Monarch: A Reimagined Browser for the Modern Web

Daniel Whitcomb

Abstract—Web browsers have become fast and flexible enough to allow web applications to be viable competition to native applications. Now that the web as a platform has become formidable, it has changed the types of web applications being produced, and the ways in which native applications are being built. This observation leads me to propose the concept of the App Web - a category of the world wide web which increases productivity - and its place in the current application experience. Finally, I present Monarch, a system designed to merge the advantages of native and web applications, improving experiences for both the developer and the user.

1 Introduction

The world wide web is one of the most pervasive technologies in the modern day. A large portion of the software being developed today is based on web services. It is now the standard to produce a mobile, web, and desktop version for every internet-based application. This can be a costly endeavor for many companies, and due to the recent advances in the speed and abilities of web technologies, it is common to forgo a desktop application as the web browser can serve as a sufficient platform. It is difficult to argue, however, that the web browser offers a more productive and integrated user experience than a native application. By focusing on web applications, vendors are sacrificing the substantially better desktop environment: a separated application icon, menu system, and access to native APIs. This is most important for vendors which are producing site web applications - a type of web site that enable its users to be more productive on a certain task. Web applications represent a specific subset of website, which we call the App Web. This new category requires the usability of a native application, but by nature, is usually delivered via a web browser.

We will discuss the specifics of App Web sites, and their place within a user's workflow. Then we will understand the differences between native and web applications, and propose the goals for an ideal platform which would have the advantages of both systems. Finally, we introduce Monarch, an web application browser and launching platform. Monarch aims to meet the proposed goals of the ideal platform and bring together web and native applications.

2 The App Web

A mix of new W3C APIs and a boom in JavaScript-centric web page development in the last 7 years lead to a shift in how web sites are designed, and allowed the web application to be an increasingly serious paradigm. The App Web is the category of web sites that can be defined as a web application.

Technically, *web application* refers to a site which uses logic to provide dynamic functionality. This logic may be executed on the server, client, or both. These applications can also sometimes save user's data, handle interactions between different users, and process data, etc. Under this definition, almost all modern

- *Daniel Whitcomb is an undergraduate student at Dartmouth College in the Computer Science Department. This work is the final product of his undergraduate thesis.*
- *The author was advised by Charles C. Palmer.*

web sites could technically be considered web applications.

In this work, I will use a narrower definition because the traditional characteristics of a web application, and the technologies that run most sites have become practically ubitquitous across the web. Rather than a technically-driven definition, we will utilize a content-driven defintion. A web application is a site which is function-driven; the site is associated with verbs like send, create, play, and manage. Without the web application, the user would lose some amount of productivity, or the utlitization of some function. Examples under this definition include: Google Photos, Trello, or WolframAlpha. These differ from content-driven sites which have the goal of providing content to the user. Users generally don't visit these sites with the goal of a function, but rather to experience some kind of content. Sites like Buzzfeed, The New York Times, or Facebook fall under this definition.

To be fair, there are sites with features that allow them to fall under both definitions. For our purposes, though, a strict definition is not required. In further sections, this definition will serve as a guide on how to approach the place these applications have in a user's workflow. The App Web is functionally different from the content-based web, and the purpose of this work is to argue and show that the differences should be accounted for by the methods with which they are accessed. The goal of the App Web should be to become so tightly integrated with the native operating system that users will see no discernable difference between native applications and web-based applications.

3 Native vs Web

The web browser, and its accompanying rendering technologies, has become an increasingly powerful platform. This allows vendors to create increasingly versatile and useful products. The abilities of web technologies have narrowed the gap between itself and the native platform, making soley web-based applications a viable option for vendors. Companies creating products which fit into the App Web must weigh the differences between the two platforms, as their products might fit well with both.

For the remainder of the paper, we will use term *native* when refering to an application that is running within the user's operating system - and uses the operating system's local APIs for graphics, network, etc. The terms *web technologies* or *web-based* will refer to applications run by classical browser technologies like HTML, CSS, and JavaScript.

Native and web applications, while their differences are shrinking, still have advantages and disadvantages. These differences continue to determine the types of applications that are developed, and which users choose to use. The type of application we are considering will by default have a server-side system accompanying it. The differences between back-end systems will not be discussed here because they generally don't affect whether a native or web application is built.

3.1 Development

The major advantage of web technologies is they provide a system agnostic medium for building and running applications. The problems with porting between platforms and operating systems are delegated to browser engineers, while the web is a standardized platform and developers can trust - to a high degree - that their applications will run the same regardless of operating system or browser. This makes it easy to see why developers would continue to choose to build web-based applications, rather than native applications. Web-based apps can have a unified codebase. This prevents updating and porting issues, which can more easily occur across multiple codebases, and keeps interface and experience consistent across all users.

Development speed is also a key difference between the two platforms. The web development community has produced many types of libraries that allow teams to increase development speed. JavaScript frameworks - a relatively new paradigm - have pushed the prefered development model toward a Model-

View-Controller pattern. These libraries generally automate much of the boilerplate code for developers, allowing them to focus on interaction and business logic. Other web libraries similarly streamline page styling, networking, and document manipulation each making a web developer's job significantly easier.

Native application development speed can be dependent on which platforms are being supported. Development speed will decrease as the number of supported platforms increase because each will generally require different systems for building user interfaces.

3.2 Features

Web technologies have a significant advantage over native apps in ease of development and depolyment, but their abilities are highly limited in comparison to native applications. Native applications generally have access to every feature and control the operating system has to offer.

Web processes are highly cordoned off from the operating system. There are many APIs the browser implements that allow web pages secure native access, but support is severly limited compared to that of native applications. This limits the capabilities of web applications when interacting with the user's system, like accessing the file system or taking advantage of OS user interface components. The reduction of native features limits the capabilities a web-based application. W3C has been working to increase the access web developers have to native systems; recent APIs like fullscreen access [1], messaging between pages [2], and socket networking [3] show that there is a commitment to increasing web features, but many major browsers have not yet implemented the numerous API specifications. Until an API has majority support, it will often go unused in production codebases due to browser compatibility issues.

3.3 Performance

Although still lacking in feature richness, web technologies - which once had serious speed disadvantages over native apps - have seen recent improvements in JavaScript speeds as well as improved web APIs to support more intensive work. Substantial photo and video editing, CAD modeling, or game rendering were once too intensive for client-side processing, but new standards like WebGL [4], which provides access a high-performance graphics library, have allowed for applications of web technologies to be realized. JavaScript performance has also been greatly improved by new engines like Chrome V8 [5] produced Google. These advances have removed speed as an inhibitor to the developer and user.

3.4 User Experience

Based on this, the primary advantage of building a native application over a web-based one is feature richness and a stronger coupling with the host operating system. These advantages lead to a significant increase in the quality of user experience. Native applications can take direct control over system menus, contextual menus, and notifications. They have a permanent presence on the user's screen and have an icon either in the Window's task bar, or OS X's dock. They can act as their own entities, without having to rely on a a separate medium - i.e. a web browser. Native applications also generally don't have to make network requests for new interface content, allowing them to have a tighter and more responsive feel. This can be taken into account in web-based systems by building single-page applications, which load all of their interface upon loading.

3.5 Summary

Each platform has significant advantages and disadvantages. Web applications are best in terms of development speed, code simplicity, and near universal operating system compatibility. They do not require installation or updating from the user, and are easily accessed on any computer with a browser.

Native applications lead in terms of operating system integration, allowing for a more fluid user experience and workflow than web browser tabs can provide. This advantage along

with reduced loading times can be a significant enough reason to choose a native implementation over a web one. The success of a product ultimately depends on people wanting to use it, making user experience paramount.

In 2011, Mokkonen and Taivalsaari described the race between native and web applications as the "battle of the decade"[6]. They discussed the potential outcomes of each winning said battle. While the battle is not yet over, the question I ask is why there needs to be a battle?

4 Platform Goals

We have discussed that the differences between web and native applications have become minimal. A platform which combined the ease, speed, and portabily enjoyed by web-based apps, with the integration of a native application, would resolve the primary disadvantages of the two types. In this section I propose such a system.

An App Web site would benefit the most from a combined native-experience web-based system. These sites would be able to produce an improved native-like interface, while keeping their web delivery scheme.

4.1 Requirements

The advantages and disadvantages of both application schemes have been discussed. The combination platform would have the following features and abilities:

- The platform would support applications written in web-based technologies like HTML, CSS, and JavaScript to be universally compatible across opereating systems.
- The platform would provide secure access to operating system features like the taskbar/dock, menu control, and OS interface components.
- Applications on the platform would never need to be installed or updated, merely navigated to.

4.2 Design

The applications will support web technologies, so it follows from a infrastructue perspective that they continued to be served as such and viewable from within a web browser as well as this new platform. Therefore, web browsers - a platform already optimized for processing the web - would be the most appropriate to meet these requirements, especially because they are already such a heavily used class of application. The web browser would be able to adapt its presentation of a page based on whether it is a web app or is content focused. Web applications would be opened as fully-featured native applications - rendered by the browser - while content-based sites could retain their current tab-bar model.

The web launcher - a combination browser and application launcher - would make the web the source of all content and applications. Rather than treating all web content equally, the web launcher would allow the user to access all content in the best possible manner.

5 Existing Systems

5.1 Adaptable Browsing

Henricksen and Indulska [7] discussed the idea of adapting the browser to contextual changes. They advocated for browsers to be able to adapt their interfaces to the sites they display. By using data like a user's browsing history, experience level, language setting and display size, it would produce a customized experience for the user. They focused primarily on the browser adapting to system data rather than page content. In 2003, Tenebaum and Caballero [8] filed a patent for a "Contextually Adaptive Web Browser". The opening of the patent's abstract reads:

> *A web browser's layout, available features and tools are adapted to the instantaneous environment, without the use of downloadable, up-loadable or resident programs, plug-ins or agents.*

The patent describes adding buttons and other interface elements based on the contents of the page, and gives examples on how this might

be done on Internet Explorer or Netscape. The patent is cited by 111 other patents pertaining to browser interfaces, but there seems to be no indication that this patent has prevented any companies from producing products that might fall under its description.

The concept of an adaptable browser has been in circulation for much of the web's existnece. Unfortunately, most of the major browsers have largely not built this type of functionality into their products. One example of a minor addition of adaptability is in the Android version of Google Chrome. Developers can add an HTML tag to their site that sets the color of the toolbar [9].

Browsers do not tend to have a built-in content adapatation systems. Vendors have generally chosen to focus on making their products fast and secure. For example, many browsers have built-in systems fo phishing and malware detection [10] [11] [12], which notify the user if they are potentially navigating to a potentially malicious site. Most adaptive browsing features browsers have built in are generally not meant to increase the quality of the experience.

5.2 Native-like Interfaces

While adaptive features have not been a topic of interest for browser vendors and web developers, creating a native-like experience has been a major goal of both parties. Mozilla was an early investigator into this user-experience concept. They developed a language called XUL (XML User Interface Language) [13], which is a simple way of defining relationships between interface elements, especially toolbars, menus, and navigation buttons. At one point, many of Mozilla's products were built with XUL. Mozilla also included a feature within Firefox that allowed sites to remotely load XUL documents [14], allowing developers to quickly build navigational systems for their sites that matched the interface of the host operating system. Remote XUL was eventually disabled in Gecko 2.0 due to major security concerns.

Two projects out of Mozilla Labs also pushed the concept forward. Prism [15] was a Firefox extension which allowed web sites to be split out of the browser and be rendered in a separate window. The site would accompanied by a desktop icon in the taskbar/dock as well. The basis of the extension was XULRunner [16] - a program that compiles and renders XUL. Prism supported customized context menus, and a printing feature as well. Prism's major disadvantage is that developers also had the option to provide additional app features by creating app bundles. These bundles had to be downloaded and installed separately from the site, and the site could then load the bundle with a custom HTML tag. Prism never reached a 1.0 release, the project was renamed WebRunner [17], and Mozilla shutdown the project.

Google produced a concept similar to Prism: Chrome Apps [18]. Chrome Apps are a type of extension that install a small app bundle on the user's machine, and are built using web technologies. The bundles are either held locally or can be hosted by the publisher. Local installation allows the apps to be used in offline-mode, and they are also able to autoupdate. Hosted applications still require the installation of a small file that provides details about the app.

Chrome has expanded their reach by producing the Chromium Embedded Framework (CEF) [19]. The package modularizes the rendering systems of Chromium Project - the open-source project which Chrome originates from - and allows it be injected into other applications to render and display web pages. CEF makes it much simple for developers to integrate web technologies into native applications. Data can be ferried between the CEF data structures and native code. CEF allows a project with an already existing web application to quickly build a native app and reuse much of their existing code. Companies like Evernote [20] and Spotify [21] each have substantial web applications, but also publish native applications which use CEF.

The CEF API allows for extensive fine-grained control over the rendering system, making it, by nature, more difficult to work into products quickly. Electron [22] is a product built by GitHub which abstracts much of the complexity out of CEF, allowing development speed to be greatly increased.

Another Mozilla Labs project called

Chromeless [17] was a combination of XUL and Prism. Chromeless is a platform that allowed native applications to be built out of web technologies, with the intial concept being to use web technologies to build a native browser application. The system is very similar to CEF and Electron, but never gained traction the other two did. The source for Chromeless is available on GitHub, but is listed as an archive project on the Mozilla Wiki. A project similar to Chromeless was developed by Sun Microsystems in 2008 called Lively Kernel [23]. It is a JavaScript framework that gave pages a desktop-like environment for interaction. The environment had all the features of a desktop, as well as development capabilities. Rather than make the web seem native, Lively Kernel made a native environment within the web.

5.3 Extending Web Features

Web technoliges have generally been limited in their access to the operating system primarily for security reasons. Remote JavaScript is the source of major security concerns [24] and much of the HTML5 standard defines rules for browsers which aim to restrict the privleges of untrusted code [25]. A hestitancy has existed in regards to extending the privleges of JavaScript because of the massive security risk it would entail. Even so, W3C has been pushing forward in defining new web APIs to provide secure access to low level functionality. Google and Mozilla also experiment with custom web APIs that are not meant for production use, but are intended as a proof-of-concept.

In order to allow for JavaScript to be given lower level control over the operating system, the most important task is to mitigate potential security risks. There is a history of proposed methods to do this. A group at Microsoft proposed a *device-local service* [26] for mobile phones. A reported 52% of developers [27] in 2013 were using HTML5 technology within mobile applications. Site JavaScript can use WebSockets [3] to communicate with the *device-local service* in order to access the native layer of the mobile device with a platform agnostic protocol. While they proposed the method specifically for mobile devices, there is no reason this could not also be implemented on a desktop. This system allows native-level APIs to be quickly developed and maintained from a standardized location. The authors did call the security of the system into question. Though they implemented a handshake-like protocol for the WebSockets they do concede that using sockets in this manner is still poking a hole through the render process's sandbox. It is worth mentioning then that Chrome and other browsers already use similar platform-based systems to implement the existing web APIs that involves using IPC to communicate with render processes, also degrading the sandbox.

Another Microsoft research group proposed a system called Embassies [28], which is meant to remedy the problem of puncturing holes in the render process by completely removing the render process from the local platform. They propose a *pico-datacenter*: a native container system that can only communicate with the outside world via IP. It allows any stack to be run on the user's machine, while maintaining strong isolation thanks to its container model. This is a huge advantage for developer freedom, and singular code-bases, but requi In this case, developers would have to send the tech stack along with their application, which can include a rendering platform for their product. Embassies also allows access to native view tools, allowing applications to use the operating system's interface elements.

6 Monarch

Monarch - for the butterfly - is a prototype web launcher, and a novel solution to the native v. web "battle". Monarch is a hybrid web browser and application launcher, allowing users to seemlessly open web sites as native applications on Mac OS X. Users do not move through an installation workflow, and apps open as soon as they are navigated to. Monarch also contains a simple HTML API that allows developers to define menu structue and execution actions for OS X's menu bar.

6.1 Design

Monarch is meant to bring the App Web closer to a native experience, without sacrificing the accessiblity and speed the web enjoys. Monarch is a fork of the Chromium web browser [29], the open-source project that is the primary source of Google's Chrome web browser. It is backed by the rendering engine Blink [30], as well as Google's JavaScript engine, Chrome V8 [5]. The rendering engine is responsible for loading web pages, parsing their HTML, and calculating how their appearance on the screen. The JavaScript engine interprets a site's JavaScript code and works with the rendering engine to enact changes to the Document Object Model (DOM), a data structure that manages the state of the page. Monarch's feature set was built on top of the Chromium and Blink source.

Chromium was chosen as the base project beacuse it is a well known open-source project with extensive documentation. Chromium also has an OS X desktop app extension system [18] that provides much of the basic infrastructure that Monarch is built upon.

Monarch is a browser and an application launcher. It contains all the normal features of Chromium - a typical browser - but also the advanced features of Monarch. By adding the native application features to an existing browser, it makes the browser the central location for all their applications.

Monarch uses two pieces of consistent vocabulary. The first is *app mode*: a page in *app mode* is opened as a native application, and differentiated from the UI of the regular browser. The second is *Monarch Dynamic Application* or MDA. An MDA is the name of the system which opens, controls, updates, and closes a web page in app mode. Each individual site is considered its own MDA.

6.1.1 Navigation

There are two primary methods of opening an MDA. The user may want to either open a URL in an MDA directly, or open an existing tab in an MDA. In order to open an MDA directly from a URL, a user simply types the desired URL into Monarch's omnibox (search/address bar) and an option in the suggestions dropdown menu will allow the URL to be opened directly as an MDA. By selecting the menu item, the MDA is opened immediately.

The second method concerns already opened web sites. There are three actions possible that will convert an existing site into an MDA. Monarch's *View* menu in the menu bar always has a *Enter App Mode* item. Selecting this item will close the current tab and open its URL in an MDA. Users can also select a similar option by opening the context menu by right-clicking on the page. The hotkey for this action is `Cmd + Shift + A`. When opening an MDA from an existing tab, the state of the page is not conserved, though this is a goal of a future implemenation.

6.1.2 Native App

The native application is designed to seem like a standalone application bundle. The main app window is a regular rendering of the web page, with the normal OS X window bar at the top. No browser interface items are present, and the window gives no indication it has a relation to the browser.

The window is accompanied by an icon appearing on the OS X dock. If Monarch has a cached favicon for the MDA's URL, then the favicon will be resized and used as the dock icon. If Monarch does not have a cached icon for the application, a full resolution custom icon made specifcally for MDAs is used.

Finally, the developer has the option to define the MDA's menu bar structure. The app window has a customized menu as well, with four default menus - File, Edit, View, and Window - as well as a menu for the title of the MDA with its customary *About* and *Quit* options. Developers can also define as many custom items as they want. The HTML API allows developers to connect JavaScript expressions to these menu items, which allows users to interact with the page via an operating system interface.

MDAs allow developers to produce an almost perfect native-like experience without requiring a different codebase or packaging. MDAs are able to recreate the experience of an

application made with Electron without requiring any download or installation.

6.2 API

Monarch's additional HTML API is its major advantage over existing systems. It gives developers the ability to keep their web application's existing codebase, allowing their application to be Monarch compatible in minutes.

The prototype API currently allows developers to create menu structures for their applications which get injected into OS X's menu bar when the MDA is in focus. Menu structures are simply trees, which is perfect for encoding into HTML. Only two HTML elements are currently in the API: `mdamenu` and `mdamenuitem`. A `mdamenu` element can contain other `mdamenu` elements or `mdamenuitem` elements, though `mdamenuitem` children will be ignored. A `mdamenuitem` represents a leaf of the menu tree, and is a selectable action, while a nested `mdamenu` represents a child menu that contains a subset of menu items.

Each element has a set of attributes that help define their appearance in the menu bar. A `mdamenu` has two useful attributes that can be assigned. The first is the `title` which defines what will be displayed as the menu name. The root menu does not need a title and it will not be displayed. The second attribute is to denote which menu at depth 1 of the tree corresponds to the MDA's name. On all OS X menu bars, the application's name is the title of the first menu. An MDA menu does not require this menu to be defined in the tree structure, but if the developer does want to define it, they can add the `app` attribute to a `mdamenu` to set the application's title.

There are three attributes for `mdamenuitem` elements. Menu items also have a `title` attribute, which is the displayed name of the menu option. Most importantly, they support an `action` attribute, which can be set to any valid JavaScript expression within the context of the site's main page. From these actions, developers can access functions defined within the JavaScript environment of the MDA in order to react to the menu item being selected. The third attribute allows the item to be disabled to prevent user interaction depending on state. By adding the `disabled` attribute to a menu item, it will gray out the menu item.

See the *Resources* section for information about complete documentation.

6.3 Implementation

Monarch's source is forked from Chromium, so it inherits Chromium's stability and reliabilty. Chromium includes the Chrome Apps feature, which allows vendors to create specialized application packages and users can download from the Chrome Web Store to run as faux-native applications through Chromium. Chromium installs a small *app-shim* bundle which is configured as a valid OS X application bundle. These bundles are very small and act as a way of notifying Chromium that the user is trying to open the application it corresponds to. The app-shim bundle then cedes control back to Chromium, which is the process running the application. Chromium opens a new window and renders the web page inside of it. Chromium then replaces it's normal menu bar with the app's menu bar. Doing this gives the illussion that the application is a separate entity.

The major problem with Chrome Apps is the user must find the desired application - if the vendor has decided to release it - and download it themselves. Many of these apps simply link back to the vendor's already responsively designed web application, making it just another step for users.

Monarch injects web page information into a template Chrome App, which it progamatically installs when the user wants to open an MDA. During the application's lifetime, the bundle is installed as an internal extension which remains invisible to the user. When the user closes the application, Monarch uninstalls the app bundle and cleans up the files. A user may have open as many app bundles as they want, even duplicate source sites.

If the web page has defined a MDA menu structure for itself, the renderer will notify the browser when the menu structure is wholly parsed and the corresponding app's menu is

updated dynamically. This also means that any changes to the menu elements in the DOM will also be propogated to the app's menu bar.

7 Discussion

Monarch is meant to be a step forward in merging the native and web experiences both in the context of the user, and the context of the developer. Monarch allows developers to use the flexible web technology ecosystem to easily build an application that is easily deployed on the web which, with Monarch, also serves as a native application.

7.1 User Interface

An MDA's interface is nearly identical to one which could be produced with Electron. Each app has its own window, dock icon, and menu bar. By separating web applications from the browser experience, it allows users to easily locate and more efficiently move between their pivotal apps. It also increases the user's sense of the app's permanence, making it feel more important than just another browser tab. A user with Monarch essentiallly eliminates the need for any Electron-run application, thus reducing the number of installed applications on the user's machine.

Monarch allows the user interfaces of web applications to become more standardized by adopting OS X's use patterns. The user is able to better predict where menu items may be, allowing them to learn how to use the web application quicker. This can be especially useful for inexperienced users like children or eldery.

7.2 Security

Security is always the most important aspect of any web-based platform. Browsers have been snuffed out of existence by public reactions to security flaws, and almost all new web APIs make security a top priority. The security in Monarch is no different story. Since Monarch is based off of Chromium, it already has Chromium's security rating behind it.

The changes to the render process involves support two additional HTML elements, which includes bindings to the JavaScript engine as well. These elements are defined and follow the same patterns as other existing HTML elements. When the renderer recognizes these elements have changed, it sends a message to the browser process to update the corresponding MDA's menu, and passes the menu structure with it. The renderer builds simple string and boolean structs based on the HTML structure, and passes it over IPC. There are multiple other instances where similar processing takes place, and poses no additional security risk.

The menu data that is sent to the browser process gets converted into Objective-C objects to be displayed in the menu bar. This system is also just as secure as Chromium. The menu action's send a string - sent to the browser process by the renderer - back to the renderer to be run as JavaScript, but this poses no more security risk than a user opening the development tools and using the JavaScript console.

7.3 Current Deficits

The current drawbacks to Monarch are few and can be easily accounted for in future releases. MDA's dock icons are the opened site's enlarged favicons, given that it is available in the browser's cache. This means that MDA app icons are either the default icon, or a horribly pixelated version of the favicon. This could be solved by extending the HTML API to include the desired dock icon as a resource, then dynamically updating the icon after the page is parsed.

Currently, the full extent of the OS X menu API is not supported within Monarch's HTML API. Some things that are not implemented include right-justified keyboard shortcuts, menu separators, and the Help search bar. These features were not considered vital for the initial prototype, but could easily be added future releases.

Finally, when using OS X's Exposé feature. Exposé allows users to view windows grouped by application and quickly switch between them. Since MDA windows are being created and rendered by the Monarch process, Exposé groups all MDAs with the Monarch windows.

This could be fixed by moving the window creation system to the app-shim process, and connecting to the renderer process to access its view. MDAs would then be separated from one another within Exposé, allowing faster application switching. This feature would mean a substantial refactoring of many pre-existing services in Monarch and was not possible within the scope of this project, but with proper resources could be done in the future.

7.4 Future Work

There are some features that do not warrant an explicit platform deficit, but would still be useful in a production version. One is an addition to the HTML API which would allow the browser to detect that the site should be opened as an MDA if available, and define specific settings for how to open it. This would allow Monarch to be more aware of the type of page it is displaying and more appropriately tailor the experience.

The inclusion of OS specific interface components within the MDA's window or menu bar would also be advantageous in making integrated experiences. The HTML API could be extended to allow the render window's initial dimensions to be designated by the developer. The window's title bar could also be customized by allowing developers to change the window title, or hide the bar all together.

8 Conclusion

I presented and discussed the idea of the App Web - a category of web sites which are meant to allow users to perform some type of action or make a task easier - and contrasted it with the content web. The app web ties into the debate over whether to build native or web applications. Monarch is presented as a solution to this debate, and a compromise between the best of both platforms. Monarch allows for a native-like experience, while allowing for the spontaneity and ease of the web application.

Monarch's simple HTML API allows web applications to easily support use in as an MDA. While it is an addition to the already bloated web API, it is the most straight forward path to support currently built applications. The API has the possibility for many simple additions that would greatly increase the ability of the developer to control their site's MDA, as well as their site's presence as an application, rather than a page.

Monarch is a meant to be a proof-of-concept for a grander vision. The use and content of the web has fundamentally changed in its short life. The next question then is how best to adapt to this change. Monarch is meant to be a glimpse at what the web could become: not just a series of sites and pages, but a complete, and immediately accesible application suite.

9 Resources

- Monarch is available for download at http://cs.dartmouth.edu/~drw.
- The source code can be accessed at https://github.com/danrwhitcomb/Monarch.

References

[1] A. van Kesteren and T. Çelik, Eds., *Fullscreen api*, Apr. 18, 2014.

[2] I. Hickson, Ed., *Html5 web messaging*, Nov. 18, 2010.

[3] I. Hickson, Ed., *The websocket api*, Apr. 19, 2011.

[4] K. Group, *Webgl specification*, ed. by D. Jackson, The Khronos Group, Oct. 27, 2014.

[5] Google. (). Chrome v8, [Online]. Available: https://developers.google.com/v8/ (visited on 04/22/2016).

[6] T. Mikkonen and A. Taivalsaari, "Apps vs. open web: The battle of the decade", in *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development*, 2011, pp. 22–26.

[7] K. Henricksen and J. Indulska, "Adapting the web interface: An adaptive web browser", in *Australian Computer Science Communications*, IEEE Computer Society, vol. 23, 2001, pp. 21–28.

[8] S. Tenenbaum and M. Caballero. (May 2003). Contextually adaptive web browser. US Patent App. 10/116,763, [Online]. Available: https://www.google.com/patents/US20030080995.

[9] Google. (). Support for theme-color in chrome 39 for android, [Online]. Available: https://developers.google.com/web/updates/2014/11/Support-for-theme-color-in-Chrome-39-for-Android?hl=en (visited on 04/22/2016).

[10] I. Fette. (Nov. 14, 2008). Understanding phishing and malware protection in google chrome, [Online]. Available: http://blog.chromium.org/2008/11/understanding-phishing-and-malware.html (visited on 04/24/2016).

[11] Mozilla. (). How does built-in phishing and malware protection work?, [Online]. Available: https://support.mozilla.org/en-US/kb/how-does-phishing-and-malware-protection-work.

[12] M. E. Team. (May 11, 2015). Microsoft edge: Building a safer browser, [Online]. Available: https://blogs.windows.com/msedgedev/2015/05/11/microsoft-edge-building-a-safer-browser/.

[13] (Apr. 14, 2014). Xul, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL (visited on 04/22/2016).

[14] Mozilla. (Nov. 21, 2013). Using remote xul, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Using%5C_Remove%5C_XUL (visited on 04/24/2016).

[15] (Apr. 14, 2014). Prism, [Online]. Available: https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Prism (visited on 04/21/2016).

[16] ——, (). Xulrunner, [Online]. Available: https://developer.mozilla.org/en-US/docs/Archive/Mozilla/XULRunner.

[17] (May 26, 2014). Chromeless, [Online]. Available: https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Chromeless (visited on 04/21/2016).

[18] M. Mahemoff. (Sep. 2010). Extensions and apps in the chrome web store - google chrome, [Online]. Available: https://developer.chrome.com/webstore/apps%5C_vs%5C_extensions (visited on 04/21/2016).

[19] (Apr. 18, 2016). Chromium embedded framework, [Online]. Available: https://bitbucket.org/chromiumembedded/cef (visited on 04/21/2016).

[20] Evernote. (). Evernote: Evernote: The note-taking space for your life's work, [Online]. Available: https://evernote.com.

[21] Spotify. (). Spotify: Music for everyone, [Online]. Available: https://spotify.com.

[22] (). Electron, [Online]. Available: http://electron.atom.io/ (visited on 04/21/2016).

[23] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, "Web browser as an application platform: The lively kernel experience", Mountain View, CA, USA, Tech. Rep., 2008.

[24] N. Bielova, "Survey on javascript security policies and their enforcement mechanisms in a web browser", *The Journal of Logic and Algebraic Programming*, vol. 82, no. 8, pp. 243–262, 2013, Automated Specification and Verification of Web Systems, ISSN: 1567-8326.

[25] W. W. W. Consortium, *Html5*, ed. by H. et al., Oct. 28, 2014.

[26] A. Puder, N. Tillmann, and M. Moskal, "Exposing native device apis to web apps", in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft 2014, Hyderabad, India: ACM, 2014, pp. 18–26, ISBN: 978-1-4503-2878-4.

[27] (Jul. 2013). Developer economics q3 2013: State of the developer nation, [Online]. Available: http://www.visionmobile.com/product/developer-economics-q3-2013-state-of-the-developer-nation/.

[28] J. Howell, B. Parno, and J. R. Douceur, "Embassies: Radically refactoring the web", in *Proceedings of the USENIX Symposium on Networked Systems Design*

and Implementation (NSDI), USENIX, Apr. 2013.

[29] (). Chromium, [Online]. Available: https://www.chromium.org/Home.

[30] (). Blink, [Online]. Available: http://www.chromium.org/blink.