Cross-Domain Embedding for Vaadin Applications

Janne Lautamäki and Tommi Mikkonen

Department of Software Systems, Tampere University of Technology,
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{janne.lautamaki,tommi.mikkonen}@tut.fi

Abstract. Although the design goals of the browser were originally not at running applications or at displaying a number of small widgets on a single web page, today many web pages considerably benefit from being able to host small embedded applications as components. While the web is full such applications, they cannot be easily reused because of the same origin policy restrictions that were introduced to protect web content from potentially malicious use. In this paper, we describe a generic design for cross domain embedding of web applications in a fashion that enables loading of applications from different domains as well as communication between the client and server. As the proof-of-concept implementation environment, we use web development framework Vaadin, a Google Web Toolkit based system that uses Java for application development.

Keywords: Vaadin, JSONP, cross-domain applications.

1 Introduction

Web applications – systems that resemble desktop applications in their behavior but are run inside the browser – are becoming increasingly common. The current trend is that web pages have dynamic components side by side with the traditional web content, such as static text and images. These dynamic components can be small widgets that for instance display current weather information or stock exchange data, or even full-fledged web applications that offer a service related to the theme of the web page where they are located [1].

Creating dynamic web pages is much more complex than building plain old web pages. However, since numerous applications are readily available, it would be attractive to simply reuse applications that already exist instead of building them from scratch for a particular application. This would be a step towards 'mashware' envisioned in [2], where the idea of composing complex applications is out of components readily available in different web sites. Furthermore, there are real-life examples of this happening. For instance, it has been possible to embed Google Maps (*http://maps.google.com/*) functionality as a part of any web site for some years by now. Similarly, the popularity of embedded Google Maps components verifies our assumption that application embedding is a valued feature.

A. Harth and N. Koch (Eds.): ICWE 2011 Workshops, LNCS 7059, pp. 62–73, 2011.

Unfortunately, reusing web applications that already exist in some web site is not straightforward, even if the applications could be downloaded in an uncomplicated fashion. The same origin policy inside the browser, defined to protect web pages from malicious code, prevents a document or script loaded from one web domain from getting or setting the properties of a document from another domain [3]. Furthermore, creating web pages that host such dynamic applications is much more complex than building plain old web pages to begin with, because the page must offer hosting services to the application. Consequently, implementing a service that readily provides small applications in web pages requires considerably more attention than simple reference to the service in the embedding web page.

At present, there are two obvious ways to perform application embedding. A web application can be embedded inside a or an <iframe> element. Both of the approaches have been associated with consequences that deplete their potential, and hence they are not too widely deployed solutions. These properties will be discussed in more detail later on.

In this paper, we describe how to embed cross-domain web applications in a web page. While the design itself is generic and technology-independent, we demonstrate the approach with a server side web development framework called Vaadin [4]. The implementation combines strengths of and <iframe> based approaches, but is not plagued by their main weaknesses. Furthermore, the implementation is kept as simple as possible for the developer who wishes to embed an application in a web page, to the extent that only a single line of HTML is needed for taking an application to use.

The rest of the paper is structured as follows. In Section 2, we give an overview to the Vaadin Framework and its particularities that are important for our implementation. In Section 3, we explain the details of the embedding we have enabled in detail, and in Section 4, we discuss some sample applications. In Section 5, we provide some directions for future work, and in Section 6 we draw some final conclusions.

2 The Vaadin Framework

The Vaadin Framework extensively relies on the facilities of Google Web Toolkit, GWT (*http://code.google.com/webtoolkit/*) [5]. GWT is an open source development system that allows the developer to write Ajax-based (Asynchronous JavaScript and XML) web applications using Java that can then be compiled to highly optimized JavaScript, which can be run in all browsers. In the Vaadin Framework, GWT is used for compiling web browser client-side engine and for Ajax-based communication – in essence asynchronous *XMLHttpRequest* calls – between a client and the server (Figure 1).

Consequently, from the developer perspective individual Vaadin applications can be implemented like Java Standard Edition desktop applications. The only difference to common Java applications is that the developer has to use the specific set of Vaadin UI components. For customized look and feel, the developer can use Cascading Style Sheet (CSS) files or directly modify the properties of the components in Java.

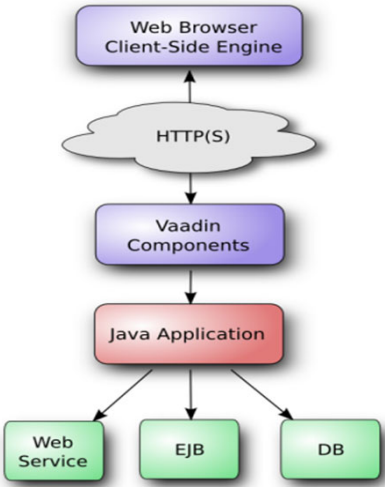

Fig. 1. General architecture of Vaadin [4]

In the Book of Vaadin [4], the main introductory paper documentation regarding the Vaadin Framework, two different approaches for embedding a Vaadin application as a part of a static web page are described – the Vaadin application can be embedded inside a or <iframe> element. For the approach, the downside is associated with the same origin policy, which makes it difficult to embed applications from other domains as a part of a website. Applications running on the same domain can be embedded, but this means that a copy of the embedded application must be available in the same domain. With the <iframe> approach, the same origin policy related problems can be overlooked, and applications can be added from any domain inside an <iframe>. However, upon enabling the download of applications, the <iframe> also traps the application inside it. Consequently, if an application running inside an <iframe> opens a dialog, the dialog stays inside original borders of the <iframe>. In contrast, with a the application would appear to be a part of the web page as the new dialog could be opened anywhere on the web page.

As an example, Figure 2 shows the same Vaadin application embedded in an <iframe> and in a . In <iframe> based embedding the "My Window" dialog is trapped inside the <iframe>, whereas with approach the dialog can move freely around the page. By examining the <iframe> solution, it is obvious that left side of the "My Window" dialog has been cut away by the <iframe>.

Furthermore, another difficult with <iframe> approach is that communication between the page and the application inside the <iframe> is limited to using URL fragment IDs or hacks of different kind as with approach the embedded application can be manipulated using the document object model (DOM) tree.

When embedding a Vaadin application in a web page, the situation gets even more complex, and being able to access an HTML file from another domain is just the beginning. We must also support communication between the client running inside the browser and the web site that runs the server part of the application. This is commonly implemented using the *XMLHttpRequest* mechanism, an integral part of Ajax, to enable data transmissions, which, even if we could download the client part of the application, would be blocked due to the same origin policy based.

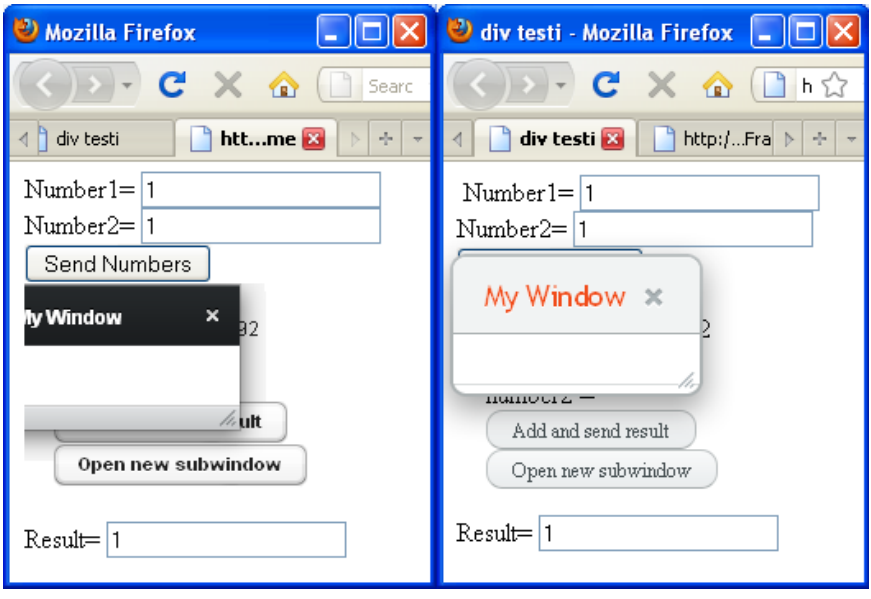

Fig. 2. Application in <iframe> (left) and in (right)

However, there are some notable exceptions to the same origin policy. Images, scripts and style sheets downloaded from another domain are not subjected to the same origin policy. Consequently these formats can be used for circumventing the restrictions related to the policy. Next, we explain how this can be implemented for Vaadin applications.

3 Embedding Vaadin Applications

The traditional way to implement a web application that comprises a Vaadin component from another web site would be to use a proxy to communicate with the component. However, setting up the proxy is in some cases impossible and in any case the introduction of the proxy is an unnecessary complicating step – the setup procedure of such proxy is much more difficult than simply editing HTML source code. Therefore, we aim at a more developer-friendly solution, described in the following.

In a nutshell, to make Vaadin application embedding work on an arbitrary web page, two problems must be solved: (1) how to download the client-side part of the application, and (2) how to enable communication between the client-side engine and the server. In the following, we discuss two problems separately and explain our solutions.

3.1 Downloading the Client-Side Engine

Vaadin application consists of the server-side engine implemented with Java and GWT based JavaScript client-side engine. The file structure of the Vaadin client-side engine is presented in Figure 3. During the startup, the Vaadin client-side engine is

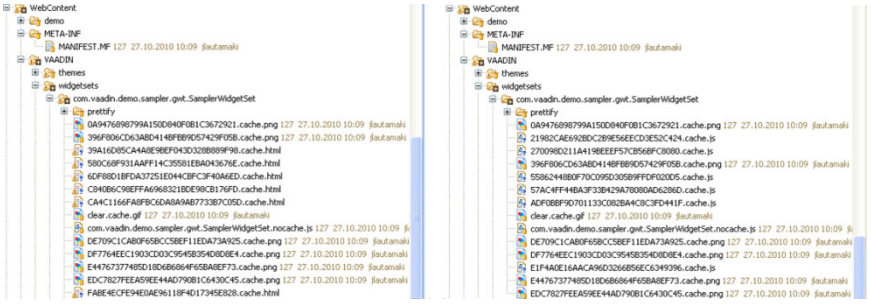

Fig. 3. Original (left) and modified (right) Structure of Vaadin WebContent

downloaded. Starting up the client-side engine is made in four phases listed in Table 1. Images and CSS files that are also needed for loading the client-side engine are not subjected to the same origin restriction, and therefore they are omitted from the table. At the first, browser requests an *index.html* file from the server and the file is returned. In the second phase, the <script> tag inside file *index.html* causes the download of file *vaadinWidgetset.nocache.js*. In the third phase, the *nocache.js* file recognizes the browser version and requests for the right client-side engine version. The server returns page *hashNumber.cache.html*, which contains the client-side engine. In the fourth phase, the system is initiated and user interface description is downloaded from the server.

Assuming that this approach is used for loading an embedded application to foreign web pages, the same origin policy prohibits us from loading HTML pages from other domains. Consequently phases 1 and 3 given in Table 1 will not succeed.

A brief analysis reveals that downloading *index.html* is not necessary when embedding the application. Instead, we could start by requesting the *vaadinWidgetset.nocache.js* script. However, *index.html* contains a lot of additional information, including the locations of associated CSS files, and overlooking this part would cause numerous problems for the developer. As a solution, we moved the modified content of the *index.html* file to the *index.js* file, which can easily be downloaded using <script> tag. The content of the *hashNumber.cache.html* files consist almost completely of JavaScript and just a thin HTML wrapper around them is needed. Consequently we moved JavaScript parts to JavaScript files and use *vaadinWidgetset.nocache.js* to generate the necessary HTML code. The generated HTML downloads scripts from JavaScript files using <script> tags. After this modification, everything that the client-side engine needs can be downloaded, and the engine can be initialized. The method for downloading the initial UIDL is described in more detailed in the next subsection.

After modifications (Table 2), *index.js* is first downloaded inside the <script> tag of the target page. In the second phase, *vaadinWidgetset.nocache.js* is downloaded. In the third phase, the script in *nocache.js* recognizes the browser version and requests for the right *hashNumber.cache.js* version to be downloaded. In the fourth phase, *HashNumber.cache.js* is evaluated, the system starts up, and the user interface description can be downloaded.

Table 1. Initialization of Client-Side Engine

	Client-side engine	Server
1	Requests root or index.html	Returns index.html
2	Requests nocache.js startup script	Returns startup script
3	Requests the client-side engine HTML file	Returns the HTML file
4	Requests the initial UIDL	UIDL returned as JSON
5	Renders user interface and starts waiting for user initiated events	

Table 2. Initialization of embedded Client-Side Engine

	Client-side engine	Server
1	Requests index.js	Returns a JavaScript file
2	Requests nocache.js startup script	Returns startup script
3	Requests the client-side engine JavaScript file	Returns the JavaScript file
4	Requests the initial UIDL	UIDL returned as JSONP
5	Renders user interface and starts waiting for user initiated events	

Since GWT is used to generate the file structure of the framework, we should not modify files by hand, since then changes would be lost at the first time when the framework is modified and recompiled. Fortunately, cross-domain scripting capabilities are something that is commonly needed, and Google has included support for such features in GWT. Therefore, we only added a single line to the GWT configuration file:

```
<add-linker name="xs"/>
```

and recompile the system. The old file structure and the new cross-domain capable web content are presented in Figure 3. Structures are the same apart from the filename extensions – the content of html files is moved inside js files.

Index.js is not generated by GWT and the system can be used and embedded without *index.js* script, but it would be much more complex for an embedding developer. Without the *index.js* script, the embedding developer would have to use paths to the actual *WidgetSet.nocache.js* script, which could be arbitrarily complex. Furthermore, the developer would also have to define the location of the CSS file, which again can be complex. Thus, in a nutshell the reason for using *index.js* is redirecting the call to the actual files and linking the system to the right style sheets.

With the above modifications, embedding the Vaadin application as simple as adding a single line of html to the body of the target document:

```
<script type="text/javascript" src="http://jlautamaki.
  virtuallypreinstalled.com/embedding/index.js">
</script>
```

3.2 Communication with the Server

As already discussed, after being able to download the client-side engine to the browser, we must also enable the communication with the server that runs the actual

application. In Vaadin applications, the client-side engine uses the *XMLHttpRequest* mechanism for all the interactions with the server side (Table 3). There are some straightforward ways to make *XMLHttpRequest* familiar from Ajax work on cross-domain environment. The most obvious alternative is to use <iframe> tags, as already discussed above, but it is also possible to use on <iframe> as a proxy for *XMLHttpRequests* [6]. In addition, W3C has proposed a method for Cross-Origin Resource sharing [7], implemented already in Firefox 3.5, using an HTTP header: *Access-Control-Allow-Origin:* * [8]. Finally, in some old Safari versions there was a security leak that permitted cross-domain *XMLHttpRequests* to work, but this has now been fixed [9].

Despite the possibilities provided by individual browser features and hacks, using them as the basis for long-lasting, generic web services is not feasible. Therefore, we decided to rely on another communication technology, JSONP (JSON with padding) [10], which is commonly used for making cross-domain calls. Furthermore, JSONP works in all modern browsers.

Our JSONP based design gains advantage of the open policy for the <script> tag and uses scripts as a communication channel. A new script can be downloaded when needed, and after the download, the scripts are evaluated and later removed. An injected <script> tag has attribute src. This attribute points to the Vaadin application and downloads the script from there. Messages from the client to the server can be sent as an attribute of the URI:

```
<script type="text/javascript"
  src="http://URL/getjson?jsonp=parseResponse&
  secondAttribute=hello">
</script>
```

Vaadin application gets called the same way as with *XMLHttpRequest* based communication. By default, Vaadin applications use JSON for communication, and consequently the only thing we must add is the padding. With this approach, the browser gets the following message:

```
parseResponse({"Name": "Cheeso", "Rank": 7});
```

In this response, the padding is "*parseResponse()*" and JSON is *{"Name": "Cheeso", "Rank": 7}*. Return value will invoke the *parseResponse* function in the client side engine. The actual message is handled similarly to previously discussed *XMLHttpRequest* messages.

The communication process is summarized in Table 4. First, the user interacts with the UI, for example pushes a button. As a consequence, (second phase) the client-side engine adds a new <script> tag to the web page and sets the appropriate URI for the source of the script. The message passed to the server side is added to the URI as an attribute. As a result the URI for the new script could be for example: *http://url/?okButton=pressed*. The server gets called and it uses *request.getAttribute* to get information that the button has been pressed. Once the server has processed the actions associated with the button, it returns the response to client-side engine, again using JSON with padding. The JSONP message is loaded inside the <script> tag we previously created and is evaluated (the third phase). The evaluation leads to calling the padding function with JSON as a parameter and results are made visible for the user.

Table 3. Original communication

	Client	Server
1	User interaction	
2	POSTS XMLHTTPRequest with requestData	Gets message and returns JSON
3	RequestCallback function gets JSON message	

Table 4. Modified communication

	Client	Server
1	User interaction	
2	New script tag is added, src is url+parameters: "http://url/parameters"	Gets message and returns JSONP
3	JSONP message is evaluated	

In comparison to *XMLHttpRequest*, JSONP has a number of weaknesses. Perhaps the biggest problem is that by allowing cross-domain accesses, the use of JSONP also introduces vulnerabilities. When a JSONP call is made, there has to be absolute trust to the other participant, since JavaScript programs are downloaded as data, and consequently the loaded code can do anything. This has not been considered as a problem in the usual case of Vaadin applications, where everything comes from the same origin. However when embedding and mashupping applications, the problem is that JavaScript loaded from the other domain inevitably gets full access to the content loaded from another domain. In case of the malicious application developers, anything can happen. In our case, we have decided that integrator of the html page just trusts all the widget developers and their services and services have no critical security aspects.

In addition, when using the *XMLHttpRequest* mechanism, there are certain methods for handling errors that take place in communication. In contrast, with JSONP there is no automatic error handling, and any actions to this end should be included in the application. There are certain libraries to simplify this, but in general error checking features are still missing.

Finally, there are a lot of minor implementation-level issues that have been encountered. In particular, in Vaadin applications, messages were already padded using *for(;;);* as a safety mechanism and for making cross site scripting more difficult. This is of course something we had to remove in our implementation, and the obvious consequence is less secure communication.

4 Examples

For demonstrating and explaining the value of our embedding facility, we have created a web page to *http://www.cs.tut.fi/* domain and used it for embedding two different web applications running on the *http://jlautamaki.virtuallypreinstalled.com/* domain. Our sample web page is based on Wikipedia's Body Mass Index (BMI) entry. BMI is a heuristic proxy for human body fat based on an individual's weight and height. BMI is defined as the individual's body weight divided by the square of his or her height. If BMI index falls between 18.5 and 25 then the person is a normal

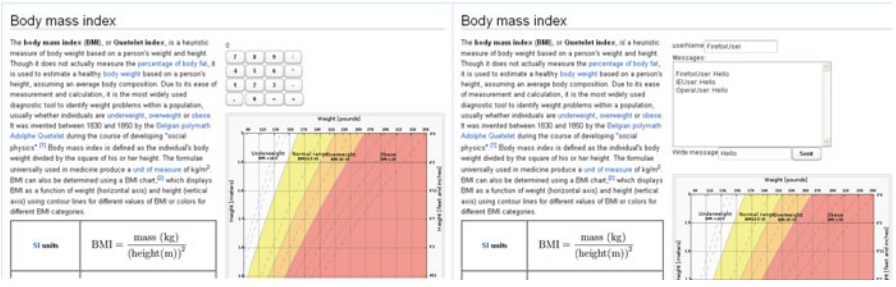

Fig. 4. Embedded Calculator (left) and Chat (right) applications

weighted. Most of the web page development systems can be used to create a simple page like this. The page is just plain text and a couple of pictures and tables. However, for our examples we have spiced up this simple page with two different Vaadin applications.

Consider a user who visits the body mass index page. If the BMI is a new concept for user, the first thing to do is obvious – the user wants to calculate his own BMI and needs a calculator. Of course it would be possible to use a calculator from a mobile phone or a separate desktop calculator application, but it would also be nice to have a calculator embedded directly in the BMI page. The calculator is a web application and is not easily implemented by every web developer. However, given a ready-made calculator, it can be easily embedded in a web page by using our system. This is visualized in Figure 4, and the actual web page is available at available for testing purposes at the address: *http://www.cs.tut.fi/~delga/vaadin/calc.html*. Furthermore, it is also possible to try out the embedding of the calculator application. Only things needed are a web page that can be edited and some trust that we are not trying to do anything hostile. To get the calculator embedded on the web page, only the following script has to be added to the body of the html page:

```
<script type="text/javascript" src="http://jlautamaki.
  virtuallypreinstalled.com/embedding/calc.js">
</script>
```

Our second example, shown in Figure 4 and available at *http://www.cs.tut.fi/~delga/vaadin/chat.html*, is providing a chat widget for the BMI page. The goal of the widget is to enable the user to chat with other users of the page and in this case for example send weight and height as a chat message and other users can then comment on those values and give feedback. It would be possible to create a channel for each page in which chat component has been embedded and then the visitors of the page could communicate with each other. In our example, it is just one channel chat. In the sense of the implementation, this application is considerably more complex since it requires communication between users. Consequently it cannot be implemented with plain client side JavaScript since the server must mediate messages between users. The sequence diagram is presented in Figure 5. In the diagram, we have cheated a little bit for sake clearness. In reality chat clients poll the server once in 2 seconds, but in Figure 5 we have presented communication like messages could be pushed directly from

server to client. Similarly to the calculator, the chat application can be embedded by adding the following script to the body of the html page:

```
<script type="text/javascript" src="http://jlautamaki.
  virtuallypreinstalled.com/embedding/chat.js">
</script>
```

5 Future Work

We still have some considerations and refinement to do with respect to the security aspects. At present, we are not fully aware what kinds of new attacks are possible against the embedded version, which would not be enabled for the original Vaadin applications. However, we do acknowledge that the embedding might introduce some properties that cannot be made as safe as without embedding, no matter how much we try.

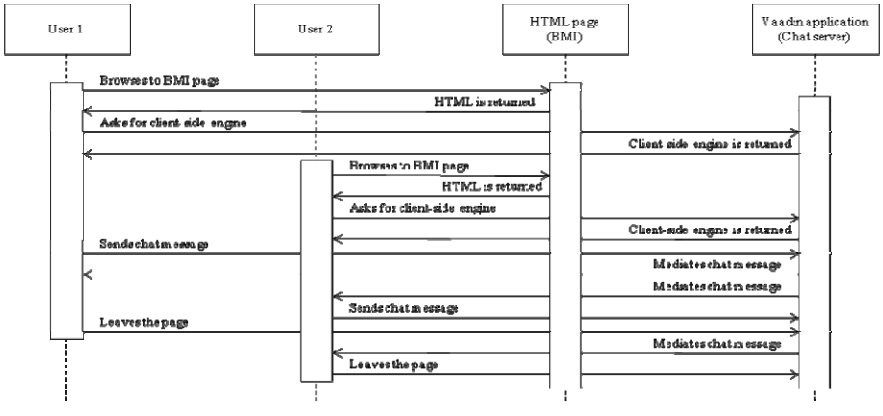

Fig. 5. Sequence diagram with two users chatting

In order to consider the wider use of embedding, an obvious target is to try out our system in real world examples. At present, the whole system is available for testing at *http://vaadin.com/directory#addon/vaadin-xs* and the next step would be to use the system in real use cases with real customers to gain feedback from actual users.

The long-term goal of this work is to create a library of Vaadin applications that could be used by anybody. Assuming that we can figure out the remaining technological details addressed above, there will also be research challenges associated with numerous business related issues. We are in the middle of considering how to set up an ecosystem of widgets, where they could be deployed in embedded mode. On one hand, we should be able to serve web developers who are interested in embedding web applications but do want to implement them, and at the same time we must provide support for developers creating new widgets for others to embed. So far, we have not introduced any business logic that would define how parties hosting embedded applications would get their income, so this is an obvious direction for future work.

Finally, there are some possible advantages that can be gained with other web technologies. In particular, instead of JSONP we could use CORS (Cross-Origin Resource Sharing), a browser technology specification for scripts originating from different domains [7]. Using CORS would introduce some potential benefits, mainly because while JSONP only supports the GET request method, CORS also support the other types of requests. Furthermore, CORS also has better error handling mechanisms than JSONP. As a drawback, CORS is only supported by limited set of modern browsers.

6 Conclusion

In this paper, we presented a way to compose embedded web applications in a fashion that combines the best properties of commonly used approaches without their major downsides. The implementation is composed using the Vaadin Framework, where any completed application consists of the client-side engine. The client-side engine acts as the front end for a Java application running on the server side. The approach was demonstrated with two applications (or widgets) that can be tried out. Furthermore, these applications embedded to any web site by just adding one <script> tag to the body of the HTML document hosting the applications.

As a part of this work, we modified the Vaadin Framework. The modifications have been contributed to the Vaadin community, and are available through Vaadin directory (*http://vaadin.com/directory#addon/vaadin-xs*) for all Vaadin developers.

The most attractive direction for future work – apart from polishing the technology itself – is the creation of an ecosystem where embeddable Vaadin applications could be hosted as a service. In the long run, the vision is to establish a full library of different kinds of web applications, available for embedding to different web pages around the world. This in turn will introduce numerous technical and business challenges for researches as well as for practitioners.

References

[1] O'Reilly, T.: What is Web 2.0: Design Patterns and Business models for the Next Generation of Software. Communications & Strategies (1), 17 (2007)

[2] Mikkonen, T., Taivalsaari, A.: The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering. In: Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research (FoSER 2010), Santa Fe, New Mexico, USA, November 7-8 (2010)

[3] Same origin policy, World Wide Web Consortium (W3C), `http://www.w3.org/Security/wiki/Same_Origin_Policy`

[4] Grönroos, M.: Book of Vaadin (2009) (uniprint)

[5] Perry, B.W.: Google Web Toolkit for Ajax. O'Reilly Short Cuts, pp. 1–5. O'Reilly (2007)

[6] How to make XMLHttpRequest calls to another server in your domain, Ajaxian, `http://ajaxian.com/archives/how-to-make-xmlhttprequest-calls-to-another-server-in-your-domain`

[7] Cross-Origin Resourced Sharing, World Wide Web Consortium (W3C), http://www.w3.org/TR/cors/
[8] HTTP access control, Mozilla Foundation, https://developer.mozilla.org/En/HTTP_Access_Control
[9] Safari same origin hole, The Spanner, JavaScript and general security blog, http://www.thespanner.co.uk/2007/06/29/safari-same-origin-hole/
[10] Remote JSON – JSONP (December 5, 2005), http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/