elements, typically within the HEAD. The script that arrives as a result of the bootstrapper

contains calls to its API that are responsible for appending elements to the document.

2.2.2 Resource Collection

The server-side process is provided with a collection of paths of resources from the rules engine. The resources can be local or remote.

Local JavaScript resources can be concatenated together if they are accessible from the process, (e.g. from a file stream). If not found, the paths are output directly to the HEAD, and are thus relative to the URI of the page. As they can contain references to relative images, local CSS resources are always output relative to the page URI.

Remote JavaScript resources can be concatenated with local JavaScript resources if the server side process can support loading these files via HTTP. This is particularly useful in order to avoid cross-site scripting warnings, as the source of these files originate within the domain of the solution, rather than remotely. As with local CSS resources, remote CSS paths are output as specified.

2.2.3 Resource Delivery

As previously outlined, the bootstrapper currently supports three resource delivery preferences. In each case, the bootstrapper JavaScript object provides API methods for resource inclusion.

2.2.3.1 Favoring HTML

When favoring HTML, the JavaScript files found by the server process are concatenated together and returned with the rules engine result. Metadata, CSS file paths and the paths of JavaScript files not available to the server are subsequently written to the document using `document.write`. Since `document.write` applies content to the document as it being process, this method matches exactly how the document would be interpreted if the includes were a part of the HEAD element source.

2.2.3.2 Favoring the Network

When favoring the Network, the JavaScript is still returned with the bootstrapper rules engine result, but following CSS and metadata is appended to the HEAD with DOM manipulation. This is done by creating elements, setting their properties and appending them to the HEAD node.

2.2.3.3 Favoring Concurrency

When favoring concurrency, the bootstrapper rules engine result only returns calls to append CSS and metadata to the HEAD, then a single JavaScript element is appended to the HEAD which requests the JavaScript portion.

2.2.4 Clients without JavaScript

The bootstrapper process will obviously not execute within clients without at least rudimentary JavaScript support. While an avoidance of any presentation-layer resources within these clients would be recommended, it may be a requirement to have a base CSS skin applied. In such cases, the bootstrapper could return a simple CSS "reset" file—one that redacts incompatible selectors by overwriting them. The same assumption could be applied to metadata.

2.3 On-demand Execution

On-demand inclusion, or the loading of resources without a page refresh, executes in much the same way as the on-load process. The only exception to this is that the favoritisms do not come into play; all resources are added to the HEAD using DOM manipulation. This method can be used to further increase performance by loading only that JavaScript and CSS necessary for the result of a behavior. The bundle attribute can be used to great effect in this use case.

3. IMPLEMENTATION

Our implementation of the bootstrapper uses Java as its language, the J2EE container and JBoss Drools as the rules engine.

3.1 The Bootstrapper Payload

In our implementation, the bootstrapper payload has been configured to gather and transmit the following attributes:

- Screen Height, (`screen.height`)
- Screen Width, (`screen.width`)
- IE Version, (if IE, using conditional comments)
- Color Depth, (`screen.colorDepth`)
- Java Enabled, (`navigator.javaEnabled`)
- Platform, (`navigator.platform`)
- Vendor, (`navigator.vendor`)

In most cases, these attributes are well supported, implemented as immutable within the client, and are fairly reliable. The absence of any attribute, however, need not halt execution; indeed, well-implemented rules can interpret and handle such cases.

3.2 A Bootstrapper Servlet

The server-side component of the bootstrapper in our implementation is a J2EE servlet. This servlet has access to any local static resources on its CLASSPATH; this includes both JARs and the web container itself. We have also implemented remote resource access through the Apache HTTPClient. [23]

3.3 JBoss Drools Rules Engine

We chose to implement connect an instance of the bootstrapper servlet to the JBoss Drools Rules Engine. [24] This enabled an expressive, static file-based rules language that could be externalized on the server and read at runtime.

3.4 Other POJOs

Several POJOs (plain old Java objects) were also created to facilitate transfer of information between the servlet to the rules engine.

4. RESULTS

We continue to apply and refine this technique. One of the primary goals of this paper is to introduce the concept and encourage vetting of its appropriateness. This section will detail some of the specific results we have seen when using this technique.

4.1 Supporting Multiple Experiences

At the 2008 Cerner Health Conference, we were able to demonstrate a bootstrapper-enabled prototype called *Activity Tracking*. The Activity Tracking prototype is Cerner's vision of using connected devices, such as a pedometer, scale or glucose monitor, to upload relevant biometric data to a consumer's health record. [25]

The goal was to produce a prototype that could flex its look-and-feel based on the device. Using the bootstrapper, four skins were

produced: plain-text, mobile, desktop and WebKit mobile. Figure 3 is a collection of screenshots from the Activity Tracking solution.

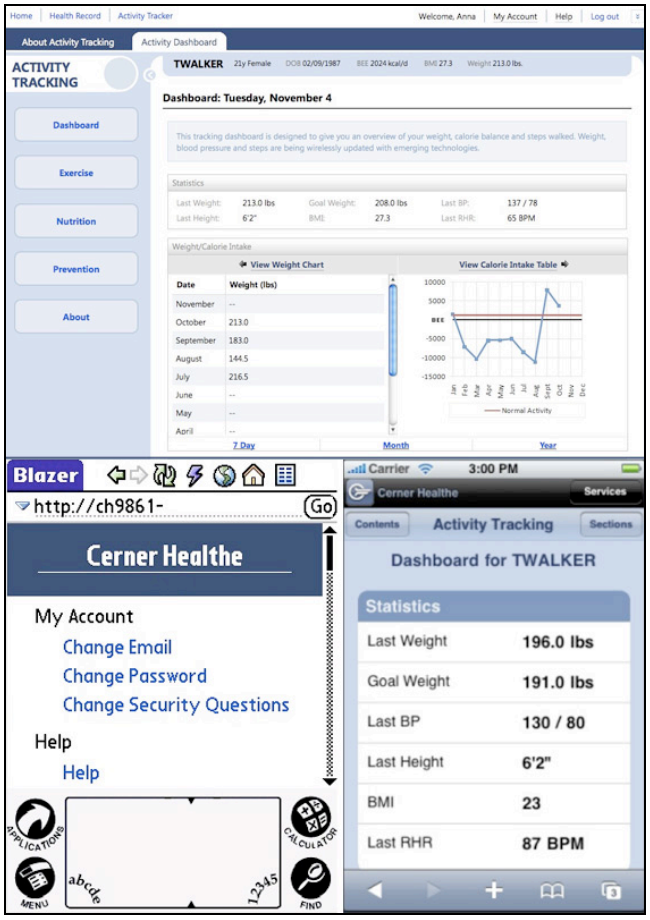

Figure 3. Screenshots of Activity Tracking Skins (clockwise from top: desktop, mobile WebKit, mobile)

4.2 Accessibility

The most obvious benefits to this approach are the accessibility features inherently included. In order to bootstrap a site effectively, the markup must be semantically constructed using progressive enhancement. Only under these circumstances can multiple experiences be created without refactoring. In doing so, the developer ensures that those with lesser or assistive technology can use the base site.

The bootstrapper can also be used to deliver skins with features such as high-contrast, alternate color and/or alternate fonts based on attributes gathered from the client. While some sites do offer alternate stylesheets for this functionality, the bootstrapper can apply these sheets without interaction from the user.

4.3 Performance

We've run several performance tests similar to those in Cuzillion. These test results demonstrate just a few of the performance best practices outlined by Souders; their inclusion here is intended to demonstrate how the bootstrapper incorporates some of these practices into its configuration.

In each example, particularly around Network and Concurrency versus HTML, we've seen significant performance improvements over loading through the bootstrapper, as opposed to loading resources in-source.

4.3.1 Local Resources: Network vs. Concurrency

When loading local resources, the Network delivery preference loads a concatenation of JavaScript code upfront. The Concurrency preference, however, favors a second pass for the concatenated JavaScript appended to the `HEAD`. In this test, the bootstrapper JavaScript resource load is delayed five seconds to simulate a large or delayed JavaScript file.

In Figure 4, the bootstrapper is using the Network load preference. Notice that the JavaScript is concatenated into the rules engine response, and due to its size, it blocks the page.

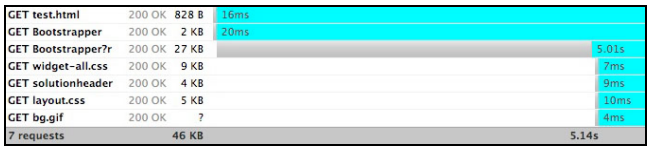

Figure 4. Loading Local Resources Favoring the Network

In Figure 5, however, an additional network transaction is appended to the `HEAD` to load the JavaScript. Notice that the excessive script does not block the page content and CSS.

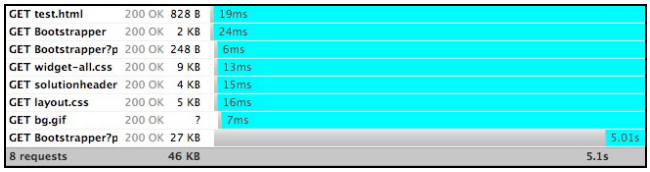

Figure 5. Loading Local Resources Favoring Concurrency.

While the difference in load time is negligible, the CSS is being blocked when using the Network load preference. As a result, the presentation of the page "freezes" while the script is loaded.

4.3.2 Remote Resources: HTML vs. Concurrency

When loading remote resources, the HTML and Concurrency resource delivery preferences showed the greatest contrast. In this test, three remote CSS files and three remote JavaScript files were loaded, each with a delayed load time of one, two and three seconds, respectively.

The HTML loading method simulates precisely the effects of including the resource tags within the HEAD element. In Figure 6, we can clearly see the later JavaScript includes blocking as they are interpreted, contributing to a total load time of 8.5 seconds.

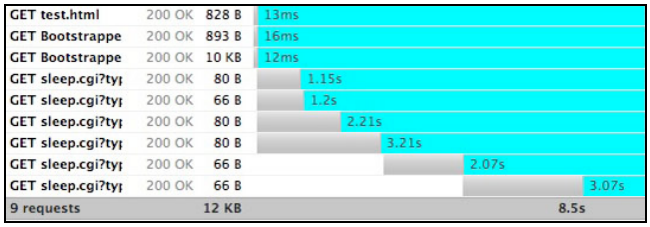

Figure 6. Loading Remote Resources Favoring HTML

In Figure 7, the concurrency model allows the same JavaScript to be loaded concurrently with the CSS files, reducing the load time to 3.3 seconds. Notice how the JavaScript file transactions are not blocking; the one, two and three second files are loaded concurrently, resulting in a significant improvement in load time.

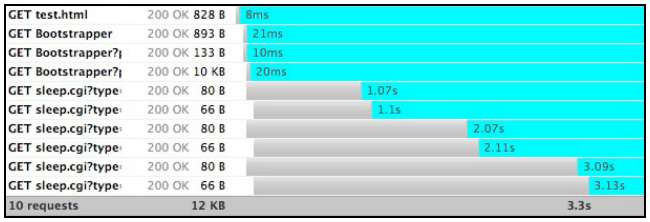

Figure 7. Loading Remote Resources Favoring Concurrency

4.4 Logging

The bootstrapper currently, through our internal logging methodologies, collects statistics on the web clients accessing the bootstrapper at any given time. Figure 8 is an example of the bootstrapper logging client attributes.

```
[2008-11-03T23:53:25] [TRACE] [btpool0-3] [com.cerner.web.bootstrapper.servlet.BootstrapperServlet]
 + Message: Bootstrapper Servlet Request
   Request Detail --
        Response: Process attributes and deliver resources.
        Methodology Favors: NETWORK
        Configured Host Path: none

   Client Attributes --
        User Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.0.3) Gecko/2008092414 Firefox/3.0.3
        Bundle: 0
        Screen Size: 1200x1920
        Platform: MacIntel
        Vendor: 0
        Color Depth: 24
        Java Enabled: true
        IE Version: 0

[2008-11-03T23:53:30] [TRACE] [btpool0-3] [com.cerner.web.bootstrapper.servlet.BootstrapperServlet]
 + Message: EXITING: com.cerner.web.bootstrapper.servlet.BootstrapperServlet#service()
```

Figure 8. Sample Console Output for a Network Request

5. FUTURE WORK

The purpose of this paper was to outline the current state of both the problem and the solution the bootstrapper proposes. There are several directions for further study, including:

- Incorporation of further presentation layer best practices;
- Exploration of other resource delivery preferences;
- Incorporation of a threaded HTTPClient for remote resource inclusion;
- Configuration of resource delivery preference per resource transaction.

I sincerely encourage questions and feedback on the Web Bootstrapper. We will continue to expand and experiment with it to further explore its potential.

6. ACKNOWLEDGEMENTS

My sincerest thanks go to the management and associates of Cerner for their support of this project. Many thanks also to Steve Souders for his impressive contribution in knowledge and tools to the web performance space.

7. REFERENCES

[1] WebAIM. Introduction to Web Accessibility http://www.webaim.org/intro/

[2] The Web Standards Project (WASP). Manifesto – The Web Standards Project http://www.webstandards.org/action/dstf/manifesto/

[3] Yahoo, Inc. Yahoo! UI Library: Graded Browser Support http://developer.yahoo.com/yui/articles/gbs/

[4] Wikipedia. Progressive Enhancement http://en.wikipedia.org/wiki/Progressive_enhancement

[5] Gilzow, Paul. 2008 Javascript Frameworks. Interface: the Official Blog of Web Communications at the University of Missouri. http://interface.missouri.edu/2008/04/javascript-frameworks.php

[6] Wikipedia. Bootstrapping http://en.wikipedia.org/wiki/Bootstrapping

[7] Shea, Dave. css Zen Garden: The Beauty of CSS Design http://www.csszengarden.com/

[8] W3C. 1999 HTML 4.01 Specification. Section 15: Alignment, font styles and horizontal rules in HTML documents. http://www.w3.org/TR/REC-html40/present/graphics.html

[9] Koch, Peter Paul. The dangers of browser detects http://www.quirksmode.org/blog/archives/2006/08/the_dangers_of.html

[10] jQuery http://www.jQuery.com/

[11] Dojo http://www.dojotoolkit.org/

[12] Prototype http://www.prototypejs.org/

[13] jQuery. Browser Compatibility http://docs.jquery.com/Browser_Compatibility

[14] Wikipedia. Mobile HTML Transcoders http://en.wikipedia.org/wiki/Mobile_browser - Mobile_HTML_transcoders

[15] S. Harper, S. Bechofer, D. Lunn. SADIe: Transcoding based on CSS. In *ASSETS'06, October 22–25, 2006, Portland, Oregon, USA.*

[16] Souders, Steve. http://www.stevesouders.com/

[17] Souders, Steve. High Performance Web Sites http://stevesouders.com/hpws/rules.php

[18] Souders, Steve. High Performance Web Sites, Part 2 http://www.stevesouders.com/blog/2008/04/30/high-performance-web-sites-part-2/

[19] Cuzillion. Help and about http://stevesouders.com/cuzillion/help.php

[20] Thatcher, Jim. Web Accessibility – Section 508 http://jimthatcher.com/webcourse1.htm

[21] Google Analytics. http://www.google.com/analytics/

[22] Souders, Steve. High Performance Websites, Rule 6, Sample 5. http://stevesouders.com/hpws/js-blocking.php

[23] Apache Software. Jakarta Commons HttpClient http://hc.apache.org/httpclient-3.x/

[24] JBoss. JBoss Drools http://www.jboss.org/drools/

[25] Cerner Corporation. Cerner Demontrations https://www.cernerdemos.com