PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

UBIQUITOUS CLIENT SIDE CUSTOMIZATION OF WEB APPLICATIONS.

RAÚL MONTES TRONCOSO

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JAIME NAVÓN C.

Santiago de Chile, January 2010

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

UBIQUITOUS CLIENT SIDE CUSTOMIZATION OF WEB APPLICATIONS.

RAÚL MONTES TRONCOSO

Members of the Committee:
JAIME NAVÓN C.
JENS HARDINGS
LUIS GUERRERO
JUAN DE DIOS RIVERA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, January 2010

Para Mariela.

ACKNOWLEDGEMENTS

There are many people I would like to thank because, directly or indirectly, they were helpful or they routed me in the right direction. But I want to give special thanks to some of them who were of huge importance in finishing this thesis.

I thank Mariela for her unconditional love and constant support and encouragement to complete this thesis. I also thank my parents because they understood and supported me in professional and academic decisions sometimes not so easy to understand.

I want to thank Professor Jaime Navón for his guidance, help, trust and also for putting pressure on me to finish this when I needed it. I thank him not only at an academic level but also at a personal level.

Finally, I want to thank my friends: John Owen for our helpful discussions on topics of this thesis, Raúl Acuña for teaching me what I needed about LaTeX and Valeria Gho for her help with the english translation and reviews of the text.

Contents

List of Figures

ABSTRACT

JavaScript has empowered users beyond developers' vision, letting them to customize Web applications to their needs. Widely used tools such as Greasemonkey made this possible by letting users modify applications through the injection of own or third party scripts.

Current efforts however involve either browser extensions or a client side proxy, approaches that are both against the nature of Web applications. The customized application is no longer available in an ubiquitous manner; the user of the application needs to install and configure software on every place he or she wants to access it from. The need for customization of Web applications is clear, but it should be aligned with the ubiquity that the Web gave to its users.

We propose a software architecture to support client side customization in an ubiquitous form, that requires minimal changes in current Web applications, using existing standards to store and fetch the needed information and letting the user work with customized applications from any modern browser and computer. This document describes the proposed architecture, faced challenges, a prototype implementation that shows the technical feasibility of the proposal and possible improvements in the future.

Greasemonkey-like userscripts were successfully used in the prototype, configuring them once and accessing the modified application in different browsers and machines. Applications implementing the proposed architecture will be customizable at the client and universally accessible, keeping this basic principle of the Web. With current working drafts, we could minimize the changes needed in Web applications further, increasing the potential adoption of the proposal.

Keywords: client-side customization, ubiquitous customization, Web applications customization

RESUMEN

JavaScript le ha dado poder a los usuarios más allá de la visión de los desarrolladores, permitiéndoles adaptar aplicaciones Web de acuerdo a sus necesidades. Herramientas ampliamente usadas como Greasemonkey lo hicieron posible permitiéndole a los usuarios modificar las aplicaciones mediante la inyección de scripts propios o de terceras partes.

Sin embargo, los esfuerzos actuales involucran extensiones del browser o un proxy en el cliente, lo que va en contra de la naturaleza de la Web. La aplicación adaptada no está disponible ubícuamente; el usuario necesita instalar y configurar software en cada lugar desde donde quiera acceder a la aplicación. La necesidad de adaptar las aplicaciones Web es clara, pero debería mantener la ubicuidad que la Web le dió a sus usuarios.

Proponemos una arquitectura de software para soportar modificaciones en el cliente de forma ubícua, necesitando cambios mínimos en las aplicaciones, usando estándares existentes para guardar y extraer la información necesaria y permitiéndole al usuario usar aplicaciones modificadas desde cualquier browser moderno y computador. Este documento describe la arquitectura propuesta, los desafíos enfrentados, una implementación prototipo que muestra la factibilidad técnica de la propuesta y posibles mejoras a futuro.

Userscripts como los de Greasemonkey fueron usados exitosamente en el prototipo, configurándolos una vez y accediendo a la aplicación modificada en diferentes browsers y computadores. Implementando esta arquitectura las aplicaciones serán adaptables en el cliente y universalmente accesibles, manteniendo este principio básico de la Web. Con actuales trabajos en borrador, podríamos minimizar aún más los cambios necesarios en aplicaciones Web, aumentando la adopción potencial de nuestra propuesta.

Palabras Claves: customización en el lado del cliente, customización ubícua, customización de aplicaciones Web

Chapter 1. INTRODUCTION

The Web has emerged as an ideal platform for many uses that desktop applications could not satisfy and even for replacing functions that used to belong to desktop applications. The Web supporting technologies have advanced enormously along its short life, giving an idea of how important the Web has become.

When a user enters a Web application, he access the same Web application every other user on the Internet. When an application lives on the Web it is accessible to a huge mass of different users from different countries, cultures and tastes. Therefore, having the exact same application for all those users could not be realistic. This is where the capabilities of the application to be adaptable by its users earns importance. Users should be able to customize its applications and every developer seems to know that – this is why users almost always can see a "Preferences" section on every application, Web or desktop based.

Before entering the terrain of customization and in particular, of client side customization, we present an overview of the evolution of the Web, specially in terms of technologies and how these technologies have helped raise new orientations and uses of the Web. Later, we define and explain the term "customization" in general and in the context of Web applications, in order to review current efforts in customization of Web applications and what they still do not satisfy. We finalize this introduction with our proposal and the results obtained.

1.1. Evolution of Web applications

For better understanding of the topics involved, we think it is important to start with a brief history of the evolution of the Web, from its technologies and capabilities point of view. Our review is based on material from the book "Foundation of Ajax" (Asleson & Schutta, 2005).

The Web begun with just HyperText Markup Language (HTML), a subset of the existent Standard Generalized Markup Language (SGML), and HyperText Transfer Protocol (HTTP). The first one allowed people to write documents that could link to other documents, while the latter was capable of transferring these documents from one server in the world to any other computer connected to Internet. Back then, though, these documents were viewed by many people with just primitive textual browsers, compared to current browsers.

Only static documents were available, for example research papers, information about university classes, contact information or documentation. These documents were published on the Web and then users could download and view them through their browsers. However, expectations and requirements to the Web raised with the popularization of personal computers and elaborated desktop applications like Microsoft Excel and Corel Word-Perfect.

Efforts in both the client (browser) and the server emerged to give users dynamic content. On the server side, the first to come up was Common Gateway Interface (CGI): programs executed on the server that could generate the content served back to the client, making possible to for example, show selling products from a database.

Later, by the hand of Netscape (creators of the first commercial browser, Netscape Navigator) and Sun (creators of the Java programming language), applets were born. Applets were programs downloaded and executed inside the browser which offered dynamic Web applications for users that have Java installed on their computers. Web applications no longer lived only in the server but primarily in the client's browser, which was responsible of executing the applet's code. This was the origin of the "thick clients" concept on the Web, in contrast to a "thin client" just rendering the content generated by the server. Later, JavaScript, a technology created by Netscape, joined the game. JavaScript was initially thought to interact with HTML documents and mutate its contents on the browser (after the documents were loaded). But it quickly became more popular and available on

other browsers with the ratification by the European Computer Manufacturers Association (ECMA) as the ECMAScript specification and the standardization by the World Wide Web Consortium of the Document Object Model (DOM) – a representation of the HTML document accessible programmatically by scripting languages like JavaScript.

The disadvantages of applets as a thick client application and insufficient capabilities of JavaScript demanded the server to evolve too. Java, the programming language of applets, addressed the server side creating the servlets, a more secure and efficient alternative to CGI, and later JavaServer Pages (JSPs). But this did not only happen in the Java world: Active Server Pages (ASPs) from Microsoft and other languages as PHP and ColdFusion were widely used on the server to create dynamic Web applications.

Nevertheless, all those server side technologies delivered a generated content that had to be refreshed entirely whenever a change by the user or by the application was made. A richer user experience like the one offered by applets was needed but without its limitations. Thus, a number of technologies emerged to satisfy this need of richer interactions on the client side.

Flash and dynamic HTML (DHTML) tried to offer a richer experience. The first is a technology created by Macromedia for delivering applications that provide a user experience very close to that of a desktop application with a lower entry barrier than applets from a developer point of view. However, Flash is a proprietary technology and, just like applets, it requires users to install a browser plug-in in order for it to work. DHTML is a combination of HTML, Cascading Style Sheets (CSS), JavaScript and the DOM that lets developers write code that modifies the page on the fly at the browser, without server intervention, but limited to the content delivered by the server.

The advantages of a Web application over a desktop one are clear. Not only are they easy to deploy (because developers just update the application in their servers and immediately the application is updated on all the clients) and easy to install by users (users really do not need to install anything if they already have an internet connection and a browser) but because of their almost null software requirements or even operative system

requirements, Web applications have “universal access”, as Berners-Lee (2006) referred to or, in one word, they are ubiquitous. Users can access the same application whether they are at work, at home, or even in a public internet lounge, needing only a browser and an internet connection.

These important advantages make a Web application solution the first choice when developing a new application, even with the poor responsiveness and user experience they had, caused by this synchronous flow of requests and responses on every user action, refreshing all the page to just update a little portion of it.

Many attempts have been made in order to improve the relatively poor user experience of Web applications. Attempts like Remote Scripting based on Java, others based purely on JavaScript without dependence on the server side, or some hacks within HTML and JavaScript to make asynchronous requests to obtain content and refresh the page. But one technique called Ajax overcame the others and is established now as standard de facto. Ajax is a shorthand for Asynchronous JavaScript + XML and, as Garrett (2005) stated, it is composed of:

- standards-based presentation using XHTML and CSS
- dynamic display and interaction using the Document Object Model
- data interchange and manipulation using XML and XSLT
- asynchronous data retrieval using XMLHttpRequest
- and JavaScript binding everything together

Ajax has made possible to close the gap between Web applications and desktop ones in terms of user experience and interface responsiveness, without the need to install anything and thus keeping itself aligned to the origins of the Web and its universal access, making Web applications an even better choice.

1.2. Customization importance

Before going into why customization in software applications is so important, we must distinguish between two different – but often confused or mixed – terms: customization and personalization. We have taken Nielsen (2009)'s definitions, which groups these two concepts as cases where "user experience is adapted to each individual user's needs" but he also differentiates both in the following way:

Customization: happens when the **user** tells the computer what he or she prefers to see

Personalization: happens when the **computer** modifies its behavior to suit its predictions about the current user's interests

Customization implies a user with preferences about the way he or she wants to see and interact with the application, and actions through which the user expresses his preferences to the application. Thus, in response to those actions, the application reacts and adapts itself to better satisfy the user's taste.

"Everyone is unique" is almost a cliche but this is why an application needs to have customization options. Software customization has been addressed in Human–Computer Interaction research because developers must "write software for millions of users (at design time), while making it work as if it were designed for each individual user (who is known only at use time)" (Fischer, 2001). Even in an application such as a simple text editor, some users might prefer to see eighty characters per line while others a hundred. A dark background color might be comfortable for a user, but other people might find it an outrage.

With the exception of some specific and often already customized developments, an application's audience is very large. Many people end up using the same application but not everyone has the same preferences about how they want the application to be. If the application has no customization options, it is difficult to satisfy a large number of users.

Since software is getting more and more complex in terms of number of features or functionalities, the need for customization options is greater because on each feature or functionality of the application, different users can have different tastes. Thus, in order to meet the requirements of more users, the application would need to be adaptable by a larger quantity of customization options. As Fischer (2001) explains, High–Functionality Applications (HFAs) allow users to control how the HFA behaves through adaptation features such as preferences and customization components and even sometimes by supporting end–user programming with macros and other simple programming languages.

Many different users use the same application, and they need to be able to customize their experience with such applications. That fact is clear enough for developers, who have included this customization options in their applications since long ago and keep doing so, either having a central preferences section, preference options spread across the application, or even both of them.

1.2.1. Customization in Web applications

Web applications are not an exception to general applications in terms of the need for customization by its users. In fact, they might even be a good example of how important it is for an application to be different for each of its users, both in terms of personalization and of customization. As Baresi, Garzotto, and Paolini (2000) explain, "Customization is assuming increasing relevance since complex web applications, such as portals, for example, are offered to a large community of users with different needs and tastes". But this complexity in Web applications is greater every day, specially since the Ajax (Garrett, 2005) revolution because it raised to a new level the interactivity and features possible in Web applications. From the year 2000 until today, Web applications offer many more features that users may need to customize and thus customization is increasingly relevant.

There even is another factor to consider customization so important in Web applications. In most Web applications, users have to register (or sign up) in order to use it – or have access to more functionalities. Thus, the concept of "user account" in Web applications is widespread in a powerful way, as opposed to desktop applications. As a

consequence, although Web applications have only one logic deployment location (one of its huge advantages, from Section 1.1) users expect to have a unique version of the application that they can adapt to their taste; after all, they see a "Welcome, John" message every time they are using the application.

The presence of customization options in Web applications can be easily found in many contexts. For example, taking any social Web application like Facebook, at least one thing must be customizable: e-mail notifications. Some users prefer to receive e-mail notifications for every event that happens within the social network while others simply hate any type of e-mail notification.

Another common example is a bulletin board Web application like phpBB or VBulletin. Besides all the customization options the admin has – affecting all of its users – each user of the application has access to many configurations that change the way they interact with the application. For example, visual changes such as the skin or font size, changes in the application's behavior such as e-mail notifications or show/do not show the user online status and changes in the way users see information, such as date and time format or display order of posts.

The need for customization in the context of Web applications is at least as important as in desktop applications, but Web developers must deal with the fact that there is only one instance of the application every user on the planet has access to. Thus, the possibilities and ways of offering customization options are very different from desktop applications, where every user has its own instance of the application and therefore any modification applied to it does not affect other users in any way.

1.3. Existing approaches to Web application customization

In the case of common desktop software installed on some machine, the code that runs when executing the application lives on the same machine, as well as the application data and the information about how the user has customized the application. Of course, there are variations where the desktop application loads some data from internet or other

computer but the point is that every user running the application has its own version of it in his computer.

In the land of customization, having this own version of the application allows powerful and broad forms of customizations, such as extensions and plug-ins (Birsan, 2005). The widely used Firefox browser, for example, was built from its foundations as an extensible application and because of that, every user has infinite possibilities of customizing it. Users only have to download and install small pieces of software called "extensions" and they can alter, extend or change the application's interface, behavior or functionalities. Since all users have their own instance of the browser installed on their machines, every one of them can install different extensions and adapt the browser to meet their needs, without affecting other users.

In Web applications the scenario is different: the code of the application itself is completely contained on the server (or servers) hosting the application, but when the application "runs", it really runs partly on the server and partly on the client (the browser that loaded the application).

Every time the user executes the Web application – that is, he enters the Universal Resource Locator (URL) where the application lives on his browser – the part of the application that is executed on the browser is loaded from the server and, within this execution, the browser asks the server for some data or other pieces of the software – and thus executing the server part of the application again.

The important thing to note about this is that the same code on the server is run for every user of the application, and a potentially different code of the application is run on every users' browser. Thus, the application could offer customization options both in the part that runs on the client side (the user's browser) and also by identifying the user on the server. At the server, this customization can be done whenever the server delivers data or pieces of software to the client – this happens at the initial loading of the application, where the server can deliver a potentially different part of the application on the client, and also when the client side of the application asks the server side for information. At

the client side it must be taken into account that the part of the application running there is loaded the same way every time the user "starts the application", so any modification done only on the client will be lost next time the user loads the application.

In the following subsections we discuss about the approaches, possibilities and limitations of current efforts to provide customization options (in some cases, unintentionally) to users of Web applications at both, server and client side.

1.3.1. Server side customization

We refer to server side customization in Web applications when the source of information about customizations is the server side of the application. Besides, most of the process of applying these customization is executed also on the server, thus delivering to the browser an already customized client side of the application. There are three important approaches for customizing Web applications that meet these restrictions. The following subsections explore each of them.

1.3.1.1. Application's Preferences section

The most basic form of customization and probably the first one everyone can think of is the equivalent to the most common way of customizing in a desktop application: the preferences section. As discussed before, the server side of a Web application is the same to all of its users. Therefore, to be able to store and retrieve customization information the application needs to identify and distinguish every user uniquely and user accounts cover exactly this requirement.

The user has to register an account on the application and then log in with a username (sometimes an e-mail) and a password. The application loaded for this user may be different to the one loaded for another user if one of them entered the preferences section and changed some of its values. Thus, while one user might see twenty items per page in a shopping application, the other one might see fifty items per page on that same application.

Almost every Web application has a preferences section. Some let the user customize just a couple of values while others have all sorts of options to alter behavior, look&feel

and features. Although we do not propose to eliminate this form of customization, there are clear limitations to it: the only preferences that can be changed are the ones the application's developers have considered for user customization and they have assigned resources for implementing them.

1.3.1.2. Plug-ins and extensions

Another form of customization, also based on desktop applications, is through plug-ins and/or extensions. We mentioned Firefox as one of many desktop examples of this approach, but within Web applications there are many as well. Blogs such as Wordpress, Forums or Bulletin Boards such as phpBB and vBulletin, Content Management Systems (CMS) such as Joomla and Drupal and Project Management Systems such as Trac and Redmine, to name a few: they all have a sophisticated plug-in or extension system.

Generally, any person with the required knowledge can develop a plug-in or extension and put it to everyone's disposition. Thus, this decreases the dependency on the application's developers like seen in the previous approach. Nevertheless, it also has very important limitations. They are not "per-user" oriented because only the Web application's "administrator" (or Webmaster) has the power to install these plug-ins or extensions. Aside from this, although these plug-ins may also inject extra options in the preferences section for configuring them, once they are installed, they alter the behavior, functionalities or look&fell of all Web application's users.

There is no more "per-user application" with this form of customization because it is not oriented to provide a different experience to many different users of the same Web application, but to provide a different base application for many different installations of the software, each one having their own users. Therefore, in practice, it is like having different applications created from one core application and any of them will offer only the previous form of customization to its respective users. For example, phpBB, before mentioned as a bulletin board application, can be customized by its administrator with a variety of plug-ins. Because of this there are many phpBB installations different from each other on

the Internet, delivering a different experience to its respective users. However, these users, as mentioned earlier, can also differentiate their experience from other users belonging to the same installation of the application using the previous form of customization already present in phpBB.

1.3.1.3. Per-user widgets or gadgets

We have said that in a Web environment the same piece of software (on the server) is running for all of its users, but this is not always true. The third approach tries to break this principle making the server side of the application different for every user, or at least it can be seen this way. The application has a – sometimes very complex – infrastructure that allows it to dynamically load and present functionality to its users that does not live in the same place of the application, and the best part is that the decision of wether or not to load some piece of software from another place on the internet depends on each user.

This approach has been popularized with Web applications like Facebook, iGoogle and Netvibes. "Facebook applications" (Gjoka, Sirivianos, Markopoulou, & Yang, 2008) and "widgets" or "gadgets" (Pierce, Fox, Yuan, & Deng, 2008) are different names to refer to these pieces of software that customize the application. Although there are substantial differences between both of them, they share one important thing: users can add these pieces of software developed by third parties – or even the same application's developers – into the primary application. These pieces of software can live and be executed on other servers but communicate to the primary application through an exposed Application Programming Interface (API). Therefore, it can be seen as a huge application where its server component lives and runs in many distributed places and every user can run a different version of the application if they use different pieces across this infrastructure.

In this form of customization, techniques such as HTML iframes, dynamically loading JavaScript code, Ajax proxies or storing the third party's piece of software in a special way in the primary application's servers are typically used to blend these other applications into the primary application, because normal Ajax requests and other techniques usually

present in Web applications cannot load application code from other servers in different domains.

Although there is minor dependency on primary application's developers – after they have built the API and all the components needed for this architecture – and it is possible for any user to have a different application customized to meet his needs, this approach has two drawbacks: the possibilities offered to third parties are limited by the API and resources provided by the primary application's developers; and supporting this architecture needs a huge amount of development effort on the Web application. In cases such as Facebook, the effort did pay off, but it is not the general case for every Web application.

Approaches on the server side are necessary, but they do not complete the picture. Needs like no dependence on the Web application's developers and unique adaptability to every different user are not always met and still have drawbacks showing something else is required.

1.3.2. Client side customization

In contrast to server side customization, we refer to client side customization when the information obtained for these customizations and the changes to the application are triggered and executed in the client side – usually, the browser.

The very existence of client side customization is because what users see about a Web application is the result of the browser's interpretation of the application. Developers can tell the browser how they want the application to look and behave (that is the purpose of Web standards), but in the end, the user will see what the browser will show to him – either controlled by the application, the user or itself – and therefore developers do not have full control of the Web application.

The client side of Web applications is growing, increasing complexity and sophistication, specially since the Ajax introduction mentioned earlier. This has created new opportunities for other forms of customization not possible (or at least not imaginable) in the past, because the empowered client side of the application can have other sources of

information besides its own server. Also the client side is completely capable of altering itself and therefore behave or look completely different.

There are only two sources where the client side of the application can store and load customization information: the browser and other servers. There are no other possibilities since there are no more actors involved. However, these two sources are very wide.

At the browser, cookies and special plug-ins or extensions offer storing and loading information capabilities. Cookies are small text strings sent by a Web server to a browser – or set on the client using JavaScript, for example. This text is stored at the browser, it is associated with the sender's site and it is sent back to the Web server on each future request to the same website (Millett, Friedman, & Felten, 2001).

The most common use of cookies is to store information about the visiting user, for example, the identification of its session so the server can know that the request is related to all other past and future requests of the same user. In terms of customization cookies are rarely used and when they are, it is for simple and non-critical things such as remembering the last state of a panel (opened/closed) or the preferred language and locale, because of its weak persistency nature. Cookies are easily and frequently deleted (and developers, as previously stated, have no control on this) so in practice, a Web application cannot depend on the persistence of a cookie beyond the scope of a user's session. Also, because they are sent back to the server on every request, they do not offer a good alternative to store big quantities of information without being inefficient (in fact, they have a limit of 4096 bytes (Kristol & Montulli, 1997)). Even if we ignore these limitations, cookies are stored only at the requesting browser. Therefore, when starting the application from any other machine or any other browser on the same machine, all the information stored before will not be available.

Plug-ins like Google Gears offer a more stable storage alternative with high data capacity. Gears is a browser plug-in that adds a local database to the browser in which a Web application can store all kinds of information and even search through it. Customization information could be easily stored and retrieved by the application but, like cookies, it is

only accessible from the browser where they were stored. Therefore a different machine implies different information. In addition, like every other plug-in, it must be installed on a browser to be accessible for a Web application so unlike cookies – part of the HTTP standard – a user can have this possibility in one browser but not in others.

The second source of customization information are other servers. However, this implies cross-domain requests from the application, and that is a restricted subject. Because of many security concerns (and that is why the "same-origin policy" (Zalewski, 2009) exists), there is no direct way to communicate to another domain from within the client side of the Web application. Things like generating an Ajax request to another server or loading another domain's application into an iframe and obtaining some data through JavaScript communication are not possible due to these security restrictions.

There are techniques however that allow getting information from other servers without falling into these restrictions, although they have some limitations. Among them we can mention loading external scripts (sometimes customized through HTTP GET parameters) by dynamically inserting HTML script tags, the use of callback functions, cross-domain requests through Flash or other browser plug-ins or extensions, cross domain request proxies, etc. One prominent solution in this area due to its security, cross browser functionality and the quality offered in communication is Subspace (Jackson & Wang, 2007), although the focus of this and the other mentioned techniques is not customization, but mashups or integration of widgets and similar type of components.

Although storing the information in another server does not compromise the "universal access" of the application as a whole (in contrast to using the browser), it has its downsides as well: unavailability of the external service, risks such as appropriation of personal information and service ceasing to be free, etc.

In this context, the browser and other severs offer only storage and retrieval facilities, but developers should build all the necessary infrastructure in order to have client side customization. We did not find any serious attempt to use, from the client side of the application, these two sources of information to provide client side customization.

The Web application's dependence on the browser interpretation mentioned earlier, opens another chance of transformation and, as a consequence, of potential customization of the application. A Web application can also be changed by changing the browser's behavior when interpreting the client side of the application.

A more aggressive but also successful approach was born from the modification of the browser: to alter the client side of the application to meet customization needs. This approach began with Greasemonkey (Brooks, 2006), a browser extension for Mozilla Firefox that can inject scripts written in JavaScript to the Web application. These scripts, called "userscripts", have a few restrictions and special rules they must obey and a few features aside from normal JavaScript libraries (such as real cross-domain XMLHTTPRequests).

With Greasemonkey, a user can select which userscripts are injected to which Web applications by filtering its URLs. These userscripts are stored locally on the browser and injected each time a Web application is loaded from the server and meets the configured criteria. Because JavaScript has the power to alter the style (CSS) and structure (HTML) of the application, we could say it can apply any customization one can think of to the user interface. Additionally, because JavaScript is indeed the behavior of the client side of the application and can even alter itself, these userscripts can also change the behavior of the application, at least in terms of what happens at the client side.

Greasemonkey was a revelation to many unsatisfied Web application users. Suddenly, they could take a userscript – published in a repository or even if they had the right knowledge, one programmed by themselves – and inject it to the Web application, changing look&feel and functionalities. They could add configurable keyboard shortcuts, a missing button, a summary panel of information, etc.

This Firefox extension is used by nearly 3 million people and over 39 thousands userscripts are published in the principal userscript's repository, userscript.org, showing itself as a successful approach to client side and user driven customization. After the success of Greasemonkey, a Mozilla Firefox exclusive extension, alternatives for other browsers also appeared. "Trixie" (2005) and "Turnabout" (Reify Software, 2008) for Internet Explorer,

"Greasekit" (Kazuyoshi, 2008) for Safari, and even out of the box support for userscripts in Opera (Opera Software, n.d.).

The Greasemonkey approach did not only appear in many other browsers, spreading its use, but was also followed by many other projects. Greasemonkey is very powerful and general purpose, but with the cost of being more difficult to use and too elaborated for many users. Thus, specific purpose extensions appeared – often built upon Greasemonkey userscripts – using similar techniques to alter Web applications. For example, "GTDInbox" (Mitchell, 2009) added "Getting things done" functionalities and "Better Gmail 2" (Trapani, 2009) provided many customization options to Google's Gmail. A user just needs to install these extensions to enjoy the new features without knowing about userscripts or worrying about anything else.

Besides these specific purpose extensions, others also appeared aiming to decrease the programming skills needed for a user to build a userscript. However, as often, ease of use comes along with less power or more narrowed scope. Printmonkey (Baldwin, Rowson, & Coady, 2008) for example, helps users make printing templates that modify the Web page for better printing. Koala (Little et al., 2007), on the other hand, can record user actions on a Web page and play them later, therefore automating processes on the Web. These two functionalities could also be added using userscripts and Greasemonkey, but these particular extensions facilitate user's tasks by avoiding writing/understanding complex userscripts and instead creating these tasks with non-developer centric tools.

Based directly on userscripts, extensions that provide a simple and more intuitive user interface have also been created. "Platypus" (S. R. Turner, n.d.) generates userscripts that change the user interface, adding, removing or changing elements in a "What you see is what you get" (WYSIWYG) mode, so a normal user can create a complex userscript that will be loaded every time he or she visits the page, but in a simple way. Of course, it is very limited compared to the capabilities of a normal userscript, but this is accessible for a normal user.

More powerful than Platypus but harder to learn and use is Chickenfoot (Bolin, Webber, Rha, Wilson, & Miller, 2005). Chickenfoot is another Mozilla Firefox extension that lets the user create complex scripts but without knowing JavaScript (they are not like Greasemonkey userscripts). The user writes the actions in a close to natural language and an interpreter understands and executes these sentences. This extension can do anything Platypus can do and much more but, even with close to natural language, the user will need more knowledge and expertise to take advantage from this software.

It is possible to modify a Web page dramatically through browser extensions, both in terms of appearance and behavior. But this approach has two important drawbacks: the need to install additional software (it is not enough with just a browser) and the storage of the customization information in the same browser. Because in this approach the browser is also the storage location of the necessary information, it is not possible to load the customized version of the Web application in another computer without having to do the whole process again – installing the extension, looking for the scripts, templates or equivalents, loading and configuring them.

Efforts in client side customization have been quite successful and it is widely used because it allowed to overcome the downsides of server side customization – specially the developers dependency. But the elimination of this developers dependency came at the cost of loosing the ubiquity, and a client customized application in one computer or even in one browser, is not accessible in any other browser or computer. Next we propose an architecture that allows client side customization without loosing the ubiquity of the modified application.

1.4. Proposed approach to Web application customization

A revision of what has been currently achieved in the customization of Web applications, reveals that there is still something missing. There have been efforts from developers, who know user customization is important, but users are still not satisfied. Initiatives

like Greasemonkey show us that users want more participation, because nobody but themselves knows how they want the application to be.

Developers, on the other hand, know they cannot adapt the application to meet every user's needs and tastes, because of development effort and also because they do not want to bloat the application for every user just because a few want some particular button.

Approaches such as Greasemonkey help solve this problem, but some developers feel threatened because they loose even more control over what the user sees – specially when concerning ads. Although this threat is real (there are many userscripts that block ads and even specialized Firefox extensions for this purpose), "resistance, as they say, is futile", claimed Google (*GmailGreasemonkey10API*, 2007) referring to users, when it released the JavaScript API in its new version of Gmail. This API facilitates writing and maintaining Greasemonkey scripts, giving direct access to interface elements and to event callbacks, among other things.

The approach we propose is an architecture that does not give full control to developers, thus limiting the customization options offered to users, and does not give users an unrestricted access to change everything either. We propose a joint venture between the original developers, third party developers and the users in order to produce a more satisfying solution to the user, all this keeping one of the basic principles of a Web application: universal access.

1.4.1. Important considerations and restrictions

There are two critical problems to solve if we want to give users customization capabilities: where to store the information needed about the customization and how to apply these customizations to obtain the desired effect on the user interface or behavior.

These problems however, must be solved considering a few restrictions and considerations that guide this work and make it a real contribution. These guidelines are:

(i) Provide the user the ability to use the customized version of the application anywhere he is, from his computer at home, his laptop computer, his office

computer or even in a public internet lounge. The only requirement should be the identification of the user.

(ii) Minimize the development effort needed on the Web application side to support the desired capabilities, because the adoption resistance by developers will increase as the cost increases.

(iii) Do not require any change or additional software installation in a computer to support the architecture proposed, because that would directly imply the lack of ubiquity in the solution. We cannot expect everybody to have a specific browser or to install a specific extension or plug-in but we can assume every computer will have a modern browser – after all, big applications like Gmail, Facebook and Youtube require this to work properly.

(iv) Use only existing technologies, standards and specifications because we want to build a solution applicable in today's technology infrastructure. Otherwise, we would have to wait perhaps five years for a new technology, standard or specification to be available in any modern browser and server.

1.4.2. An architecture for ubiquitous client side customization

The architecture we propose considers three participants. The first one is the Web application to be customized. According to one of our guidelines, the only change required in the application is the inclusion and configuration of a general library that adds support for ubiquitous client side customizations. This library has a customization engine that stores and retrieves customization information and applies the customizations in the client (browser). The library is not specific to any application, therefore it can be included in any application without requiring additional development (although for the server side of the library, platform specific components should be provided).

The second participant is the client or the user's browser. Aside from being a modern browser there are no additional requirements for it (therefore fulfilling the third guideline). The browser, as always, is the responsible of running the client side of the application, including the library installed on the original application. Thanks to this library, the browser

applies the customizations for the user and therefore presents him the customized version of the Web application.

The final participant ties together the other two in terms of the customization process. It is the place from where the library installed in the application will retrieve the customization information through the browser. Customization information cannot be stored in the client computer because of the first guideline. Neither can it be stored in the server of the application because of the second guideline (this would require development effort on the application). Having this information in a centralized service has considerable drawbacks too (explained in Chapter 2). Therefore, in our proposal the third participant is an OpenID service provider.

OpenID (Fitzpatrick, Recordon, Hardt, Bufu, & Hoyt, 2007) is an open and decentralized standard (users can have any number of service providers, change them at anytime and even have their own server to provide this service) originally created for authentication control on different services or applications with the same digital identity. Therefore, users can use the same identification information to sign up and sign in on any application (with OpenID support).

The OpenID specification has an extension called "OpenID Attribute Exchange" (Hardt, Bufu, & Hoyt, 2007) for storing and retrieving any kind of information using the same user identification. With this extension we can provide support for storing and retrieving necessary customization data using technologies and standards available today. Since this information lives in a universally accessible place (an OpenID provider), we fulfill the first and last guidelines.

This is just an overview of the proposed architecture. Section 2.3 of Chapter 2 refers to the architecture in more detail.

1.4.3. Prototype implementation and results

Based on the architecture described above we developed a prototype implementation and we tested it with customizations for an – also developed – example Web application.

The prototype implementation consists of three parts: a basic customization library, an example Web application that includes this library and an OpenID provider prepared to work in this context.

In this prototype implementation of the architecture we loaded customizations (in the form of Greasemonkey-like userscripts) from the developed OpenID provider and applied them in the client side of the application. This was done in different browsers and in different machines obtaining the same results without requiring any software installation. Thus, we provided access to the customized Web application on different browsers and computers transparently for the end-user. In addition, some existing Greasemonkey userscripts were successfully tested on this prototype.

Chapter 2. UBIQUITOUS CLIENT SIDE CUSTOMIZATION OF WEB APPLICATIONS

The following chapter is a paper, submitted for publication in the Journal of Internet Technology.

2.1. Introduction

2.1.1. The browser as the Web application's customization engine

Almost every application has a "Preferences" section, where the user can customize it in several aspects that can go from simple visual details to the actual behavior of the application. The preferences section exists because developers have no way to know exactly how each one of the immense variety of users that could use the application will want to interact with it. These customization options allow the application to adapt to the user specific needs and tastes.

Web Applications are not the exception in the need for customization. They are almost always used by many different users. As Baresi et al. (2000) pointed out, "Customization is assuming increasing relevance since complex web applications, such as portals, for example, are offered to a large community of users with different needs and tastes". Web applications are today even more complex and they offer many more forms of interaction. Supported by technologies such as Ajax (Garrett, 2005), the Web has emerged as a platform that can host applications with interactivity and features close to those one can find only on desktop applications and therefore they have a large customization potential.

Web applications designed to support forums allow customization of the look&feel with templates, change messages' time-zone or show/hide avatars. Popular mail applications such as Gmail let you decide whether to activate keyboard shortcuts, choose the date and time format or save a signature for your e–mails. All these customizations are possible only because the service provider and/or the developers thought you might need to change those settings. If some users want to be able to change the way they visualize part

of the application, they are totally dependent on the developer's interest in making this change, the time to work onto it and, in some cases, even on the service provider agreeing to update the software with the required change.

The users' dependence on the application's developers is partially solved through the plug-in technology. If the application supports plug-ins, a piece of software developed by a third party can be added to extend or change the original application. However, the dependence on the service providers for plug-in installation remains, and since server plug-ins change the application for all users, it does not represent a satisfactory solution to the problem of customization for the individual user. Facebook (Gjoka et al., 2008) and widget aggregators (Pierce et al., 2008) like Netvibes solved this limitation by allowing per-user integration of "applications", "widgets" or "gadgets". This approach however is many times impractical for many applications because it needs software and hardware infrastructure and third party developers to support it.

At the end of 2004, Greasemonkey (Brooks, 2006), a free extension for Mozilla Firefox browser, was born. This extension let users inject JavaScript code to Web pages, in the form of "userscripts", locally on the browser after serving the page. Since JavaScript combined with the DOM API (Document Object Model) can change completely the user interface, this extension in fact created a back door for customization. When open, this door inverts the paradigm of Web server-client interaction from a top-down Web to a bottom-up one, where "the focus shifts to the many ways readers will transform Web pages to suit themselves" (Brooks, 2006).

But this paradigm is not new. It was 2001 when S. Turner used the concept of "Active Browsing" to refer exactly what Greasemonkey does: "Instead of accepting web pages 'as is', active browsers transparently modify, delete and edit web pages according to specific user needs" (S. Turner, 2001). Because the browser receives data from the server and is its responsibility to show it to the user, it has also the power to decide what to show and how.

With Greasemonkey, the user no longer depends on application developers. Even if developers do not change the application or do not put an extra option for customization,

the end-user can customize the Web application to meets his own needs. Although not many end-users know JavaScript to change things by themselves, the universe of people that has the necessary knowledge is still vast – being JavaScript top ten in programming community (TIOBE Software, 2009) – so the possibilities to adapt the application are augmented.

The success of Greasemonkey is indisputable. The extension has been downloaded almost 25 millions times (Lieuallen, Boodman, & Sundström, 2009). There are now over 39 thousands userscripts submitted to the biggest community repository (Andrews, n.d.), and that does not even counts personal userscripts not shared to the community. The key problem solved by Greasemonkey was so real that equivalents in other mayor browsers (with some limitations and differences) appeared like "PithHelmet" (Solomon, 2009) and "Greasekit" (Kazuyoshi, 2008) for Safari, "Turnabout" (Reify Software, 2008) and "Trixie" (2005) for Internet Explorer and bundled support in Opera (Opera Software, n.d.). This new paradigm has been widely used and quickly popularized because it addressed an unsatisfied and important need.

Even with Greasemonkey, end-users are still dependent on knowledgeable people capable of generating the needed userscripts. Although many people are creating and contributing with their userscripts, some of them could require specific changes not available in the community. Several proposals have been presented to allow inexperienced users modify their Web pages. PrintMonkey (Baldwin et al., 2008) can be used to easily modify a page for better printing, Koala (Little et al., 2007), focuses on Web process automation, "Platypus" (S. R. Turner, n.d.) for creating userscripts that make visual changes to Web pages with a graphical interface, Chickenfoot (Bolin et al., 2005) helps in creating scripts expressed in a high-level language and Accessmonkey Framework (Bigham & Ladner, 2007) can inject scripts written in JavaScript into Web applications to improve its accessibility. The last four are currently active projects, showing the importance of client side modification of pages even for people without expert knowledge.

2.1.2. Huge customization possibilities, but without universal access

There are of course situations when a desktop application is preferred to a Web application, but Web applications have a huge and exclusive advantage: the only thing a user needs to use the application is a browser, which is a program that is free and comes bundled with most operating systems. The user can start the application from any computer (even modern mobile devices), without installing anything. This advantage can be summarized in "ubiquity" or "universal access" as Berners-Lee (2006) refers to.

Many people are customizing the Web through scripts that run at the browser in the way we described above. There is however one problem: this goes against the most important advantage of Web applications we mentioned before, the universal availability from every place, and every time when needed. Since the application we want is not the one we get from the server, but the browser altered version, ubiquity is lost. To use the Web application from another computer, we need to install not only a browser extension like Greasemonkey, but also every userscript and its configuration. Furthermore, the browser where our extension can be installed may not be available at that time.

We believe that although it is important that a regular user without expert knowledge can change the look and behavior of a Web page according to his needs, this should not come at the cost of doing it against the nature of Web applications. Consequently, the changes implemented via userscripts or any other way should have the same availability and omnipresence that the Web applications they target.

2.1.3. A new architecture for client side customization

We propose an architecture that supports ubiquitous customization, so modified versions of existing Web applications are accessible anytime and anywhere. The customization lives on the Web but it is loaded and applied at the browser in the same way Greasemonkey does.

We chose not to create a new protocol or specification to support this architecture so Web browsers do not need to be changed. The architecture is based on existing Web

standards (HTTP, HTML, ECMAScript, CSS) and on applications that use these protocols like OpenID (Recordon & Reed, 2006), so it can be implemented in today's Web infrastructure with a minimal impact on existing Web applications. Web applications just need to include and configure a light library to support this architecture.

In this paper we present first an overview of the main issues that need to be considered. We propose then an architecture that takes those issues into account. The implementation of this architecture into a working prototype allows us to show that the solution is not only feasible but also quite simple. Finally, we present a preliminary evaluation of the results and some ideas for further work.

2.2. Main challenges

To create an architecture capable of storing and loading modifications of Web applications that are accessible from everywhere requires solving two main problems:

- Where to store the customization scripts to be applied to the Web application
- How to apply the modifications to the Web application

Furthermore, these problems must be solved under the following restrictions:

(i) Minimal (or none) impact on existing Web applications. Otherwise, the additional development costs involved in building a customization-ready application would discourage people resulting in just a few applications that could be customized.

(ii) It should work on a standard browser. We do not expect this to work on very old browsers, but it should work on any modern browser available today. This is critical so we don't have to wait perhaps for a long time before we could start.

(iii) Use of existing standards and specifications. We do not want to create new standards for storing and loading of Web page modifications. This is to allow the use of existing Web infrastructure and also not adding to the development costs.

In the following subsections we examine the whole range of possible solutions to the two main problems described above.

2.2.1. Storing and retrieving customizations

The information needed to be stored, and loaded later on, depends on the form of the modifications or customizations, but it should contain a list of the Web applications (URIs), the instructions to be applied to each of them to get the desired customization and optional configuration parameters. This information can be stored only in one of the following three places: at the client (i.e. browser), at the server (where the Web page to be modified lives) and at "the cloud" (other servers on the Internet).

The first option (client) is used in Greasemonkey. This extension stores the information on the client machine and requires a small browser intervention: installing the Greasemonkey extension. Unfortunately, this solution therefore violates one of our restrictions. Even if we avoid installing anything at all in the browser (using cookies could be a way, although very limited) the information would not be accessible from another machine and therefore the ubiquity is lost anyway.

If the customization information is stored at the server and it is loaded from the application itself, the customized version of the application will be as ubiquitous as the original one. The problem is that this approach causes a big impact on Web application development. It would need not only database intervention but also changes in both the model and the interface of the application. This is because the information is associated to the user who will have to indicate the modifications that will be applied in the application itself. The biggest drawback, however, is that the modifications would be present only for one particular application, and therefore could not be shared by other Web pages, as occurs when the customization information is stored in the browser.

The important drawbacks associated to the first two possibilities lead us to the last option: other servers on the internet. Although this approach doesn't completely free the Web application's developers from implementing some way of loading the information

from the other server, either from the client (browser) or the Web application's server, the extra code needed is quite small and it will work with a standard browser (doing the loading part only from the application itself). We discarded Web proxies because of privacy and availability concerns, and we also discarded client side proxies because they represent a form of client side alteration that compromises solution's ubiquity too. Storing is less problematic because in the worst case it could be done in this new server directly by the user.

Perhaps the first idea that comes to mind to implement the approach of information on another server would be a centralized external service. In the Mozilla Weave Project for example, a Firefox extension keeps bookmarks, passwords and other personal information on a service centralized by Mozilla for syncing later on other machines. This solution has two big problems: first the eventual downtime of the only service that holds the information and second, that personal information will be on "other people hands". For many people the fact that the owner of the service is also the holder of the information is risky because for example, they could start charging money or simply interrupt the service at anytime. Centralized services can disappear at any time, like almost happened to "tr.im" (*tr.im Resurrected*, n.d.), an URL-shortening service that was very close to shutting down because they could not monetize their service. Even when there are many similar services, they all share a centralized philosophy and if you decide for one of them, you must stick with it or loose all the information.

Fortunately, centralized services are not the only choice. There are also some widely used decentralized services. OpenID (Recordon & Reed, 2006), for example, is a "decentralized standard for user authentication and access control" that let users to have one authentication information (and even personal information) and use it on every site that supports this standard. There is no central service holding all the information. Every user can choose any of the existing OpenID Providers, change it when they want and even have multiple OpenID Providers simultaneously.

We chose the approach of an external service holding the information but in a decentralized way, like OpenID. For this to happen the Web application has to be able to load the information from this external service. The solution is based on OpenID 2.0 (Fitzpatrick et al., 2007) and in an OpenID extension, OpenID Attribute Exchange 1.0 (Hardt et al., 2007). This has the following advantages:

- It's a widely used standard and with Attribute Exchange extension – already implemented in many libraries and OpenID Providers – it can support all the requirements and there is no need to add or modify anything, so no new standards and/or protocol is needed.
- It doesn't need special support by the browser, so the ubiquity of the application is not compromised.
- The development effort is small.
- It has no dependency on a centralized service. Anyone can build his own service using existing open source libraries. Furthermore, there are many OpenID providers that support Attribute Exchange extension, so little development would be required to support storing and loading the customization information.

To minimize development effort, most of the necessary code for OpenID authentication should be moved to the client side. Doing so, developers would only need to include some client and server scripts into their applications to support the customization architecture.

2.2.2. Applying customizations to Web applications

Applying the modifications is not hard. Since JavaScript can insert "script tags" and evaluate arbitrary JavaScript code, any information obtained from the external server as JavaScript code or an URI that points to a script, can be easily applied. It is not even necessary for the information to be JavaScript code. If there is an adequate parser at the client side of the application, then the modification can be expressed even in a language close to a natural language.

Besides a parser or the evaluation of JavaScript code, the client side of the application should have an application programming interface (API) that allows at least storing and retrieving options for the customizations (like Greasemonkey). Since we use OpenID and Attribute Exchange, this can be done with Fetch and Store requests to the external server to obtain and put back optional values for the modifications.

In a normal execution, the optional values and the modifications to be applied should be fetched on the same request. When the customizations need to store a value for future reference, then a store request should be generated.

2.3. An architecture for ubiquitous client side customizations

We describe here an architecture for ubiquitous client side customization of Web applications. The architecture considers three parts: the Web application to be customized – composed of client and server components – , the browser where this application is running, and the external service, where the information needed for customization of the application is stored (Figure 2.1).

The browser does not need to be modified. In fact, it only acts because of the client side component of the Web application and indirect responses from the external service – detailed later. This is a key aspect in the proposal, because as we explained before, any special behavior or feature required on the browser immediately compromises the ubiquity on the customized version of the application. Thus, by interpreting JavaScript code as usual, the browser communicates to the application to be modified, indirectly to the external service, and apply the modifications to the application.

The external service is very close to any existing OpenID Provider (OP) supporting Attribute Exchange extension. This extension supports fetch and store requests of arbitrary attributes. However, only when the OP and the Relying Party (RP) – the application in this case – understand the attributes and give them the required semantic, the standard can be useful. Thus, for example, the standard permits to ask for the first name of the OpenID Identifier's owner. If both OP and RP understand that attribute, the communication is

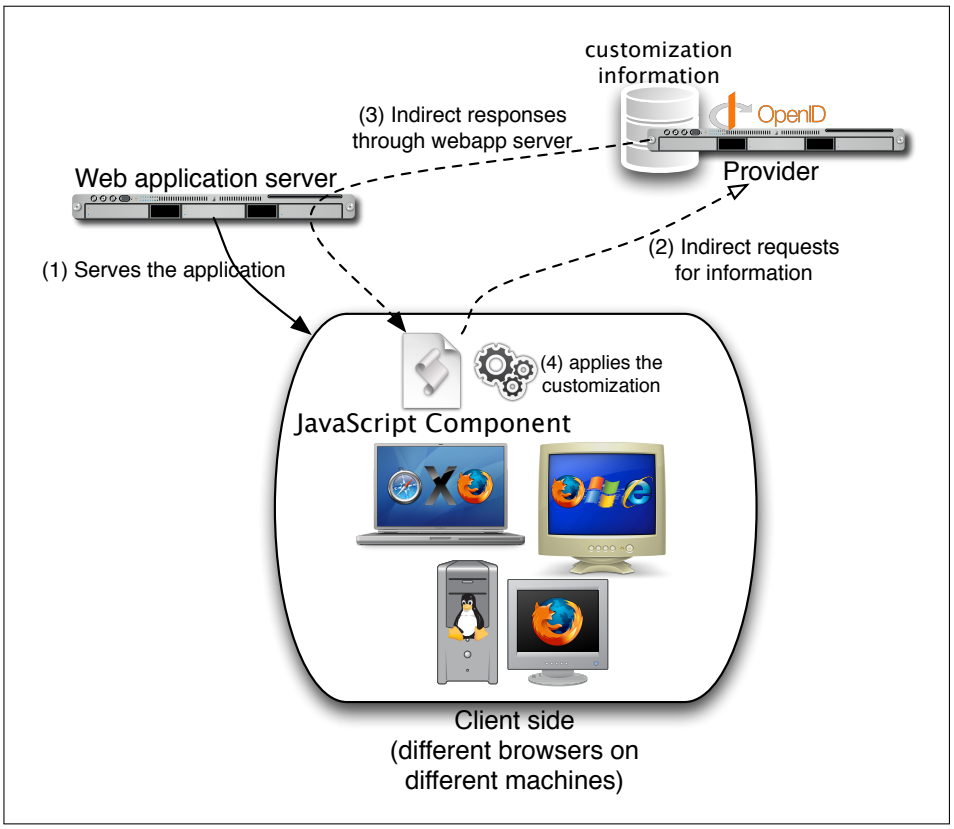

FIGURE 2.1. The architecture has three parts: the Web application (living in the server and the client), the client browser and the external service that holds the customization information.

possible, but if the RP asks for favorite songs and the OP doesn't know what is a favorite song, then there will be no useful response to that fetch request (the OP will just ignore that attribute).

There are many attributes defined nearly as standards (Sxip Identity, n.d.) that any OP should understand and be capable of manage. Thus, the only development needed for an OP to support the architecture is to be capable of managing and understanding the new necessary attribute types that are needed for storing and fetching the customization information. This comes along with any new user interface to give the user the possibility of accept or reject petitions from applications that want to fetch modifications.

The application, on the other hand, can be of any type, can use any server side technologies and can have any design both of software and user interface. The only requisite

for these applications is to use standard client side technologies like HTML, CSS and JavaScript (Flash, Silverlight, and other plug-in dependent client side technologies are not covered).

One of our goals was that the solution imposed minimal additional development effort on existing applications. This objective is indeed fulfilled since the changes needed on existing applications are limited to the simple inclusion of a small script in Web pages that want to offer client side customization and provide a RP endpoint to which OpenID responses can be sent and passed back to the JavaScript component to apply the fetched modifications.

The JavaScript component is independent of server side technologies, so a unique script can be provided with no development cost (just including it in the Web page). The RP endpoint at the Web application's server is a little more complicated and, if security is a matter – as always should be – then it is dependent of server side technologies because it must execute code on the application's server. Nevertheless, the code needed is very simple and since there are OpenID libraries for most server side technologies it is easy to implement. Furthermore, since there is no dependence on the actual Web application, libraries for many server side technologies could be generated and delivered together with the JavaScript library.

We present an outline of data flow of this architecture in action, to clarify how this works (Figure 2.2):

(i) The JavaScript component on the application shows the user interface to start the process of customization.

(ii) If the user completes the required data (an OpenID Identifier), the JavaScript components queries the server for discovery of necessary data (maybe none) to start an OpenID authentication.

(iii) With data collected from user and server, the JavaScript component starts an OpenID authentication and fetches the information to customize the Web application.

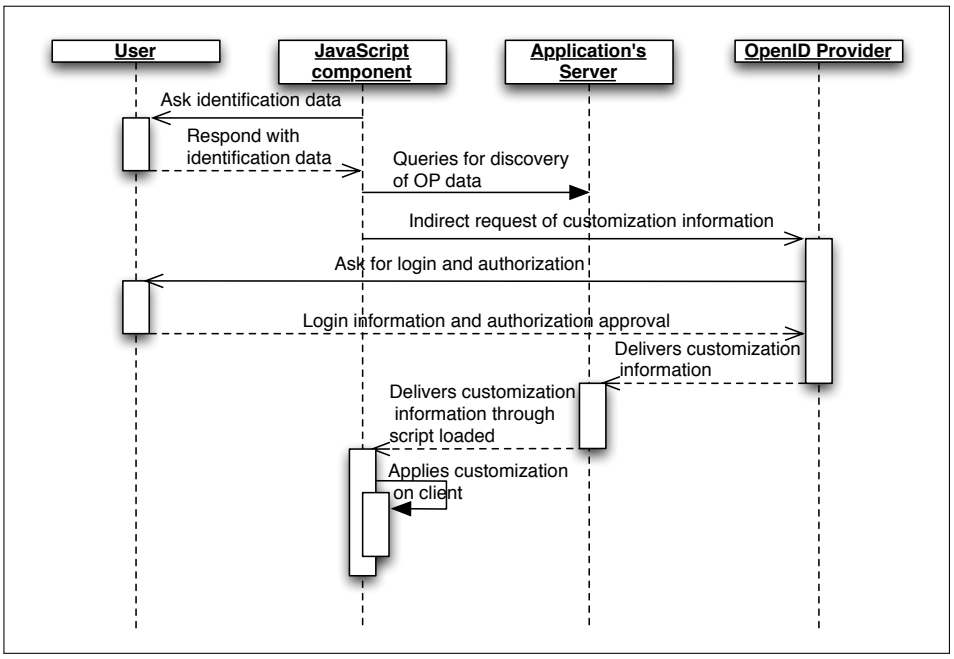

FIGURE 2.2. Sequence diagram of a customization process, where the application's server just acts as an OP discovery and RP endpoint to receive the OP indirect response and pass it to the JavaScript component, which does most of the work.

(iv) The OpenID Provider asks the user to authenticate and to accept delivering the information asked by the JavaScript component.

(v) The OpenID Provider responds to the application endpoint with the requested information.

(vi) The application endpoint loads a script that passes the information to the JavaScript component.

(vii) The JavaScript component interprets the information (modifications to apply and their configuration values) and finally applies the customization to the rendered Web page.

The first two steps and those in which the user authenticates himself and authorizes the exchange of information in the OpenID Provider must be done only the first time in the duration of a Web session because after that, the necessary data and the authentication in the OpenID Provider can be done almost in background, without user intervention.

2.3.1. Prototype Implementation

To test the technical feasibility of the proposed architecture, we developed a prototype implementation that included the three parts explained in previous section. The modifications used in this prototype were in the form of Greasemonkey userscripts (with some limitations because the JavaScript component cannot emulate the full API provided by the Greasemonkey extension such as the method for cross-site XMLHTTPRequests).

The OpenID Provider was developed in Java using OpenID for Java (Bufu, Baur, & Scurtescu, 2009) library and Apache Tapestry 5 Web framework (Ship, 2009). It has the user interface to login with a registered account and it can receive OpenID requests according to OpenID 2.0 Specification and Attribute Exchange Extension 1.0 (AX) Specification. Comparing it with other current OpenID providers, the main difference is that our implementation can understand two special attribute types in the AX fetch request containing the modifications to be applied to the origin hostname of the request and the optional key-value pairs of configuration. The implemented OP can have values for these attributes stored and associated to the user account so it can respond to the fetch request with this information.

The application example to be modified by client side customizations is very simple: it is just a section of an online music shop that lists available songs with a link to a free mp3 sample file for download. It was also developed in Java with Apache Tapestry 5 Web framework and ran in a Jetty (Eclipse Foundation, 2008) Web server.

This test application was then modified in order to include the third part involved in the proposed architecture: a JavaScript component and an OpenID RP endpoint.

The JavaScript component depends on the Prototype JavaScript framework (Stephenson et al., 2009). Although this is not necessary at all, it was done for ease and clarity of the implementation. This JavaScript component, when inserted at the bottom of a Web page, inserts a little box on the bottom-right corner that triggers the user interface when clicked

(also contained in this component) for asking the required parameters to fetch and apply the customizations. Once obtained the modifications, this component evaluates the userscripts so the code is executed and the customization is applied.

The JavaScript component generates the requests to the OpenID Provider using an HTML iframe or popup depending on the mode of OpenID requests needed (immediate or setup mode). When it needs information to generate the correct OpenID requests (like discovering the OpenID provider URL from the identifier), it asks them via Ajax requests to the OpenID RP endpoint also developed (and integrated into the application) which can create cross-domain requests. Besides this task, the endpoint also receives OpenID Provider responses and passes the data to the JavaScript component. This is done by loading a Web page (in the same popup or iframe created by the JavaScript component) that also loads the JavaScript component and passes the data via JavaScript callback functions – which is possible now since the page is back onto the same domain. This form of JavaScript OpenID authentication is based on "openid-selector" (2009) and "OpenID Demo" (Ellin, 2009).

To increase compatibility with existing Greasemonkey userscripts, this JavaScript library also provides an API close to the one Greasemonkey has, so many userscripts can be used with this prototype without changing them. Of course, almost all userscripts were written to run on Firefox (since Greasemonkey is a Firefox extension), so in order to run those userscripts on different browsers it may be necessary to alter them.

The JavaScript component itself was tested on Firefox 3.5, Internet Explorer 8 and Safari 4. However, in order to successfully apply a customization in some specific browser, the customization in the form of a userscript must be compatible with that browser too because it will be evaluated with the specific browser's JavaScript engine.

Two userscripts that modify the test Web application were loaded and configured into the OpenID Provider and were fetched and applied on the test Web application – with the JavaScript library and RP endpoint added – successfully. One of these userscripts transforms the links to sample mp3 files into a Flash player that can reproduce the mp3

file via streaming. The second userscript just does some visual alterations to customize the application to another visual taste (Figure 2.3).

(A) Original Web application (B) Customized Web application

FIGURE 2.3. The Web application is customized in the browser through customizations in the form of Greasemonkey userscripts. This is done however keeping the universal access of the customized application.

With minimal additional development we were able to get an application that conforms to the proposed architecture. The users can enjoy a customized version of the application with the same universal access of the original application. The customized version of the application was tested on different machines (Windows, Linux and Mac OS machines) and on different browsers (Firefox, Safari and Internet Explorer), behaving exactly the same on all of them.

2.4. Conclusions and Future work

2.4.1. Conclusions

The prototype implementation of the architecture demonstrated that it is indeed capable of storing the information required to apply customizations on Web applications, loading this information on the client side of the application using only an OpenID identifier and applying them right in the browser, without installing anything on the client machine and therefore making it independent of it.

The prototype also demonstrates that minimal effort is required to get ubiquitous customization beyond the preferences or possibilities originally created by application developers. Users get a customized version of the original application, adapted to their particular needs and preferences, and easily accessible from any computer with any modern browser.

2.4.2. Future Work

Current works on new Web specifications could well minimize the changes required on the Web application further. For example, allowing cross-domain JavaScript requests (like the JSON Requests (Crockford, 2009) proposal) and adaptation of OpenID protocol to that type of requests, the endpoint functions should be less needed, or with a little more adaption of the protocol, unnecessary.

The requests needed to load and apply the customizations on Web applications could be also minimized by using cache. For example, the requests to the OpenID Provider could be done only once per session and parts of that information stored in cookies and browser cache, thus eliminating the need of per Web page requests.

Chapter 3. CONCLUSION AND FUTURE RESEARCH

More aggressive approaches in client side customization appeared because users of Web applications were not satisfied with the customization options offered by original developers. Users required more control and better adaptation capabilities in Web applications so they could modify them and make them meet their needs and tastes. Unfortunately, current efforts solve this limitation at the cost of transgressing a basic principle of Web applications: the modified application is not universally accessible.

The alteration of the browser as a mean to modify Web applications in the client, without any cooperation from developers, leads to important limitations. For this reason we favor joint venture approach where the original developers, third party developers and final users participate. The application developer provides customization support to the end users, something that does not add much to the development costs. In addition, third party developers can greatly intensify development of customizations that are beyond normal users' potential.

We presented an architecture that enables Web application developers to extend customization options offered to their users, with minimal development cost on Web applications. These customizations are applied in the client, giving more control and possibilities to users, but in a way that is neither tied to a specific browser nor to the installation of additional software. Since the architecture is not browser dependent and does not store the customizations in the client, it keeps the ubiquity of the modified version of the Web application.

3.1. Review of the Results and General Remarks

The spreading of new technologies, standards and specifications is a long process, specially when browsers must support them. The prototype implementation of the proposed architecture proved its feasibility within the current Web infrastructure, so our architecture should not require a time gap for its adoption.

The prototype also demonstrated the minimal effort needed to integrate this new functionality to Web applications. This integration adds almost infinite possibilities to how Web applications can be adapted by every particular user without increasing development cost to both existing and new Web applications.

The use of Greasemonkey-like userscripts to represent the modifications to be applied in the Web application gave us an important advantage. A potentially large base of already developed userscripts could be available for users if existing Web applications adopt this or a similar form of the developed prototype. This could dramatically increase the immediate value of a Web application adopting our architecture.

With the prototype implementation of our architecture, we obtained similar results to one of the successful examples of client side customization: the Greasemonkey Mozilla Firefox extension. Our approach however does not depend on Mozilla Firefox, software installations or local storage of customizations, therefore providing a universally accessible and client side customized Web application.

3.2. Future Work

With current works in progress on Web specifications, it should be possible to completely avoid (or at least minimize even further) the server component of the customization library in the future. These works include forms of cross domain JavaScript requests such as JSON requests or secure JavaScript messaging between iframes from different domains. Although this would also require adaptations of the OpenID protocol, it would improve this proposal both in terms of facilities to developers and simplicity of the customization library.

The efficiency of the prototype implementation could also be improved. Currently, we do not consider possible points of caching using browser's cache or cookies. This could minimize the required OpenID requests and therefore improve the process of applying customizations.

The form of customizations is another point of extension. The prototype used user-scripts, but the proposed architecture is not tied to any particular customization form. Therefore, more accessible customization forms such as Chickenfoot's scripts could be possible if a parser is provided. Even a customizations generator could be integrated to the customization library to let users create modifications from within the application itself.

Finally, another important topic is sharing customizations. The database userscripts.org shows people are willing to share their efforts in customizing applications. With the proposed approach a sharing section could be easily integrated, where users publish customizations or look for customizations from other users. This opens the door for massive adoption because of the increased value of sharing.

References

Andrews, J. (n.d.). Userscripts.org: Power-ups for your browser [Computer software]. Author. Available from http://userscripts.org/

Asleson, R., & Schutta, N. T. (2005). *Foundations of Ajax* (illustrated ed., Vol. 13). United States: Apress.

Baldwin, J., Rowson, J. A., & Coady, Y. (2008). PrintMonkey: giving users a grip on printing the web. In *Proceeding of the eighth ACM symposium on Document engineering* (pp. 230–239). Sao Paulo, Brazil: ACM.

Baresi, L., Garzotto, F., & Paolini, P. (2000). From Web sites to Web applications: New issues for conceptual modeling. In S. Berlin & Heidelberg (Eds.), *Conceptual modeling for E-Business and the Web* (Vol. 1921/2000, p. 89-100). Springer.

Berners-Lee, T. (2006). The World Wide Web - Past, Present and Future. *Journal of Digital information*, *1*(1). Available from https://journals.tdl.org/jodi/article/view/3/3

Bigham, J. P., & Ladner, R. E. (2007). Accessmonkey: a collaborative scripting framework for web users and developers. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)* (pp. 25–34). Banff, Canada: ACM.

Birsan, D. (2005). On plug-ins and extensible architectures. *Queue*, *3*(2), 40–46.

Bolin, M., Webber, M., Rha, P., Wilson, T., & Miller, R. C. (2005). Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology* (pp. 163–172). Seattle, WA, USA: ACM.

Brooks, T. A. (2006). No bad Web pages: reader empowerment and the Web. *Information Research*, *11*(3), 257-257. Available from http://informationr.net/ir/11-3/paper257.html

Bufu, J., Baur, T., & Scurtescu, M. (2009). Openid for Java [Computer software]. Author. Available from http://code.google.com/p/openid4java/

Crockford, D. (2009). *JSONRequest.* Retrieved 2009-10-03, from http://json.org/JSONRequest.html

Eclipse Foundation. (2008). Jetty Webserver [Computer software]. Author. Available from http://www.eclipse.org/jetty/

Ellin, B. (2009). Openid Demo [Computer software]. Author. Available from http://openid-demo.appspot.com/

Fischer, G. (2001). User Modeling in Human–Computer Interaction. *User Modeling and User-Adapted Interaction*, *11*(1), 65–86.

Fitzpatrick, B., Recordon, D., Hardt, D., Bufu, J., & Hoyt, J. (2007). *Openid Authentication 2.0* (Final ed.; Tech. Rep.). OpenID Foundation. Available from http://openid.net/specs/openid-authentication-2_0.html

Garrett, J. J. (2005). *Ajax: A new approach to web applications.* Adaptive path. Retrieved Nov 10, 2009, from http://adaptivepath.com/ideas/essays/archives/000385.php

Gjoka, M., Sirivianos, M., Markopoulou, A., & Yang, X. (2008). Poking facebook: characterization of osn applications. In *Proceedings of the first workshop on Online social networks* (pp. 31–36). Seattle, WA, USA: ACM.

GmailGreasemonkey10API. (2007). Retrieved 2009-12-05, from http://code.google.com/p/gmail-greasemonkey/wiki/GmailGreasemonkey10API

Hardt, D., Bufu, J., & Hoyt, J. (2007). *OpenID Attribute Exchange 1.0* (Final ed.; Tech. Rep.). OpenID Foundation. Available from http://openid.net/specs/openid-attribute-exchange-1_0.html

Jackson, C., & Wang, H. J. (2007). Subspace: secure cross–domain communication for web mashups. In *Proceedings of the 16th international conference on World Wide Web* (pp. 611–620). Banff, Alberta, Canada: ACM.

Kazuyoshi, K. (2008). Greasekit – User Scripting for all WebKit applications [Computer software]. Author. Available from `http://8-p.info/greasekit/`

Kristol, D., & Montulli, L. (1997). *HTTP state management mechanism* (RFC No. 2109). Fremont, California, USA: Internet Engineering Task Force.

Lieuallen, A., Boodman, A., & Sundström, J. (2009). Greasemonkey [Computer software]. Mozilla. Available from `https://addons.mozilla.org/firefox/addon/748`

Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E. M., & Kandogan, E. (2007). Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 943–946). San Jose, California, USA: ACM.

Millett, L. I., Friedman, B., & Felten, E. (2001). Cookies and Web browser design: toward realizing informed consent online. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 46–52). Seattle, Washington, United States: ACM.

Mitchell, A. (2009). Gtdinbox for gmail [Computer software]. Mozilla Foundation. Available from `https://addons.mozilla.org/en-US/firefox/addon/3209`

Nielsen, J. (2009, August 17). Customization of UIs and Products. *Alertbox*. Available from `http://www.useit.com/alertbox/customization.html`

openid-selector [Computer software]. (2009). Available from `http://code.google.com/p/openid-selector/`

Opera Software. (n.d.). *Opera: Tutorial – user javascript.* Retrieved Oct 20, 2009, from `http://www.opera.com/browser/tutorials/userjs/`

Pierce, M. E., Fox, G., Yuan, H., & Deng, Y. (2008). Cyberinfrastructure and Web 2.0. In L. Grandinetti (Ed.), *High Performance Computing and Grids in Action* (Vol. 16, p. 265-287). Amsterdam, The Netherlands: IOS Press.

Recordon, D., & Reed, D. (2006). OpenID 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management* (pp. 11–16). Alexandria, Virginia, USA: ACM.

Reify Software. (2008). Reify Turnabout [Computer software]. Author. Available from `http://software.filestube.com/software,520e1a53,Reify+Turnabout.html`

Ship, H. M. L. (2009). Apache Tapestry 5 [Computer software]. Apache Software Foundation. Available from `http://tapestry.apache.org/tapestry5`

Solomon, M. (2009). Pithhelmet [Computer software]. Author. Available from `http://www.culater.net/software/PithHelmet/PithHelmet.php`

Stephenson, S., Fuchs, T., Dupont, A., Langel, T., Porteneuve, C., & Zaytsev, J. (2009). Prototype JavaScript framework [Computer software]. Author. Available from `http://www.prototypejs.org/`

Sxip Identity. (n.d.). *Attribute types.* Retrieved 2009-10-20, from `http://www.axschema.org/types/`

TIOBE Software. (2009, December). *TIOBE software: Tiobe index.* Retrieved 2009-12-08, from `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Trapani, G. (2009). Better Gmail 2 [Computer software]. Mozilla Foundation. Available from `https://addons.mozilla.org/en-US/firefox/addon/6076`

tr.im resurrected. (n.d.). Retrieved Oct 20, 2009, from `http://blog.tr.im/post/160697842/tr-im-resurrected`

Trixie [Computer software]. (2005). Available from http://www.bhelpuri.net/Trixie/

Turner, S. (2001). Active Browsing. In *Proceedings of the IASTED International Conference Internet and Multimedia Systems and Applications* (pp. 181–186). Honolulu, HI, USA.

Turner, S. R. (n.d.). Platypus [Computer software]. Mozilla. Available from http://platypus.mozdev.org/

Zalewski, M. (2009). Same–origin policy. In *Browser Security Handbook* (chap. 2.1.1). Retrieved 2009-12-03, from http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy