PAPER

Practice and Evaluation of Pagelet-Based Client-Side Rendering Mechanism

Hao HAN^{†a)}, Member, Yinxing XUE^{††b)}, Nonmember, Keizo OYAMA^{†††,††††c)}, Member, and Yang LIU^{†††††d)}, Nonmember

SUMMARY The rendering mechanism plays an indispensable role in browser-based Web application. It generates active webpages dynamically and provides human-readable layout through template engines, which are used as a standard programming model to separate the business logic and data computations from the webpage presentation. The client-side rendering mechanism, owing to the advances of rich application technologies, has been widely adopted. The adoption of client side rendering brings not only various merits but also new problems. In this paper, we propose and construct “pagelet”, a segment-based template engine for developing flexible and extensible Web applications. By presenting principles, practice and usage experience of pagelet, we conduct a comprehensive analysis of possible advantages and disadvantages brought by client-side rendering mechanism from the viewpoints of both developers and end-users.

key words: rendering mechanism, template engine, Web application, Web UI, XML, XSLT, Java, JavaScript, JSP, DOM scripting, DHTML, mashup, Web service, rich client, mobile browser, MVC, JSON

1. Introduction

Browser-based Web application is one of often used computer software applications designed for information interactivity on the Web. It is reliant on a common Web browser to render the application executable, and usually coded in browser-rendered/supported languages such as HTML (HyperText Markup Language) and JavaScript. At server side, rendering mechanism is indispensable for human-readable presentation and generates webpages based on a template engine. Figure 1 describes the basic architecture of a typical server-side rendering system. In general, a rendering system processes templates and content to generate the output Web documents. A template engine specifies a template and fills the template with the assigned values to obtain the

Fig. 1 Server-side rendering system.

output page, by which the underlying business logics and data computations are separated. In the case of Web application development, this means that “no logic computation in HTML and no HTML in logic computation”.

As shown in Fig. 1, server responses may be determined by requests sent from client side such as data in a posted HTML form, parameters in the URL, or the type information of the Web browser being used. Server-side scripting (program running on the server) is used to change the Web content on various webpages. Such webpages are often created with the help of server-side languages such as ASP (Active Server Page), JSP (Java Server Page), Perl, and PHP. Actually, the design based on server-side rendering is becoming obsolete, since it leads to a long wait for the server to generate and send the entire webpage.

There is a new trend that Web applications are getting more and more complicated with the advancement of diverse Web technologies in order to meet various demands from users such as short response time, flexibility in data presentation, user-friendliness in UI, better interactiveness in operation and so on. For example, Ajax (Asynchronous JavaScript and XML) dynamically interchanges content that sends a request to the server for data. The server returns the requested data subsequently formatted by a client side script, which reduces server loading time because the client does not request the entire webpage regenerated by the server-side rendering. This trend is accelerating the devel-

Manuscript received December 24, 2013.

Manuscript revised April 11, 2014.

[†]The author is with the Department of Information Science, Faculty of Science, Kanagawa University, Hiratsuka-shi, 259–1293 Japan.

^{††}The author is with the Temasek Laboratories, National University of Singapore, Singapore.

^{†††}The author is with the National Institute of Informatics, Tokyo, 101–8430 Japan.

^{††††}The author is with the Graduate University for Advanced Studies (SOKENDAI), Tokyo, 101–8430 Japan.

^{†††††}The author is with the School of Computer Engineering, Nanyang Technological University, Singapore.

a) E-mail: han@computer.org

b) E-mail: dcsxuexy@nus.edu.sg

c) E-mail: oyama@nii.ac.jp

d) E-mail: yangliu@ntu.edu.sg

DOI: 10.1587/transinf.E97.D.2067

opment of interactive and animated websites. Many websites use the DHTML (Dynamic HTML) or DOM scripting technology to create the interactive webpages. It contains HTML, client-side scripts (such as JavaScript), DOM (Document Object Model), etc. The scripts change variables programmatically in an HTML document, and affect the look and function of static content.

Client-side rendering mechanism is one of state-of-the-art solutions responding to this trend. In this paper, we propose a client-side rendering system producing skeletal segments, instead of producing a fully rendered HTML page. Such skeletal segments are called **pagelets** in this paper, which are combined and transformed into an HTML page at the client-side Web browser in our approach. XML (Extensible Markup Language), XSLT (XML Stylesheet Language Transformations [61]), JSON (JavaScript Object Notation) and JavaScript are usually used to generate or update the dynamic content (interactive and animated parts) at client side. Each pagelet presents an individual topic or functional region of webpage. Thus, the independence of each pagelet allows that only the pagelet subjected to changes is transmitted. These interactive webpages offer the users various visual effects, yet they still consume fewer resources on the server.

In this paper, we present a developing framework, and realize the implementation as a simple client-side rendering mechanism. We further conduct a comprehensive analysis of possible advantages and disadvantages in terms of accessibility, caching, personalization and speed. *Our analysis emphasizes issues such as performance, practice and experience at client side rather than the business/processing logic at server side. Common technology of server-side rendering and the specific Web browser plug-ins are not discussed here.* We implement both client-side rendering and page template partition in the real software development projects. Experimental evaluations on rendering cost and development time are also discussed. Our preliminary evaluation results show that our rendering mechanism may entail some learning curve at beginning and increase the development hours, but it benefits in the upgrade/maintenance of an application, and provides testing with independence and flexibility.

The contributions of this paper are summarized as follows:

- We investigate the existing client-side rendering techniques in terms of the interaction, caching, layout customization, refreshing and so on.
- We propose a client-side rendering system based on skeletal segments, namely pagelets, which realizes both client-side XML-based rendering and template partition.
- We evaluate merits, drawbacks and developmental costs of pagelets. The evaluation sheds light on the discussion and reflection of general client-side rendering techniques.

The organization of the rest of this paper is as follows.

In Sect. 2, we explain the concept of pagelet and its difference with portlet. We also introduce some related approaches/systems. In Sect. 3, we describe our approach of client-side rendering mechanism, including the architecture style, pagelet framework and also the integrations of pagelets. In Sects. 4, 5 and 6, we evaluate our approach in terms of a set of software metrics, the performance at runtime, and cost-effectiveness in development/testing. In Sect. 7, we discuss the general design philosophy of client-side rendering mechanism. Finally, the conclusion and the future work are given in Sect. 8.

2. Background and Related Work

2.1 Portlets and Pagelet

Compared to traditional portlets [26] (or WSRP [57]) technologies, there is neither uniform nor mature structure of client-side rendering system widely accepted as rendering paradigm by general Web applications. A portlet is a reusable user interface widget displayed in the context of an enterprise portal environment. Portals typically display portlets as a series of windows or boxes arranged in columns on a webpage with borders, title bars, buttons, headers and footers rendered by the portal framework. Portlets may have the functionality that is tightly integrated with the portal, including a range of settings stored in the portal database.

In contrast, a pagelet defined in our paper is also a reusable user interface widget. The key difference between pagelet and portlet lies in that: portlets are intended for display only in portals, while pagelets are designed to run on any webpage. Any HTML fragment can be a pagelet, and pagelet developers can also take advantage of a large number of APIs available to portlet developers to write pagelets. Pagelets are configurable, and can dynamically interact with other pagelets. Assembled with annotation tags and scripting framework, pagelets have the following characteristics:

- Browser-level variables (session state) can be stored and shared among pagelets, even pagelets that are not on the same webpage. For example, a value entered by the user in one pagelet can be retrieved by another.
- Events are leveraged in page-level. A pagelet can respond when specific events happen, such as webpage loading or focus changing.
- Pagelets can refresh their internal content without reloading the entire webpage.
- Rich defined XML-based pagelets offer flexible accessibility and data caching.

In Sect. 3.2, the implementation of pagelets defined in our paper is elaborated. The implementation of pagelet framework aims at achieving the above four characteristics.

2.2 Related Work

The open source community has created a huge number of

client-side templating solutions. Logic-less templating techniques follow the strict model-view separation rule that there should be little or no logic in templates. They usually provide a clean separation between presentation and logic without sacrificing ease of use. So they are especially well-suited for asynchronous and streaming applications. Twitter[†] uses Mustache [38] and JSON to move significant pieces of functionality to the browser, and Google plus^{††} employs Java-friendly Closure [3]. LinkedIn^{†††} moves from server-side templates to client-side JavaScript templates powered by dust.js [11].

Besides these logic-less templates, there are embedded templating options, which allow developers to embed regular JavaScript code directly within the template. Underscore.js [53] is based on microtemplating and provides good support on functional programming usually expected in Prototype.js [45]. jQuery [30] simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid Web development. Jade [25] is influenced by Haml [20] syntax and implemented with JavaScript for node.js [41]. For client-side JavaScript templating, Jemplate [28] allows the usage of Perl's template toolkit syntax in JavaScript. Django [8] is an open-source Web framework implemented in the Python programming language, and has full support for multi-language applications. Ajax Pages [1] bring easy scripting templates to the client-side and enable rapid development of Ajax Web-based applications without the necessity to manipulate complicated Document Object Models. JavaScript Templates [27] is an open-source templating framework, which offers a set of functions including expression modifiers, special syntax for loops/conditions, and macros. The core engine is written entirely in standard JavaScript and has no critical DOM/DHTML/browser dependencies. JsonFx.NET [32] has an open-source implementation for a browser-side templating framework. It implements the JSonML [31] browser-side templates pattern which has syntax similar to ASP/JSP but uses JavaScript as the templating language. The templates are compiled into pure JavaScript at build-time. QueryTemplate [46] provides DOM and CSS oriented template engine, which completely separates markup and data. It uses load-traverse-modify pattern through jQuery-like chainable API and provides several rapid data injection methods. Although there are plenty of template engines, there is a lack of the comprehensive analysis of practice and evaluation based on various implementations, rich APIs and cross-domain communication.

In addition to the widely used and general-purpose techniques mentioned above, many template generators/systems are proposed for specific purposes such as safety, robustness and so on. Some systems adopt XML oriented technology. Hartmann [21] focused on the template engine enabling safe authoring. An XML-template engine

was proposed for adequate error handling, broad applicability and separation of concerns. Zdun [63] proposed an XML-based page templates consisting of static XML documents and dynamic class definitions. Both parts are dynamically loaded into the Web application environment and composed of the Web objects. The dynamic page template architecture is present for decomposing configurable and representational fragments of Web applications by the end-user with a graphical tool.

Some systems are based on the page transition or data navigation modeling. WebML (Web Modeling Language) [56] is a conceptual visual language for specifying and designing data-intensive and service-intensive Web applications. The core model of the WebML is used to reflect the navigation of user on the Web. T-Web (Template-Web) Generator System is a software application generator for design and construction of Web applications and Web services based on Web Transition Diagram [48], which reflects the Web page Transition. T-Web generator can automatically generate a Web application or Web service from a diagram, which is composed by using T-Web editor without manual programming. Without such manual programming, the calculation ability of generated Web application is still weak.

Some partitioning systems are also used for client-side rendering. Hilda [62] is a high-level language for data-driven Web applications, which allows dynamic partitioning of the Web application between the client and the server in a manner that is completely transparent to the developer. FlyingTemplate [51] produces a skeletal script which includes only dynamic values of the template parameters and the bootstrap code. It is an automatic partitioning system that transfers some workload of generating HTML to the client browsers. There are some component-based frameworks such as Fiz [14], which provides a library of flexible components ranging over various kinds of objects and value editors.

There are also many methods proposed for evaluation of interactive script-based templates. Fernandes et al. gave accessibility evaluation [13] of Web applications by triggering possible events that partially change a webpage. Fire-Crystal [42] is a browser extension that allows developers to indicate interactive behaviours of interest, and shows the specific code (Javascript, CSS, and HTML) that is responsible for those behaviours.

Several works have discussed about whether the “enforcing strict model-view separation” in template engines is possible [44] or impossible [17] to achieve in practice. In our opinion, with the development of Rich Internet Applications, achieving fully strict separation becomes more difficult. The separation should take into account the practicality, and not always be limited to strict separation.

According to documents (APIs and function libraries), references and demos/examples provided by the official websites of these developed methods, Table 1 shows a comparison in the following evaluation factors and levels. Here, the comparison objects are limited to the template engines, rendering systems and template-based application genera-

[†]<https://twitter.com/>

^{††}<https://plus.google.com/>

^{†††}<http://www.linkedin.com/>

Table 1 Comparison between pagelet and others.

system name	output	interaction	rendering	caching	customization	refreshing	editor	skill
Mustache	segment	○	low	low	low	low	×	JSON/JavaScript/Mustache tag
Closure	segment	○	medium	low	low	medium	×	JavaScript/Closure syntax
dust.js	segment	○	low	high	medium	low	×	JavaScript/JSON/dust node
Jade	segment	○	low	low	low	low	×	JavaScript/JSON/Jade syntax
Jemplate	segment	○	low	medium	low	high	×	JavaScript/JSON/perl
Django	segment	○	medium	low	medium	medium	×	JavaScript/JSON/XML/Python
Ajax Pages	segment	○	low	low	low	high	×	JavaScript
JS Templates	segment	○	medium	low	low	medium	×	JavaScript/JST markup
JsonFx.NET	segment	○	medium	low	medium	high	×	JavaScript/JSON/JsonML
QueryTemplate	segment	○	high	high	low	medium	×	JavaScript/jQuery/XML
WebML	application	×	low	low	low	medium	○	Interaction Flow Modeling Language
T-Web	application	×	low	low	low	low	○	none
Hilda	application	×	medium	low	low	low	×	Java/Hilda syntax
FlyingTemplate	segment	○	medium	medium	low	medium	×	JavaScript
Fiz	component	×	low	low	medium	medium	×	Java/JavaScript
pagelet	segment	○	high	high	high	high	×	JavaScript/XML/XSLT

tors. The pure JavaScript libraries (e.g., jQuery, Prototype.js, Node.js, underscore.js) are out of our comparison objects. Because pagelet is constructed based on client-side rendering mechanism and template partition, the comparison is mainly focused on client-side performance.

- Output: produced unit (segment/component/application)
 - segment: an individual topic or functional region of webpage
 - component: a functional segment employed in a application generator
 - application: a Web application with entire functionality and templates/pages
- Rendering (high/medium/low): client-side rendering mechanism is (fully or partially) supported or not
 - high: Client-side XML-based rendering mechanism is well supported on page segment, and templates are easily extensible. Rendering is executable anywhere and there is no need to directly manipulate the DOM.
 - medium: Special syntax and templates compiled into JavaScript at client-side limit the flexibility of rendering and reusing of templates.
 - low: Client-side rendering is not well supported or not available to page segments.
- Client-side performance: client-side flexibility and functionality
 - Interaction (yes:○/no:×): whether it interacts with other same-origin/external segments or components for logic/data composition
 - Caching (high/medium/low): template/data caching at client-side (for reusing and customization)
 - * high: Cached data and templates (e.g., XML/XSLT) are easily reusable at client-side even between different platforms.
 - * medium: Cached data or page segments are partially reusable.

* low: Caching is not supported or functions are not provided in library.

– Customization (high/medium/low): page layout customization

* high: End-users can customize the layout by rearranging/resizing/hiding page segments.

* medium: End-users can customize the layout by rearranging page segments.

* low: Customization is not supported.

– Refreshing (high/medium/low): refreshing page segment

* high: Client-side page segments can be dynamically refreshing with DOM scripting.

* medium: Client-side page segments can be dynamically refreshing with template-dependent DHTML.

* low: Page segment oriented refreshing is not supported at client-side.

• Editor (have:○/not have:×): graphical editors provided in an integrated development environment

• Skill: necessary programming/modeling/markup languages in development excluding HTML and CSS

The main differences are summarized as follows.

- Most of the abovementioned systems compile templates into pure JavaScript before rendering at client-side. Comparatively, pagelet system sends data/templates to client-side for further composition and rendering. The XML+XSLT+JavaScript architecture is the technically unique. For example, compared with QueryTemplate, the partitioned templates and XML bring more excellent performance in client-side refreshing and customization by DOM scripting supported by our function library.
- Pagelet employs widely used XML and JavaScript, and developers do not need to learn new techniques such

as modeling language or configuration of transition diagram. Compared with the traditional template dependent DHTML, the XML-oriented DOM scripting brings more flexible and rich client operation. Moreover, our system provides lightweight function library to reduce the learning curve for the developers. It is available to general Web applications and client environments without the limitation of predefined components or special browsers/plugin-ins.

3. Pagelet-Based Client Side Rendering Mechanism

In this Section, we provide an intuitive and visible image of the proposed client-side rendering mechanism for the ease of analysis and discussion in the following sections. We elaborate the architecture style of the proposed rendering mechanism at client side. Then we illustrate the implementation strategy for the pagelet in our paper, and finally we describe the integration of pagelets among the different webpages.

3.1 Overall Architecture Style

As mentioned in Sect. 2.1, the traditional portlets are only supported in portals. A Web portal is a specially-designed webpage which has information fusion from diverse sources via Mashup (Web application hybrid) or Intranet “dashboards”. Actually, developing portals by portlets implicitly adopts component-based architecture, which refers to a software engineering approach to system design and development. It focuses on the top-down decomposition of features or logical components. Each feature or logical component is modularized and encapsulated for better reuse. Here, a portal is a system and each portlet inside the portal is a component constituting the system.

The essential idea about the architecture of our client side rendering mechanism is that we need to support such kind of component-based architecture for any webpage without being confined to only portals. In Fig. 2, every portlet is assembled as a component in portal pages. Analogically, each pagelet servers as a building block in the webpage in which these pagelets reside. Traditionally, in the component-based architecture, the communication between components is not focused. However, in the architecture of our pagelet-based rendering we concern about the communications of these pagelets, as one pagelet may contain or call some other pagelets from the current webpage or other external webpages. Additionally, these pagelets inside the same page may share the browser-level variables (see Sect. 3.3).

To manage possible dependency among pagelets, patterns such as Dependency Injection pattern[†] or Service Locator pattern^{††} can be used. In this way, loose coupling and

Fig. 2 The components of portals and those of pages based on pagelets.

reuse of pagelets are promoted. The basic idea is to remove the hard-coded dependencies and enable the changes of dependencies, whether at runtime or compile-time. We aim at building composite pagelets that combine and reuse pagelets across multiple webpages. For instance, in a webpage, we can have one composite pagelet which contains a set of small pagelets pertaining to the environment information like *weather status*, *traffic condition* and *pollution index*. Each of these small pagelets can be called from some third party websites without the notice of server side. Sometimes, the aforementioned patterns allow the client side to have a list of candidate widgets on *weather* to be bound to the unimplemented *weather* pagelet at compile-time or at runtime.

From the perspective of software development process, the architecture style of our client-side rendering mechanism actually encourages the loose coupling of server-side development and client-side development. The server side development can focus on the implementation of the logic, while the client side development can focus on the presentation. The communication packages between server and client side are mainly simple data in XML format. Thus, the coupling between server-side application and client-side application is data coupling, a type of loose coupling. Owing to those design patterns like Service Locator pattern, the actual implementation of a pagelet can be delayed to runtime according to the dynamic binding to a third party widget-based pagelet. The development and testing for these pagelets at client side can be individually done due to the loose coupling between each other. From this perspective, the development and testing can be done in a similar way as that in Service Orient Architecture^{†††} (services are quite independent in development and testing, and they are orchestrated together to compose a larger service or to constitute the function of a application).

3.2 Pagelet Framework

We develop a pagelet-based framework to generate flexible and extensible Web applications based on the client-side rendering mechanism shown in Fig. 3. Here, each webpage is divided as a set of static pagelets and dynamic pagelets. A static pagelet always comprises and displays the same information in response to all requests from all users and in all

[†]http://en.wikipedia.org/wiki/Dependency_injection

^{††}http://en.wikipedia.org/wiki/Service_locator_pattern

^{†††}http://en.wikipedia.org/wiki/Service-oriented_architecture

Fig. 3 An example of client-side rendering system.

Fig. 4 A possible example of pagelet layout.

contexts. (e.g., navigation bar and site information placed in page footer, as shown in Fig. 4). A dynamic pagelet presents dynamically generated content that can be accordingly in response to changes in the contexts or conditions shown in Fig. 5. There are two basic ways to achieve this kind of effect/change. Generally, server-side program is used to change the webpage source determined by such conditions as data in a posted form, parameters in the URL, the type information of browser, or a database. At the client, client-side scripting is used to change interface behaviors within a specific webpage in response to mouse or keyboard actions or specified timing events. In this case a dynamic behavior occurs within the webpage presentation. Figure 3 describes the architecture^{††††} of a client-side rendering system based on pagelet framework as follows.

^{††††} Some technical details are ignored here and will be explained respectively in the following sections.

Fig. 5 The workflow of rendering a dynamic pagelet.

1. The server responds to the request from the client with the pagelet container file. A container is a layout page that surrounds or references static pagelets, XML data, XSLT files, or external files such as JavaScript and CSS (Cascading Style Sheets) files in its page body. All files are concatenated dynamically at runtime and downloaded via a single request rather than more HTTP (HyperText Transfer Protocol) requests. The container file also contains the information of one-to-one mapping between the XML data and XSLT file. These files are certificated by the fragile watermarking chain scheme[16] for avoiding the unauthorized modifications and malicious code injections.
2. The main source of an entire webpage at client-side comprises both of static pagelets and dynamic pagelets. A dynamic pagelet is generated at the client-side browser as shown in Fig. 5. Initially, the server sends the XML/XSLT to the client-side rendering system. The XML can be updated by JavaScript functions, and XSLT is used to convert XML into XHTML. Sometimes, the context changes when the user triggers an event defined in XSLT. Then the update of XML will be done. The new XML is provided to XSLT to produce a new XHTML segment.
3. When the end-users trigger the mouse/keyboard events defined in XSLT processors of dynamic pagelets, the variables in the corresponding XML data are changed by JavaScript functions after the context changes in Fig. 5. The XSLT processors retransform the updated XML data and refresh pagelets. Figure 6 describes the core part of a simple example of event definition in XSLT, which shows a list of customer information containing name and other attributes. Text in this list is set as editable status and function *Clear()* (clear node value) is added to respond the *click* event. When the *onClick* is triggered by end-users, the text of customer information is cleared to blank for removing or revision. Table 2 shows a lightweight JavaScript library, which contains some basic client-side functions designed for pagelet framework such as node-oriented operation, rendering, access control, exception handling and com-

```
... ..
<xsl:element
  name="CustomerList:TextEditable">
  <xsl:attribute name='id'>
    <xsl:value-of select='$Name' />
  </xsl:attribute>
  <xsl:attribute name='runat'>
    List
  </xsl:attribute>
  <xsl:attribute name='Text'>
    <xsl:value-of select='$Value'
      disable-output-escaping="yes"/>
  </xsl:attribute>
  <xsl:attribute name='onClick'>
    clear(this);
  </xsl:attribute>
  <xsl:for-each
    select="$Attributes/Attribute">
    ... ..
    ... ..
    ... ..
  </xsl:for-each>
  <xsl:copy-of select="$Events"/>
  ... ..
</xsl:element>
... ..
```

Fig. 6 An example of event (clear) definition in XSLT.

Table 2 List of main JavaScript functions.

function name	function description
loadInfo	load the specified XML data from capsulated XML
getIntValue	get integer value from specified node
getStringValue	get character string from specified node
getNumber	get the number of node with specified name
getItem	get single node with specified name and index
getEncode	get encoding value
getNext	get the next sibling node
getTag	get the anatation tag with specified name
lock	lock the setting information
unlock	unlock the setting information
getLastError	get the code of last error message
getErrorMsg	get the error message with specified index
add	append specified node to the end of sibling nodes
clone	create a deep copy of specified node
replace	replace/update the node value
del	delete the specified node
insert	insert the node at the specified index
exchange	exchange the index/position of specified nodes
remove	remove all of the child nodes from specified node
clear	clear node value of specified node
transform	transform the XML data into HTML by specified XSLT
postMsg	transfer message to other pagelets

munication between different pagelets.

Besides the client XML-oriented JavaScript functions, the layout customization is also supported as follows. The

Table 3 Layout customization functions.

function name	function description
move	move the position of pagelet to arrange the layout
rotate	rotate the pagelet around at the given degrees of X-axis and Y-axis
scale	resize the pagelet to a given size
skew	turn the pagelet in a given angle
hide	hide/undisplay (not delete) the undesired pagelet

```
... ..
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC ... ..>
<HTML>
<HEAD>
<META ... ..>
... ..
<script type="text/javascript"
  src="/js/concatenat_file_list">
</script>
<script type="text/javascript">
... ..
</script>
... ..
<link href="/css/sample.css"
  rel="stylesheet" type="text/css">
</HEAD>
<BODY>
  <TBODY class="..." id="pagelet01">
    <RDF-like description of actions,
      mapping, path01, etc.>
    ... ..
    XML01.load(XML01.xml, path01)
    XSL01.load(XSL01.xslt)
    pagelet01.write(XML01.transformNode(XSL01))
    ... ..
  </TBODY>
  ... pagelet02 ...
  ... pagelet03 ...
  ... pagelet04 ...
  ... ..
</BODY>
</HTML>
```

Fig. 7 An example of container file.

users can arrange the layout or control the visibility easily. Table 3 gives functions designed for layout customization, such as diverse layout arrangement (move, rotate, scale and skew) or easy visibility control (hide).

For the Web application developers, they need to design the container files to decide the layout of pagelets and mapping relation, create the XSLT files for the template of each XML segment, and set the attribute (e.g., writable or read-only, event type) of XML data. Figure 7 presents a simple example of container file including JavaScript and

CSS files in <HEAD> section. The pagelets are concatenated and encapsulated in <BODY> section. Each pagelet is embedded into a <TBODY> and contains the mapping information, XML and XSLT, which are transformed into an HTML segment at client-side.

3.3 Pagelet Integration

After the implementation of pagelet framework, another important issue in our rendering mechanism is how to have a flexible integration model between pagelets without the compromise of performance or security (e.g., access control).

Integration enables users to view diverse pagelets in an integrated manner. Both pagelet combination (application development) and mashup (application hybrid) are aimed to combine information or functionality from two or more existing Web sources to create a new webpage or application. Annotation tags [55] are frequently adopted to display contextual data in proxied pagelets and control functionality from pagelets. Message communication mechanism [19] supports the parallel model and the sequential model based information transfer between pagelets in the integrated application.

The Same Origin Policy (SOP) [4] is enforced by the Web browser as an all-or-nothing mechanism. It is assumed that two pages (pagelets) derive from the same source or origin if the application layer protocol, port and domain from both pages are the same. During the combination into an application, all the contents are from the same origin, and there is an all-trust relationship among the pagelets. In contrast, if these pagelets come from different sources, they run in an isolated environment with no access to others in a no-trust relationship. As mentioned in [47] and IFrame Proxy (IframeXhrProxy) in Dojo [9], cross-domain communication can be realized by replacing fragment identifier of attribute “location”. Besides, *window.name* transport also supports a secured cross-domain communication. Here, we provide *postMsg*, a function simulated as *postMessage* API [58] specified by HTML 5, to safely enable asynchronous communication between DOM windows [22]. It is supported by modern browsers, such as Internet Explorer, Firefox, Opera, and Safari. To send a message, a pagelet needs to invoke the *postMsg* method of its target pagelet for message reception.

In Fig. 8, we show an example of *postMsg* usage with a client-side integration under SOP. The containing page (P.Main), pagelet PL_1 and pagelet PL_2 are from the same domain (<http://p0.com/>) and the content of the third pagelet PL_3 are provided by another domain (<http://p3.com/>). PL_1 is embedded in containing page P.Main by <div>, while PL_2 and PL_3 are embedded in P.Main by using <iframe>. Assume there is an object in each page/pagelet respectively. In Fig. 9, we show the JavaScript code of non-cross-domain access. P.Main and PL_1 can access each other's object (Obj_P.Main and Obj_PL_1) in the all-trust mode under SOP by using *document.getElementById()* since P.Main and

Fig. 8 A client-side integration of pagelets.

```
// For P_Main to get Obj_PL_1
document.getElementById(Obj_PL_1);
// For PL_1 to get Obj_P_Main
document.getElementById(Obj_P_Main);

// For PL_2 to get Obj_P_Main
parent.document.getElementById(Obj_P_Main)
// For P_Main to get Obj_PL_2
window.frames[PL_2].document.getElementById
(Obj_PL_2);
```

Fig. 9 Non-cross-domain access.

PL_1 are in the same document. On the other hand, P.Main and PL_2 are not in the same document (PL_2 due to the another <iframe> tag). In order to access Obj_P.Main, PL_2 calls *parent.document.getElementById(Obj_P.Main)*. P.Main is able to obtain the document of PL_2 by using *window.frames[PL_2].document* and further gets Obj_PL_2 by using *window.frames[PL_2].document.getElementById(Obj_PL_2)*.

However, it is not easy for P.Main and PL_3 to call each other's objects cross the access boundary. In order to perform such cross-domain communication, as described in Fig. 10, P.Main first obtain the window of Obj_PL_3, and then *postMsg* is invoked to send a message. PL_3 registers an event handler *window.addEventListener()* for receiving such request. The handler will be called once the *postMsg* event is received by PL_3 from P.Main. The *event.data* carries the request of retrieving object Obj_PL_3. Then, PL_3 can reply such request by invoking another *postMsg* by using *event.source.postMessage()*, where the *event.source* is the window object of the caller. Finally, PL_3 can send a message back to its caller P.Main. Similarly, P.Main also registers a handler for receiving the returned message as well. In this way, P.Main and PL_3 can exchange messages directly and conveniently between each other.

```
// P_Main's code, for P_Main to get Obj_PL_3
Win_3 = window.frames[PL_3].contentWindow;
// Msg_To_PL_3, the string message sent to PL_3
// http://p0.com is the domain of P_Main
postMsg(Win_3, Msg_To_PL_3, 'http://p0.com');
// register an event handler
window.addEventListener('message', callback);
// define the handler function
function callback(event)
{
// received message Msg_Back_To_P_Main
  data = event.data;
}

// PL_3's code, for Obj_PL_3 to get P_Main
window.addEventListener('message', callback);
// define the handler function
function callback(event)
{
// received message Msg_To_PL_3
  data = event.data;
// Msg_To_P_Main, the message sent back to
// P_Main
// http://p3.com is the domain of PL_3
  postMsg(event.source, Msg_To_P_Main,
    http://p3.com);
}
```

Fig. 10 Cross-domain access.

4. Potential Merits of Pagelet-Based Rendering

Our pagelet-based rendering mechanism is powered by the modern Web technologies such as RIAs, XSLT and so on, which enable the merits in accessibility. Imagining each pagelet as a component in component based software development, the paradigm of pagelet integration facilitates the layout customization and caching of the webpage shown at the client side.

4.1 Accessibility

RIAs (Rich Internet Applications or we call them rich client applications) and its supporting techniques embody an implementation of rendering mechanism in the user agent or in the browser at the client side. They provide more dynamic Web content and more attractive and interactive websites than those in the past. However, not all end-users benefit from such interactivity. For example, users with weak or challenged eyesight access the Web using assistive technologies, such as a screen reader (e.g. IBM Home Page Reader [24]) that delivers audio content. For a fully static webpage, an effective linear presentation is sufficient, but interactive technology involves content updates

```
<contact role="contact list">
  <personal role="personal contact">
    <person role="main contact">
      <name writable="000">Nakata</name>
      <address writable="100">
        Yokohama, Japan
      </address>
      <mainemail writable="100">
        nakata@example1.com
      </mainemail>
      <otheremail writable="111">
        nakata@example2.com
      </otheremail>
      <tel writable="101">81-3-1234-5678</tel>
      ... ..
    </person>
  </personal>
  <business>
    ... ..
  </business>
  <others>
    ... ..
  </others>
</contact>
```

Fig. 11 Names and attributes (embedded in XML) add semantics to webpage.

that are impossible for a linear approach. If a client-side rendering/programming technique is used by a webpage, the screen reader cannot read the page-based operation (page modification), which is particularly problematic for the keyboard navigation essential to accessibility. With the metadata or HTML5 [23], interface developers should be able to adapt interfaces to meet specific needs partially. However, metadata is recommended but not necessary in HTML page, and until now most of developers do not use HTML5 tags.

Rich defined XML is better for screen reader, not only in reading but also for page-based operation [37]. Figure 11, which describes contact information such as name and email address in different categories, shows an example of XML document employed in Fig. 3. It adds semantics to webpage components and content so that assistive technologies can interpret their operation. Regional landmark roles, defining each tag's role, provide a navigable structure within a webpage. Node names and attributes present a guarantee of accessibility of controls. For example, "writable" represents the dynamic content-update notifications: update, remove, and add (0: cannot, 1: can).

The addition of semantic data to XML enables assistive technologies to better represent user interface components and dynamic interactions. Available as a plug-in for Web browsers, Silverlight uses the Extensible Application Markup Language (XAML) [60] to define the user interface. It can dynamically manipulate XML code with the

Fig. 12 Layout rearrangement and resize pagelet (5).

DOM and enable assistive technologies to interpret content. Instead of XAML, Flex Framework uses the Multimedia Extensible Markup Language (MXML) [39] to define user interfaces, communicate with application servers, and execute other operations. It also has APIs defined to incorporate properties and accessible metadata in objects, allowing assistive technologies to interpret content. Applying these modules adds semantics to webpage components and widgets so that assistive technologies can interpret their operation.

4.2 Layout Customization

A configurable page is a webpage equipped with built-in layout personalization and content selection capabilities. Each pagelet, also referred to as a resource displayed within a cell of a layout region, can be rearranged, transformed, and hidden. Client-side XSLT customization is as flexible and powerful as CSS personalization, which is often used in blog pages. For example, it is easy to rearrange/transform the layout or control the visibility by changing the attribute values.

- Layout arrangement/transform (Fig. 12): End-users can move pagelets by dragging and dropping operations to adjust the locations (e.g., $object.style.left = event.clientX - event.offsetX + document.body.scrollLeft$; $object.style.top = event.clientY - event.offsetY + document.body.scrollTop$); which is more compact or suitable for user browsing habits than the default layout arrangement of pagelets. The transform property allows end-users to rotate, scale, skew pagelets by setting different *size* values of CSS (e.g., $object.style.cssText = object.style.cssText + ';$ -webkit-transform: $scale(' + size + ');$ '); for changing how a pagelet is displayed [5].
- Visibility control (Fig. 13): End-users can hide or undisplay the undesired pagelets by setting the property “display” of attribute “style” to “none” ($object.style.display = "none"$;). If the undesired pagelets

Fig. 13 An example of hiding pagelet(s).

are directly removed from HTML documents of webpages, the original execution environment of JavaScript would be broken (the JavaScript could not run normally if the undesired pagelets used/updated in JavaScript programmatically are removed). Exception handling lacks wide consideration in most JavaScript, and an error or exception like “an expected available is not found” would easily cause an abnormal halt in the execution of JavaScript.

4.3 Caching

A Web data caching is a mechanism for temporary storage of Web documents, such as HTML pages, to reduce bandwidth usage, server load, and perceived lag. For Web applications distributed by HTTP, freshness, validation, and invalidation mechanisms are used for controlling caches. Dynamic webpages are often cached when there are few or no changes expected. The page is anticipated to receive considerable amount of Web traffic that would increase loading time for the server if it had to generate the pages on the fly for each request. Despite of the popularization of broadband networks, page or page segment is still the most basic and often used data transfer unit for Web applications. Therefore, it can release the load of server if we reduce the data transfer of page.

Pagelet based client-side rendering mechanism brings more efficient caching performance than the server-side rendering or the client-side rendering without pagelet does. Compared with the traditional page-based caching system, static pagelets would be widely cached and independent of changeable dynamic pagelets. Moreover, the XSLT templates of dynamic pagelets could be reused or cached at client Web browsers and the layout at client side does not affect the cache. This reuse/caching could reduce the data transfer between the server and client. As a experimental example, we reconstructed the response webpage (search result page) of Google Search into XML and XSLT data. The experimental results show that the transferred data size is significantly reduced from 21kb to 7kb as shown in Table 4 if this page is rendered at client side.

Fig. 14 CPU usage percentage of presenting a webpage rendered at server side.

Fig. 15 CPU usage percentage of presenting a webpage rendered at client side.

Table 5 CPU usage and process time.

page	process time (seconds)	CPU usage (max)	CPU usage (script)	CPU usage (parsing HTML/XHTML)
HTML	0.665	77.25%	7.06%	50.67%
XML/XSLT	0.819	86.82%	56.12%	36.81%

Table 4 Response data size comparison.

structure	data format	size (bytes)
Server-side rendering	HTML	20,916
Client XSLT caching	XML	6,794

For the Web browsing at mobile devices, the pagelets can be simplified for better presentation at mobile Web browsers. Android developers also recommend upgrading the Web UI to an XML layout [54]. Moreover, mobile devices could partially share the XML data in server side caching-system with the general desktop Web browsing like XML-based message exchange between different platforms.

5. Speed and Delay

If server-side load is reduced and rendering tasks are moved to client-side, the number of client side rendering tasks will inevitably increase. Client-side rendering and rich client applications usually bring users an impression that they are executed heavily/slowly and high CPU/memory-consuming, which leads to a latency in page loading or refreshing at client Web browser. Actually, such latency is caused by large data streaming in plug-in such as JSON text stream, Flash player and Silverlight, or JavaScript loading. Client-side rendering and script function execution are not high CPU-consuming processes.

The performance data of the data caching examples given in Sect. 4 is captured by Windows Performance

Analyzer Tools [59] in a conventional personal computer (Browser: Internet Explorer 9, CPU: Intel Core i7 2.93GHz, RAM: 4.00 GB). As shown in Figs. 14 and 15, there is no significant difference in CPU usage percentage (the maximum values are both around 24% usage; the loading times are around 0.15–0.19 seconds) between presenting different types of webpages. A similar statistical analysis is conducted on a tablet device (Browser: Google Chrome, CPU: Intel A110 800MHz, RAM: 500 MB). As shown in Table 5, client-side rendered webpage costs more CPU usage and process time, which are mainly caused by client-side script run in XML transformation. At the same time, CPU usage of parsing a transformed XHTML document is less than that of HTML document containing untidy parts. On the whole, client-side rendering only brings tablet users a short delay (about 0.15 seconds), which would be gradually faded with the fast evolution of CPU in tablet and other mobile devices.

As clients are increasingly becoming sophisticated, there are more tasks for a browser to do than before. For websites that rely greatly or incredibly on client-side rendering running, too much JavaScript makes the browser slow. Many large websites use one common set of JavaScript files and one common set of CSS files in every template and webpage. In the methods of including scripts in documents, the most popular is embedding code in a SCRIPT element, adopted in over 88% of all webpages using scripting external scripts. Event handler attributes have similar popularity and are used in 65% of all URLs that used script. The sizes of scripts vary greatly, depending on the used meth-

ods. External scripts are typically the largest, averaging 26,500 characters. Embedded scripts average 2,500 characters, while event-handler attributes and JavaScript protocol URLs average about 500 characters each [34].

Usually, the used JavaScript files and CSS files change over time and often contain elements that are no longer needed by the webpage or anywhere on the website. We need to keep track of what elements are being loaded on each webpage. Client side uses JavaScript libraries that often contain massive amounts of functions, inside which only a handful of those functions are used actually in total execution time [35]. Therefore, in order to speed up presenting webpage, we remove any render-blocking JavaScript or CSS that is not being used on the webpage by using PageSpeed Insights [43]. Discussions about bridging the gap between the browser view of a UI and its JavaScript implementation are elaborated in [33].

Moving JavaScript to the bottom of a webpage also enables all other requests to be processed quickly, which would make webpage appear to be loaded more quickly. The browser can begin rendering faster and do not wait for all of JavaScript to be loaded at the top of webpage. However, not all JavaScript can be moved to the bottom of a webpage. If the JavaScript is a library required by other components on the page to render, it must be loaded early in the page lifecycle.

Another rule in high performance websites is to reduce HTTP requests. This can be done by concatenating external JavaScript and CSS files. Whenever possible, multiple files on the server should be combined into a single file that can be downloaded to the client browser. The ideal setup is to have no more than two references to external JavaScript files and two references to external CSS files per page load. Traditionally, files were concatenated at build time. It is more common for concatenation to happen at runtime using a CDN (Content Delivery Network).

Apache module `mod_concat` [36] is used to easily concatenate files dynamically at runtime in our system. The module works by using a special URL format to download multiple files through a single request only. Instead of referencing each of files separately, `mod_concat` allows them to be combined into one request using the URL: `http://www.example.com/js??main.js,p1.xml,p2.xml`. This URL concatenates `main.js`, `p1.xml`, and `p2.xml` into a single response in the order specified. The double question marks indicate to the server that this URL should use the concatenation behavior.

6. Developmental Costs in Coding and Testing

We have to consider requirements on development efficiency and programming skill if we employ client-side rendering mechanism.

Here, two experiments were conducted to prove our opinions by comparing development time in different conditions. The experimental results are discussed and the pagelet-oriented testing methods are also explained.

Table 6 Upgrade time cost comparison.

version	language	time (hours)	#page	#pagelet
Old	JSP/JavaScript	184	9	/
New	XML/XSLT/JavaScript	80	6	25

6.1 Experiment 1

- **Opinion:** In the model-view separation, XSLT serves as client-side template and offers a great expressiveness, allowing a translator to produce complex HTML documents from XML data. The XML-XSLT based rendering mechanism advocates model-view separation, and model-view separation leads to the division of labor, which benefits in proficiency and concurrence in development.
- **Process:** We upgraded Web UI (User Interface) of an online file information management system, which mainly has *create*, *show (list)*, *show (detail)*, *update*, *delete*, *search* and *confirmation (for create/update/delete)* pages for managing information (e.g., file name, type, access property, description) of the uploaded files. The UI has two versions, of which the old one is developed by JSP (Java Server Pages) and the new one is developed by XSLT. In a version upgrade, a new function needs to be added to personalize webpage layout based on user access authorization. On the premise of the familiarity with the requirements and implementation of both versions, an experimenter did the code update for two versions.
- **Experimenter:** one software engineer (B.E. in CS) who has 3 years' experience in coding (Java, JavaScript, HTML, XML, XSLT) and testing
- **Result:** Table 6 shows that the XML-XSLT architecture is more efficient. The engineer does not need to write an entire template file of a webpage, and the webpage development can be subdivided into functionality-oriented development. For dynamic pagelets, the JavaScript library in Table 2 is used to access or update client XML data. Unlike the traditional dynamically updating client HTML source, the XML data access/update/transformation process is independent of the layout of webpage, and avoids using the HTML tags and HTML-oriented functions such as *innerHTML* or *innerText*, which are not easily reusable. XML-oriented JavaScript functions are customized more easily than HTML parsing functions.

6.2 Experiment 2

- **Opinion:** XSLT files development requires acquaintance of a quite different functional programming language and high proficiency with JavaScript frameworks/libraries (e.g., jQuery and Prototype.js). Therefore, XSLT-XML separation methodology and client-side rendering mechanism might not be ideal if the developers are proficient in JavaScript rather than XSLT.

Table 7 Development time cost comparison.

page (function)	language	programming (hours)	testing (hours)
show	JSP/JavaScript	6	8
update	JSP/JavaScript	12	16
show+update	XML/XSLT/JavaScript	40	64

It would waste the time in analyzing with no notable effect, especially to small personal webpages.

- **Process:** We developed a CIM (Customer Information Management) system, which provides the *add*, *search*, *show*, *update*, and *delete* functions mainly. By the traditional method, it needs five basic functionality pages (and other pages such as listing page): add a new customer (input the customer information), search for a customer (input the search keyword), show the customer information (detailed information such as name, address, telephone, and statistic of purchase history), update a customer information (e.g., change the address, add a new contact), and delete a customer. By using dynamic page, the *show* page and *update* page are merged into a *show+update* page. The users can update the information in *show+update* page by clicking the corresponding value area (trigger *OnClick* event) without the page jumping/transition. The updated information includes the changed user information and the calculated new purchase statistic (e.g., sum, average price, and graphic statistic).
- **Experimenter:** four software engineers
 1. one software engineer (B.E. in CS) who has 2 years' experience in coding (Java, JavaScript, HTML)
 2. one software engineer (B.E. in CS) who has 4 years' experience in coding (Java, JavaScript, HTML, DHTML)
 3. one software engineer (B.E. in EE) who has 3 years' experience in testing
 4. one software engineer (B.E. in CS) who has 4 years' experience in testing
- **Result:** Compared with the two standalone pages *show* and *update*, the *show+update* page needs much more time in programming and testing, as shown in Table 7 (the same developing engineers and quality/testing engineers). The developing engineers had to write additional JavaScript (about 250 lines) to deal with the DOM and hidden values programmatically, and also to face the fact that XML syntax is far more restrictive than HTML. Moreover, browsers' debugging support is still very poor in contrast with server-side debugging support. Therefore, although the sum of pages is reduced, the cost of programming and testing becomes higher.

6.3 Testing Summary of Experimental Results

Client-side rendering and model-view separation also facilitate the division of labor at testing. Therefore, testing of

pagelet entails lower and simpler testing environments. Except test data (XML documents) and test cases, server side is not necessary in page presentation and communications between pagelets. It is the biggest difference in the terms of development cost between client-side rendering and general server-side rendering, and brings more technical independence and flexibility to testing engineers.

There are three characteristics distinguishing pagelet-based applications from traditional Web applications in testing: event driven, communications between pagelets and DOM/XML manipulation. Mutation analysis [40] based on component [49] and automated test data generation [2] is suitable for pagelet oriented testing. Each developed pagelet is tested individually. Apparently, pagelets may be developed separately, and even executed on different servers. Therefore, integration testing [15] are necessary to ensure that communications between pagelets are correct. The characteristics of pagelet-based application, such as heterogeneity, reusability lead to different types of interoperability problems. Different pagelets can be built under various infrastructures, which could cause incompatibility problems. Most interaction-related faults are probably to be overlooked during unit-level testing by traditional integration testing methodologies, such as function decomposition-based integration testing and call graph-based integration testing.

6.4 Threats to Validity

Our pagelet technique is implemented in real industrial projects for an exploratory practice, and the findings basically reflect the benefits and costs of using pagelets. However, there are still several threats to validity. First, the statistics are collected from a small size project with several engineers involved. A further more solid empirical study should involve more software engineers with different skill levels. Second, the experiments are currently conducted on information systems. To address the problem of generality of the findings, different types of systems such as data centric systems and rendering centric systems need to be investigated. Last, the current experiments only compared XML-XSLT based pagelet rendering with JSP based rendering. A further comparison of developmental costs between our pagelet technique and other advanced rendering techniques (except sole JavaScript, CSS) will shed light on the practicability of our pagelet technique, and complete the comparison in Table 1.

7. Discussion

In this section, we discuss about the design philosophy of

our client-side rendering mechanism. Besides, we also envision the possible challenges and opportunities in information extraction of Web content for which our proposed rendering mechanism is used.

7.1 Design Philosophy

The design philosophy behind our approach/opinion aims at achieving the following goals.

1. To extricate the server side from the overloaded rendering work requested simultaneously by multiple client ends.
2. To further separate the concerns of Web applications (e.g., the separation of presentation from the business logics) and distinguish the handling of various types of data.
3. To facilitate the personalization and customization of the displayed output webpages for different users and types of client ends.

For the goal 1, in the client-server model, there is one common controversy about the choice of the thin-client architecture or the thin-server (full-client[29]) architecture. Yang et al. [62] found that thin-client systems can perform well on Web and multimedia applications in LAN (Local Area Network) environments. This shows that the rendering mechanism of the traditional Web application worked fine when it was usually at the server side. However, nowadays with the better and better graphical quality displayed by Web applications and more various requirements from users, the rendering tasks are becoming more and more computationally costly. Thus, keeping the rendering mechanism still at server side becomes an obsolete design. Now there is a trend that TSA (Thin Server Architecture) is advocated recently [12], [52]. Note that in our approach, we are not at the extreme to have most functions of Web applications exclusively limited at the server side or at the client side. The idea of our approach is not to use server-side templates to create and transmit the webpage, but to separate concerns using protocol between client and server, in such way to get a much more efficient and less costly development model. Using TSA leads to the following three potential advantages.

- The server side can only focus on the business logics.
- The client side can focus on the presentations.
- The communications between server and clients just exchange the raw data such as XML.

As shown in Table 4, the exchanged data between server and client is reduced in our approach, which improves the usage of data caching. Besides, moving the rendering from server to client also facilitates the concurrency of presentation. The XSLT used in our approach, as a functional programming language, can easily enable concurrency, since the order of execution of side-effect free functions in functional programming is not important. Thus, the client can run the multiple instances of the same XSLT thread simultaneously to speed up the data presentation.

For the goal 2, after separation of presentation and content, for the ease of reusability, maintainability and extensibility, it is desired to further separate the concerns of the Web application on multiple other dimensions [50]. The accessibility is one of the dimensions we try to separate concerns on. By adopting the RIAs or rich defined XML, the client side can provide navigable structure and guarantee of accessibility of controls (see Sect. 4). Another dimension of concern important to the performance is the different types of content. For example, the static and dynamic content in the different segments in the output webpages should be differentiated. In the original server-side rendering mechanism, the data exchanged between server and client is usually webpages. In our approach, with the skeletal segments generated by XML, XSLT and JavaScript, the exchanged data between server and client can be narrowed down to the segment level. Additionally distinguishing the static and dynamic segments also improves the efficiency of data caching (see Sect. 4). Besides the different handling for static and dynamic content, the similar idea can be applied to yet some other types of content.

For the goal 3, as the user-adaptive and context-aware characteristic is the trend for the future Web applications [10], Web application should have different rendering strategies for the different types of client ends towards various user requirements. Layout arrangement and visibility control (see Sect. 3.3) are just one type of user customization and personalization. In some more sophisticated client ends like smart-phone, rendering in accordance with the gesture capturing or eye tracking of users' response on the screen is better to be handled by the client end, otherwise waiting for the rendering results from server-side will cause the extra delay.

Certainly, the three goals of client-side rendering mentioned above do not come without any compromise. As shown in Table 7 and Table 6, using our approach together with the involved techniques may introduce the extra learning curve and initial investment for the early separations than the commonly used server-side rendering mechanism (taking almost more than double time, see Table 7). However, in the long term, the developers will benefit from this in the evolution and extension of the Web applications.

7.2 Opportunities and Challenges

The potential new opportunities may be brought to the further usage of Web applications by the pagelet structure. Information extraction on the Web focuses on the acquisition of information/knowledge from Web applications mainly. Many methods are proposed to extract the target information from webpages excluding the HTML tags by analysis of HTML sources. These methods evolving from machine learning or pattern recognition or XPath designation are costly in maintenance over a long period of time. In case that RDF-like description is given for dynamic-functionality-semantic in the container files, the information extraction from Web applications becomes convenient and the XML

data of dynamic pagelet is a clean extraction result. Also, the mashup of Web applications becomes much systematic. The reuse of Web application data, computation and UI becomes more efficient and thorough. It is easier to transform the pagelets into mashup components than general Web applications [6] and portlets [7], especially for typical users with no or little programming experience. Moreover, it may accelerate/realize such technical development tasks below:

- XML-based message exchangeability between the desktop computers and mobile devices
- Client-side template customization by end-users
- XML-based data transfer (layer) such as Ajax+JSON architecture
- Active webpages containing Microsoft Object Library or Flex/flash
- Towards efficient retrieval and update of client-side XML data in XML/XML-enabled database (e.g., XPath, XQuery operation) or by object-relational mapping technique
- Multilingual Web UI

Client-side XML transformation also poses new challenges to search engines because the dynamic pagelets are not seen as general HTML sources. The container files do not contain information presented on webpages itself. An even more devastating argument is that pagelets cannot be semantically crawlable [18] since the crawler robots fail to process JavaScript to parse the pagelets to acquire the information. Robots see only a mixture of HTML plus other code, which robots cannot know how to interpret. Moreover, the client transformation costs time and leads to an unavoidable short delay in page presentation at client-side Web browsers.

8. Conclusion and Future Work

We gave a comprehensive analysis of the advantages and disadvantages of a client-side rendering mechanism in different viewpoints of practice and experience. Our proposed framework can bring diverse new opportunities for the extensible reuse of Web applications, which has been proved by our experimental evaluation.

There is still plenty of work to be done before client rendering mechanism becomes a more mature technology, since it currently cannot attain a solution satisfactory to both users and developers. The dynamic visual effects are realized by client script functions, which bring the flexible operation but increase the development costs for the developers or development scenarios. Thus, a proper architecture is desired to attain the balance between developmental/maintenance difficulties for developers and flexibility of rendering output for users.

As future work, we will explore further the problems of system security, scalability, and server-side database update of client-side rendering mechanism. The current architecture in our ongoing implementation is proposed without careful consideration of security and fine-grained access

control, such as the malicious modifications that may occur in client-side XML accessing. In addition to the ongoing developing JavaScript libraries and frameworks, we will develop more various supporting technologies for efficient client rendering in future.

Acknowledgements

We gratefully acknowledge the technological support provided by Achievo Japan. We also acknowledge the advice from Takehiro Tokuda (Tokyo Institute of Technology, Japan), Hirofumi Yokouchi (Gunma University, Japan), Chuanqi Tao (San Jose State University, USA) and Bo Liu (Fuji Xerox, Japan). This work was supported by a Grant-in-Aid for Scientific Research A (No.22240007) from the Japan Society for the Promotion of Science (JSPS).

References

- [1] Ajax Pages, <http://ajax-pages.sourceforge.net/>
- [2] S. Artzi, J. Dolby, S.H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of javascript Web applications," Proc. 33rd International Conference on Software Engineering, pp.571–580, 2011.
- [3] Closure, <http://code.google.com/closure/>
- [4] Cross-Origin Resource Sharing, <http://www.w3.org/TR/cors/>
- [5] CSS Tutorial, <http://www.w3schools.com/css/>
- [6] F. Daniel and M. Matera, "Turning Web applications into mashup components: Issues, models, and solutions," Proc. 9th International Conference on Web Engineering, pp.45–60, 2009.
- [7] O. Diaz and J.J. Rodriguez, "Portlets as Web components: An introduction," J. Universal Computer Science, vol.10, no.4, pp.454–472, 2004.
- [8] Django, <https://www.djangoproject.com/>
- [9] Dojo, <http://dojotoolkit.org/>
- [10] P. Dolog, "Designing adaptive Web applications," Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science, pp.23–33, 2008.
- [11] Dust, <http://akdubya.github.com/dustjs/>
- [12] Extreme Scale: Thin Server Architecture, <http://www.slideshare.net/spacemonkeylabs/thin-server-architecture>
- [13] N. Fernandes, D. Costa, S. Neves, C. Duarte, and L. Carrico, "Evaluating the accessibility of rich internet applications," Proc. 9th International Cross-Disciplinary Conference on Web Accessibility, no.13, 2012.
- [14] Fiz, <http://fiz.stanford.edu/home/home>
- [15] J.Z. Gao, H.-S.J. Tsao, and Y. Wu, "Testing and Quality Assurance for Component-Based Software," pp.193–196. Artech House Computing Library, 2003.
- [16] P. Gao, H. Han, and T. Tokuda, "IAAS: An integrity assurance service for Web page via a fragile watermarking chain module," Proc. 6th International Conference on Ubiquitous Information Management and Communication, p.83, 2012.
- [17] F.J. Garcia, R.I. Castaneda, and A.A.J. Fuente, "A double-model approach to achieve effective model-view separation in template based Web applications," Proc. 7th International Conference on Web Engineering, pp.442–456, 2007.
- [18] F.J. Garcia-Izquierdo and R. Izquierdo, "Is the browser the side for templating?," IEEE Internet Computing, vol.16, no.1, pp.61–68, 2012.
- [19] J. Guo, H. Han, and T. Tokuda, "Towards flexible mashup of Web applications based on information extraction and transfer," Proc. 11th International Conference on Web Information Systems Engineering, pp.602–615, 2010.
- [20] Haml, <http://haml-lang.com/>

- [21] F. Hartmann, "An architecture for an xml-template engine enabling safe authoring," Proc. 17th International Workshop on Database and Expert Systems Applications, pp.502–507, 2006.
- [22] S.-W. Hsiao, Y.S. Sun, F.-C. Ao, and M.C. Chen, "A secure proxy-based cross-domain communication for Web mashups," Proc. IEEE 9th European Conference on Web Services, pp.57–64, 2011.
- [23] HTML5, <http://www.w3.org/TR/html5/>
- [24] IBM Accessibility, <http://www-03.ibm.com/able/>
- [25] Jade, <http://jade-lang.com/>
- [26] Java Portlet, <http://www.jsp.org/en/jst/detail?id=286>
- [27] JavaScript Templates, <http://code.google.com/p/trimpath/wiki/JavaScriptTemplates>
- [28] Jemplate, <http://search.cpan.org/dist/Jemplate/>
- [29] J. Jing, A.S. Helal, and A. Elmagarmid, "Client-server computing in mobile environments," ACM Computing Surveys, vol.31, no.2, pp.117–157, 1999.
- [30] jQuery, <http://jquery.com/>
- [31] JSON Markup Language, <http://www.jsonml.org/>
- [32] JsonFx.NET, <http://www.jsonfx.net/>
- [33] P. Li and E. Wohlstadt, "Script InSight: Using models to explore JavaScript code from the browser view," Proc. 9th International Conference on Web Engineering, pp.260–274, 2009.
- [34] MAMA: Key findings, <http://dev.opera.com/articles/view/mama-key-findings/>
- [35] J.K. Martinsen and H. Grah, "A methodology for evaluating JavaScript execution behavior in interactive Web applications," Proc. 9th ACS/IEEE International Conference on Computer Systems and Applications, pp.241–248, 2011.
- [36] modconcat, <http://code.google.com/p/modconcat/>
- [37] L. Moreno, P. Martinez, B. Ruiz, and A. Iglesias, "Toward an equal opportunity web: Applications, standards, and tools that increase accessibility," Computer, vol.44, no.5, pp.18–26, 2011.
- [38] Mustache, <http://mustache.github.com>
- [39] MXML, <https://learn.adobe.com/wiki/display/Flex/MXML>
- [40] K. Nishiura, Y. Maezawa, H. Washizaki, and S. Honiden, "Mutation analysis for javascript Web application testing," Proc. 24th International Conference on Software Engineering and Knowledge Engineering, pp.159–165, 2013.
- [41] node.js, <http://nodejs.org/>
- [42] S. Oney and B. Myers, "Firecrystal: Understanding interactive behaviors in dynamic Web pages," Proc. 2009 IEEE Symposium on Visual Languages and Human-Centric Computing, pp.105–108, 2009.
- [43] PageSpeed Insights, <http://developers.google.com/speed/>
- [44] T.J. Parr, "Enforcing strict model-view separation in template engines," Proc. 13th International Conference on World Wide Web, pp.224–233, 2004.
- [45] Prototype, <http://www.prototypejs.org/>
- [46] QueryTemplates, <http://code.google.com/p/querytemplates/>
- [47] T. Shintani and T. Ozono, "An introduction to mashup programming for web intelligence," Inf. Proc., vol.50, no.5, pp.444–453, 2009 (in Japanese).
- [48] M. Taguchi, K. Jamroendarasame, K. Asami, and T. Tokuda, "Comparison of two approaches for automatic construction of Web applications: Annotation approach and diagram approach," Proc. 4th International Conference on Web Engineering, pp.230–243, 2004.
- [49] C. Tao, B. Li, and J. Gao, "A systematic state-based approach to regression testing of component software," J. Software, vol.8, no.3, pp.560–571, 2013.
- [50] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns," Proc. 21st International Conference on Software Engineering, pp.107–119, 1999.
- [51] M. Tatsubori and T. Suzumura, "HTML templates that fly: A template engine approach to automated offloading from server to client," Proc. 18th International Conference on World Wide Web, pp.951–960, 2009.
- [52] Thin Server Architecture, <http://www.thinserverarchitecture.com/>
- [53] underscore.js, <http://documentcloud.github.com/underscore/>
- [54] Upgrade the UI to an XML Layout, <http://developer.android.com/guide/tutorials/hello-world.html>
- [55] J. Wang, H. Chen, and Y. Zhang, "Mining user behavior pattern in mashup community," Proc. 10th IEEE International Conference on Information Reuse and Integration, pp.126–131, 2009.
- [56] Web Modeling Language, <http://www.webml.org/>
- [57] Web Services for Remote Portlets, www.oasis-open.org/committees/wsrp/
- [58] Window.postMessage, <https://developer.mozilla.org/en-US/docs/Web/API/window.postMessage>
- [59] Windows Performance Analysis Developer Center, <http://msdn.microsoft.com/en-us/performance>
- [60] XAML Overview, <http://msdn.microsoft.com/en-us/library/ms752059.aspx>
- [61] XSL Transformations, <http://www.w3.org/TR/xslt20/>
- [62] S.J. Yang, J. Nieh, M. Selsky, and N. Tiwari, "The performance of remote display mechanisms for thin-client computing," Proc. General Track of the Annual Conference on USENIX Annual Technical Conference, pp.131–146, 2002.
- [63] U. Zdun, "Dynamically generating Web application fragments from page templates," Proc. 2002 ACM Symposium on Applied Computing, pp.1113–1120, 2002.

Hao Han received a Ph.D. in computer science from Tokyo Institute of Technology in 2009. He is currently a research associate in the Department of Information Science, Kanagawa University, Japan. His research focuses on Web engineering. He is a member of IEEE CS, ACM, JSSST, DBSJ, IEICE, IPSJ.

Yinxing Xue was born in 1982. He received Ph.D. from National University of Singapore in 2012. He is now a research fellow in Information Security Group, Temasek Laboratories, National University of Singapore. His research interest includes software analysis, reverse engineering and Web engineering.

Keizo Oyama received the B.E., M.E. and Dr.Eng. from the University of Tokyo in 1980, 1982 and 1985, respectively. He is a professor of National Institute of Informatics (NII), Research Organization of Information and System, and the School of Multidisciplinary Sciences, the Graduate University of Advanced Studies (SO-KENDAI). His research interests are Web information access, information retrieval systems, and full-text search technologies. He is a member of IEICE, IPSJ, JSIMS and DBSJ.

Yang Liu was born in 1981. He received Ph.D. from National University of Singapore in 2010. He is now an assistant professor in School of Computer Engineering, Nanyang Technological University. His research interest includes formal methods, cyber security and software engineering.