SVC: Selector-based View Composition for Web Frameworks

William P. Zeller and Edward W. Felten
{wzeller,felten}@cs.princeton.edu
Princeton University

Abstract

We present *Selector-based View Composition* (SVC), a new programming style for web application development. Using SVC, a developer defines a web page as a series of transformations on an initial state. Each transformation consists of a selector (used to select parts of the page) and an action (used to modify content matched by the selector). SVC applies these transformations on either the client or the server to generate the complete web page.

Developers gain two advantages by programming in this style. First, SVC can automatically add Ajax support to sites, allowing a developer to write interactive web applications without writing any JavaScript. Second, the developer can reason about the structure of the page and write code to exploit that structure, increasing the power and reducing the complexity of code that manipulates the page's content.

We introduce SVC as a stateless, framework-agnostic development style. We also describe the design, implementation and evaluation of a prototype, consisting of a PHP extension using the WebKit browser engine [37] and a plugin to the popular PHP MVC framework Code Igniter [8]. To illustrate the general usefulness of SVC, we describe the implementation of three example applications consisting of common Ajax patterns. Finally, we describe the implementation of three post-processing filters and compare them to currently existing solutions.

1 Introduction

The growth of Ajax has resulted in increased interactivity on the web but imposes additional developmental costs on web developers. Many browsers do not support JavaScript (and therefore Ajax), including search engine crawlers, older browsers, browsers with JavaScript (JS) disabled, and screen readers. These browsers will not be compatible with portions of a site which exclusively use Ajax.

The standard "best practice" when creating a site that supports both JS and non-JS browsers is to use a technique called *progressive enhancement* [5]. A developer first creates a site that works in non-JS browsers and then uses JavaScript to add interactivity. For example, a page might include a link titled "Click here for more" which loads a new page when clicked. Progressive enhancement might involve modifying that link to load and insert additional content in-line using Ajax, without navigating to a new page. Progressively enhanced sites work in both JS and non-JS browsers, but require the developer to duplicate much of the site's functionality in order to respond appropriately to both Ajax and non-Ajax requests. (We use the term *non-Ajax request* to refer to a request that loads an entirely new page and *Ajax request* to refer to a request made by an existing page which completes without leaving the current page).

Alternatively, a developer may choose to only support JS browsers. With this approach, a developer directly implements functionality using JavaScript or uses a framework such as Google Web Toolkit (GWT) [20], which provides server-side support for generating client-side JavaScript, but provides no easy way of supporting non-JS browsers.

Currently, developers face a choice. They can either support both JS and non-JS browsers by duplicating much of a site's functionality or risk preventing certain users and search engines from accessing and indexing their sites.

We believe this tradeoff is unnecessary. We present a new programming style, SVC, which allows a site to be constructed in a way that can support Ajax and non-Ajax requests automatically, providing both interactivity and backwards compatibility.

1.1 Background: MVC

The model/view/controller (MVC) architectural pattern is commonly used in web frameworks (e.g., Django, Ruby on Rails, Code Igniter, Struts, etc. [11, 29, 8, 34]). In web MVC (which differs somewhat from traditional MVC), a model encapsulates application data in a way that is independent of how that data is rendered by the application. The view accepts some number of models as input and transforms them into appropriate output that will be sent to the browser. The controller connects the models and views, typically by bundling up model data and sending it to the view for rendering.

The view accepts data as input and produces a string as output. This output may include information about its type (e.g., HTML, XML, or JSON [23]), but is otherwise treated as an opaque string by the framework. After being manipulated by some number of post-processing filters, the string is sent directly to the browser. Because the view's output is treated as an opaque string, it is difficult for the framework to reason about the structure of the content. These views are complicated by the need to provide both Ajax and non-Ajax versions of a site.

1.2 Our Approach: SVC

SVC is a programming style that changes how views construct their output. Instead of returning an opaque string, a view describes the output as a sequence of transformations. Each transformation consists of a selector, used to query the document, and an action, used to modify the DOM [2] nodes matched by the selector. The framework, which previously operated on an opaque string, now has knowledge of both the structure of the page as well as how the page was composed. SVC expresses page content in a manner that is succinct, powerful, and portable.

A key benefit of SVC is that the framework can choose whether to apply the transformations on the server or on the client (where possible). When called on to respond to an Ajax request, it returns a list of transformations needed to convert the current page into the new page, using client-side JS. The use of selectors allows both client- and server-side code to convert the list of transformations into the same complete document, allowing SVC to provide automatic Ajax support and automatic progressive enhancement to pages written in this style. This benefit relies on the portability (from server to client) of SVC's transformation rules.

SVC does not attempt to replace existing template systems. Instead, a developer *composes* different *views* (which may be the output of a template). The developer describes where the composition should occur using *selectors* (described in Section 3.1.1). Developers may continue to use any template language with SVC.

Additionally, SVC does not interfere with existing code. Both JavaScript and existing controllers not using SVC will continue to work without modification. This allows SVC to be added to an existing site and used only when necessary (without the need to immediately refactor legacy code).

Finally, SVC is able to use its knowledge of a page's structure to provide developers with a succinct and familiar post-processing mechanism. This allows developers to write code to filter a page based on its content without forcing them to parse the page themselves.

1.3 Contributions and Organization

This paper makes the following contributions:

- §2 We describe the architecture of SVC and discuss how it differs from the traditional MVC model.
- §3 We describe the server-side and client-side components that make up SVC and discuss our design decisions.
- §4 We describe the implementation of an SVC prototype. The prototype consists of a PHP extension written in C++ using the WebKit engine [37], a PHP plugin for the MVC framework Code Igniter [8], and a client-side JavaScript plugin that handles Ajax responses.
- §5 We present a complete minimal example of a site implemented with and without SVC to show how our approach differs. We also briefly describe a number of example sites and filters we implemented using SVC.

We evaluate the performance of our implementation in Section 6, before discussing related work and concluding.

2 Architecture

SVC extends the model/view/controller (MVC) pattern to manage responsibilities currently handled by the developer. Fig. 1(a) shows how a request travels through a traditional MVC application which supports both Ajax and non-Ajax requests. (The "model" in MVC is irrelevant to this discussion and omitted.)

The request is sent to a controller which calls the appropriate view depending on the type of request. If the request is a non-Ajax request, the non-Ajax view is called. This view outputs the HTML document which is rendered by the browser.

In the case of an Ajax request, the controller calls the Ajax view which outputs data in a format that can be read

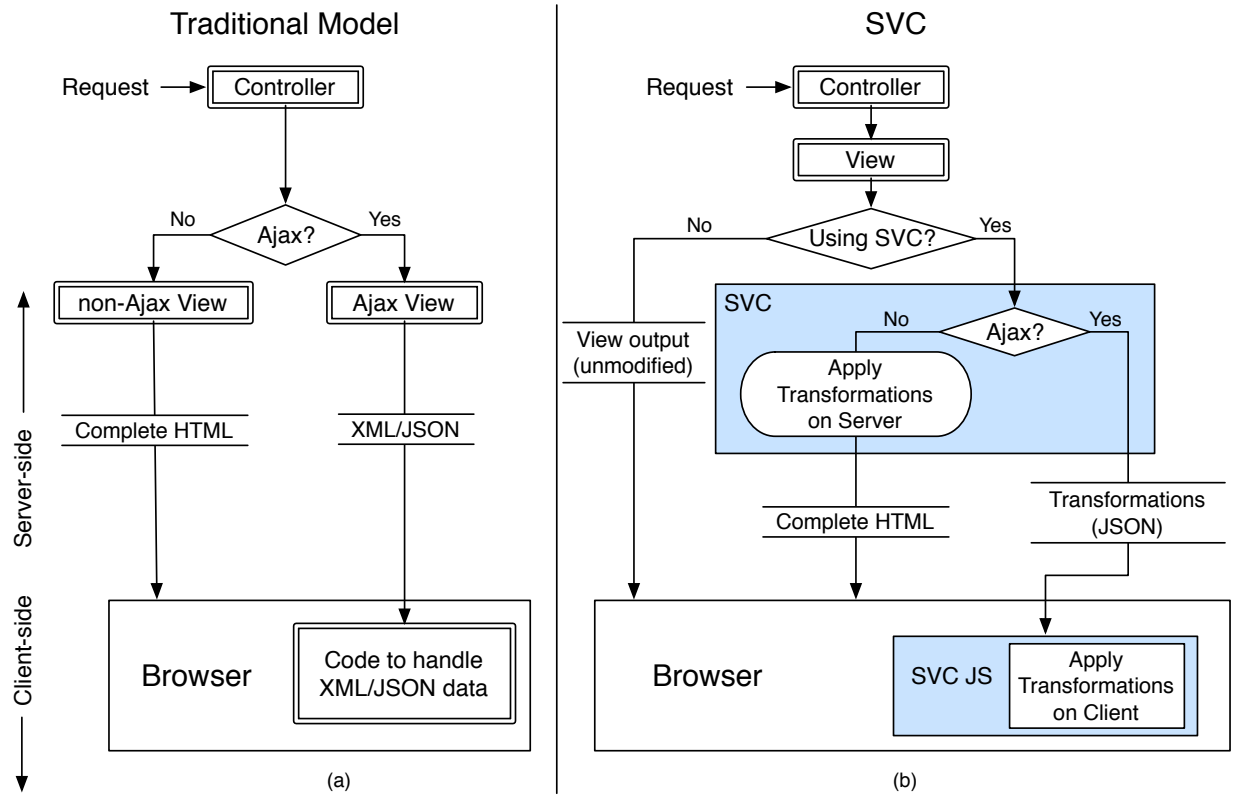

Figure 1: Architecture of SVC. Boxes surrounded by a double line represent components that are the developer's responsibility. Fig. 1(a) shows the traditional MVC architecture (with models omitted) for sites supporting both Ajax and non-Ajax requests. A request is sent to the controller which loads either an Ajax or non-Ajax view. The developer must write both views, as well as client-side code which handles the output of her Ajax view. Fig. 1(b) shows the SVC model. A request is sent to a single view, which can decide to use SVC or not use SVC. The view uses SVC by sending it a transformation list. SVC responds to a non-Ajax request by applying the transformations on the server and returning a complete HTML page. SVC responds to an Ajax request by simply returning the transformation list as JSON. The transformation list is applied by the client-side SVC JS to the current document in the browser. In Fig. 1(b), SVC allows the developer to write only one view and no client-side code.

by client-side code (typically XML or JSON). While this output may contain HTML snippets, custom client-side JS is required to insert the snippets in the page at the appropriate positions. Both views must be created by the developer, who must duplicate functionality to provide both Ajax and non-Ajax output. In addition, the developer needs to write client-side code to handle the output created by her Ajax view. Because the framework knows nothing about the view's structure, it cannot assist in this process.

SVC's extension to MVC is shown in Fig. 1(b). The request is sent to a controller which now calls only one view regardless of the type of request. Instead of outputting HTML for a non-Ajax request and XML/JSON for an Ajax request, the developer describes the page as a series of transformation rules as well as any items on which the transformations depend. The transformation list is sent to the SVC, which decides how to act based on the type of request.

In the case of a non-Ajax request, SVC applies the transformation list on the server-side, creating a complete HTML document on the server. This document is then sent to the browser.

In the case of an Ajax request, SVC converts the transformation list to a form readable by client-side code. This serialized list is sent to the client where the transformations are applied directly to the current document in the browser. As SVC includes all the client-side code necessary to apply the transformations to the document, the developer does not need to write any client-side JS when creating an Ajax-compatible site.

Using the architecture in Fig. 1, SVC is able to automatically generate both Ajax and non-Ajax versions of a site. In addition, SVC needs to progressively enhance the site and inject a script into all pages to handle client-side actions. Specifically, SVC adds a `CLASS` attribute

to each element needing to be enhanced and inserts a `SCRIPT` tag into the head of the document which points to the client-side SVC JS. This script adds a *click* event handler to each element with the above class and also manages requesting and applying the transformation list. Progressive enhancement and script injection are made possible by a post-processing filtering system that SVC uses internally and exposes to developers.

SVC is stateless–it operates only on the transformation list provided by the view. It does not store HTML or output from previous requests, nor does it depend on the state of the browser (i.e., which requests have previously occurred).

SVC consists of a server-side API, server-side code to handle both Ajax and non-Ajax requests, client-side code to handle the output of Ajax requests, and a filtering module, both used internally and made available to developers.

3 Design

3.1 Server-side Components

Developers use the SVC server-side API to describe how a page should be constructed.

3.1.1 Selectors and Actions

SVC asks a developer to construct a page by defining a list of transformation rules. These rules consist of a selector, used to locate one or more nodes in the DOM, and an action, used to modify nodes matched by the selector. By defining a page as a series of transformation rules, SVC is able to respond to both Ajax and non-Ajax requests appropriately. SVC is able to decide whether to send the list of transformations to the client for conversion using JS or to convert the list on the server (by applying each transformation, in order) and return a full HTML document to the browser.

We consider next how a transformation rule is expressed: how the web programmer specifies the *selection* of portions of a page, and how the programmer expresses which *action* to take on the selected portions.

Selectors Selectors provide a way to query HTML documents. Selectors were introduced as part of the CSS1 Specification [24], where they were used to identify the DOM nodes to which styling rules would be applied. Table 1 shows a few examples of selectors.

The selector syntax is simple and includes shortcuts that make HTML queries more succinct. Additionally, developers have grown accustomed to selectors due to their ubiquity in CSS. These benefits resulted in a number of JS libraries adopting selectors as a query language, including Closure, Dojo, jQuery, MooTools, Prototype, and YUI [7, 13, 22, 25, 26, 39]. Initially, JS libraries were forced to re-implement much of the selector syntax due to a lack of browser support, but recent proposals (i.e, the Selectors API [36]) have led to increased support in browsers.

Selector Examples

Selector	Description
`*`	All elements
`#foo`	Elements with id `foo`
`.bar`	Elements with class `bar`
`div`	All `div` elements
`div[f="b"]`	`div` elements with attribute `f` = `b`
`div > a`	`a` elements that are children of `div` elements

Table 1: A few examples of selectors. The complete selector syntax can be found in the W3C Selectors Level 3 Recommendation [4].

We chose selectors because they are expressive, succinct, familiar to developers, designed with HTML in mind, and supported by a growing number of browsers.

An alternative choice for a querying mechanism might be XPath [10] or a custom template language. XPath is more expressive than selectors but is more verbose and designed for XML, not HTML, so HTML shortcuts do not exist. Also, developers are less familiar with XPath, because it is not as widely used as selectors in front-end web development.

Another option might be to offer a custom template language. A custom template language would force developers to annotate their code with template instructions, which could conflict with existing template systems in use. A template language would also need to be implemented on both the client and server-side and would require developers to learn a new layout language. We chose not to take this approach, as selectors already meet our needs.

Actions The second part of a rule is an action. Actions define the set of operations a developer can perform on a document during composition. We modeled our API after the jQuery manipulation functions. We consider the jQuery manipulation functions a reasonable approximation of the actions needed by developers. Table 2 shows the actions our SVC implementation makes available.

SVC allows developers to define additional actions. These actions only need to be written once and could be distributed with SVC or as a set of plugins. Creat-

Examples of Actions

Action	Result
`addClass(s,c)`	Add class `c`
`append(s,el)`	Append `el` inside
`attr(s,a,b)`	Set attribute `a` to `b`
`css(s,a,b)`	Set css property `a` to `b`
`html(s,el)`	Set inner HTML to `el`
`prepend(s,el)`	Prepend `el` inside
`remove(s)`	Remove from DOM
`removeClass(s,c)`	Remove class `c`
`text(s,t)`	Set inner text value to `t`

Table 2: Actions supported by our SVC prototype. Each action is passed a selector `s` as the first argument. The result of each action is performed on all nodes matching the selector. More actions could be added to an SVC implementation, with the only requirement that they be implemented in both server and client-side code.

ing a new action consists of writing client- and server-side code that implements the action on the client and server, respectively. Since the only requirement for an action is that it can be executed on the server and client, actions could manipulate the DOM or provide unrelated functionality. For example, a `location` action could be written to redirect a page, which would set an HTTP `Location` header on the server and use `window.location` to redirect on the client.

3.1.2 Example

To illustrate how a developer would use selectors and actions to modify a page, we show a few example transformations in Table 3. Our implementation of SVC provides a class called SVCList which represents a list of transformations. Each command consists of an action, which is a method that accepts a selector as the first argument along with any additional arguments needed by the action.

We now describe how SVC responds to a request using commands 1-5 from Table 3 (for brevity, we ignore commands 6-10).

If SVC receives a non-Ajax request, it needs to construct the entire page and send it to the browser. SVC does this by taking the initial page (here, simply the string "`<a></a>`") and applying each transformation to it, in order. The result of applying the transformations can be seen in in the "Output" column of Table 3.

If SVC receives an Ajax request, it sends only the transformation list to the client. This list is encoded as JSON [23], so the actual data sent to the client is:

```
// action, selector, arguments
[["text",    ["a", "x"]],
 ["append",  ["a", "<b>y</b>"]],
 ["html",    ["b", "t <s>u</s> u"]],
 ["prepend", ["b", "<i>z</i>"]]]
```

This JSON list of transformations is applied to the current document in the browser by the SVC client-side JS.

3.1.3 Initial Content

SVC responds to an Ajax request with a list of transformations that operate on an existing document (residing in the browser). However, when a non-Ajax request is made, SVC must respond with a complete page. One option would be to always generate a complete page and discard the unnecessary portion when responding to Ajax requests. This would require generating the complete initial page for each request. Instead, we provide the method `initial`, which allows a developer to define the initial page on which to apply transformations. If the argument to this method is the name of a controller, that controller is called. If the argument is a text string, that text is used as the initial page.

```
function page() {
  $this->svc->initial('base');
  $this->svc->text('title', 'Page title');
  $this->svc->html('body', '<b>The body<b/>');
}
```

When an Ajax request is made to `page`, the controller `base` does not run and only the list of transformations (`text` and `html`) is returned to the client. When a non-Ajax request is made, SVC (on the server side) applies the transformation list to the output of the `base` controller.

The separation of the initial page from the transformation list allows SVC to only run the necessary code in response to an Ajax request.

3.1.4 Progressive Enhancement

Once the developer has specified a list of dependencies and transformations, SVC is able to respond to both Ajax and non-Ajax requests correctly. Remember, however, that the developer has written the site without Ajax in mind, so no links on the site cause an Ajax request to occur. SVC needs to progressively enhance pages to use Ajax where appropriate. The developer is in no position to do this herself, since she does not know (and should not need to know) the internals of our client-side library.

Links should only be enhanced to use Ajax if an Ajax request makes sense in the context of the current page. If no selector in the transformation list matches an element on the page, the transformation list will have no

Using Selectors and Actions to Modify HTML

	Command	Output of `$s->toHTML()`
1	`$s = new SVCList('<a></a>');`	`<a></a>`
2	`$s->text('a', 'x');`	`<a>x</a>`
3	`$s->append('a', '<b>y</b>');`	`<a>x<b>y</b></a>`
4	`$s->html('b', 't <s>u</s> u');`	`<a>x<b>t <s>u</s> u</b></a>`
5	`$s->prepend('b', '<i>z</i>');`	`<a>x<b><i>z</i>t <s>u</s> u</b></a>`
6	`$s->remove('i, s');`	`<a>x<b>t u</b></a>`
7	`$s->css('a', 'color', 'red');`	`<a style="color:red;">x<b>t u</b></a>`
8	`$s->attr('[style]', 'style', '');`	`<a style="">x<b>t u</b></a>`
9	`$s->addClass('b', 'c1');`	`<a style="">x<b class="c1">t u</b></a>`
10	`$s->remove('.c1');`	`<a style="">x</a>`

Table 3: Each command is run in order, from top to bottom. The output (on the right) shows the HTML output if the SVCList were converted to a complete HTML document at each step.

effect. However, a developer may define transformations that may happen (if the selector matches) but do not need to happen. These will effectively be ignored when the transformations are applied on the client.

Our SVC implementation provides the method `rewrite(foo,bar)` which specifies that all links pointing to controller `foo` should be rewritten to use Ajax if the current controller or any dependencies of the current controller is `bar`. Providing `bar` is necessary because Ajax requests may only be appropriate in the context of a specific page. For example, loading a new tab using Ajax would only work if the element containing the tab exists in the current page.

3.1.5 Filters

SVC also provides *filters*, which allow the modification of the output of a view. Filters exist in other frameworks (e.g., Django Middleware [12]) and are used to rewrite pages in a variety of ways (see Sec. 5.2.2 for examples of filters). Our implementation of SVC provides the class SVCFilter to manage filter registration.

SVC uses filters internally to rewrite links on a page and to automatically inject the SVC client-side JS into the page. To illustrate the use of filters, we show how SVC injects its client-side code into the page.

```
function insert_svc_js($head) {
  $svc_js = '<script src="SVC.js"></script>';
  $head->append($svc_js);
}

// converts ...<head></head>... to:
// <head><script src="SVC.js"></script></head>
$svcfilter->register('head', 'insert_svc_js');
```

SVCFilter also supports action shortcuts, allowing this to be written more succinctly:

```
$svc_js = '<script src="SVC.js"></script>';
$svcfilter->append('head', $svc_js);
```

Filters are similar to view transformations with two important differences. First, filters are always applied on the server-side and never on the client. This is necessary in certain situations, such as in the above script example. It is also necessary if the developer wants to prevent data from ever appearing on the client site. For example, a transformation list would not be the appropriate place to sanitize user input, because the user input would be sent to the client as part of the transformation list. Writing code as a filter ensures that conversion happens on the server.

The second difference between filters and view transformations is that filters can run on pieces of a document before the pieces are converted into the full document. For example, take commands 3-5 from Table 3. In an Ajax request, the server returns a list of transformations. This list contains three snippets of HTML with may match a selector (i.e., "`<b>y</b>`", "`<s>u</s> u`", and "`<i>z</i>`"). Filters are run on each snippet independently, allowing post-processing to occur on Ajax output. In a non-Ajax request, filters are run on the complete document before being sent to the browser.

To allow filters to run on pieces of a document before they are converted to a full document, we permit only a limited subset of selectors to be used in filters. Specifically, we only allow *simple selectors*, which are defined by the W3C Selectors spec [4]. Simple selectors may only refer to specific nodes and not their position in a document. For example, "`title`" and "`a[href$=jpg]`" are examples of simple selectors, while "`div > a`" is not. Simple selectors are necessary because filters run on pieces of the document in response

to an Ajax request, independent of where they may later reside in the DOM. SVC is unable to determine where those pieces will exist in the complete document after they are composed on the client-side.

3.2 Client-side Components

The client-side component of SVC consists of a JS script. This script has two responsibilities. The first is to progressively enhance the page to make Ajax calls. The second is to load a transformation list from the server using Ajax and apply that transformation list to the current document in the browser. The client-side script must apply this transformation list in the same way it would be applied on the server. This ensures that the resulting document is the same, regardless of where the conversion occurred.

4 Implementation

The server-side API consists of four classes that are needed by compatible implementations.

SVCList SVCList represents a transformation list. This class provides action methods (append, addClass, etc), each of which pushes a new transformation to the end of the list. SVCList provides the method `toHTML` which applies each transformation in order and then returns the HTML representation of the final document. Also provided is `toJSON`, which serializes the transformation list as JSON.

SVCManager SVCManager provides two methods to developers. The first is `initial`, which accepts one or more controller names as arguments. The second is `rewrite(foo, bar)`, which rewrites all links to the controller `foo` when the controller `bar` has been executed.

SVCFilter SVCFilter allows the developer to register post-processing actions. These actions will run regardless of the controller (or view) called. SVCFilter provides the method `register(simple_selector, callback)`, which runs the callback function `callback` with an argument of each node matched by `simple_selector`. SVCFilter also provides action methods as a convenience, allowing simple filters to be easily created.

Snippet The Snippet class represents a parsed HTML fragment. Snippet objects are used internally by SVCList to actually perform transformations. Note that the site will only output the result of the parser used by Snippet, which means the developer is constrained to the HTML supported by the parser. Implementations of SVC should use liberal parsers. The Snippet class is typically hidden from developers by accepting strings as input to most action methods. For example, the following two function calls are equivalent:

```
$svclist->html('body', 'foo <b>bar</b>');
$svclist->html('body',
               new Snippet('foo <b>bar</b>'));
```

The Snippet class supports all action methods. These methods can accept a selector as the first argument, in which case the selector runs on the nodes in the snippet matched by the selector. Snippet action methods can also be called on the snippet itself, which runs the actions on all top-level nodes.

In addition to these classes, the implementation has the following responsibilities:

Script Injection Our SVC implementation injects its client-side JS into the current page using a filter which appends a `script` tag to the `head` element of the page.

Progressive Enhancement Progressive enhancement involves a step on the server and the client. On the server, each link that should be rewritten (checked by comparing `rewrite` calls to the current controller and dependency list) is annotated with a special `CLASS` attribute using a filter.

When the page loads, the SVC JS searches for any element having this special class and adds a `click` or `submit` event to this element. When this event occurs, the script uses Ajax to load a transformation list from the server.

Client-side Code The client-side component of SVC consists of a single JS script. This script progressively enhances the site, loads a transformation list from the server, and applies this list to the current document.

4.1 Implementation Details

We implemented a prototype of SVC as a plugin for the PHP MVC framework Code Igniter (1.7.2). We implemented the Snippet class as a PHP extension written in C++. We used the WebKit engine (used by the Safari [30] and Chrome [6] browsers) to parse HTML and WebKit's `querySelectorAll` function (defined as part of the Selectors API [36]) to perform selector matching. Specifically, we used WebCore (which is a component of WebKit), from the WebKitGTK+ 1.1.15.3 distribution [38]. WebKitGTK+ was chosen due to its compatibility with Ubuntu (no GUI code was used). SVCList, SVCFilter and SVCManager were written in PHP. We implemented Snippet actions in C++.

The client-side code consists of a 4.3/1.7KB (uncompressed/compressed with YUI Compressor [40]) JS file. This code uses jQuery (1.3.2) to handle actions and selectors. We chose jQuery because it supports the same CSS 3 Selector syntax supported by WebKit's `querySelectorAll`, and because many of our actions exist as functions in jQuery. The file size above does not include the size of jQuery, which is 19KB (compressed and Gzipped).

The client-side SVC library has been successfully tested in Internet Explorer (6,7,8), Safari (4), Chrome (3.0) and Firefox (3.5).

5 Examples

5.1 Complete Minimal Example

To make SVC more complete, we give a full example of a minimal site implemented with and without SVC in Figure 3.

This site consists of a page containing the text "A brief overview" and a link, titled "Show more", which loads more content when clicked. This link should load more content inline using Ajax when available but continue to work in non-JS browsers. Figure 3 (a)-(d) shows this site implemented without SVC. A controller contains the methods `index` and `more` which are executed when `/` and `/more` are called, respectively.

Both controllers pass an array consisting of a title, a boolean (whether to show more content) and an optional content string to the template (b). This template replaces the title with the passed value and either outputs a link pointing to additional content or the content itself. The `more` method responds to an Ajax request with a JSON version of the data array.

On the client, custom JS is required to interpret the JSON (c). Note that this JavaScript performs much the same function as the template (setting the title, inserting content, etc).

An SVC version of this site can be seen in Figure 3 (e)-(g). The template (f) consists of the initial page, which is loaded by the `index` method in (e). The method `more` defines three transformations which are either returned as JSON (in response to an Ajax request) or applied to the output of `index` method (in response to a non-Ajax request). Note that no custom JS is necessary because SVC JS will apply the transformations on the client-side automatically. Figure 3 (h) shows how SVC transforms the `index` page (the `more` page is omitted for brevity) by inserting its client-side script and added the class `svc_rewrite` to the appropriate links.

5.2 Additional Examples

5.2.1 Sites

We implemented three sites to illustrate the usefulness of SVC. We briefly describe these sites below.

Tabs We created a site consisting of a number of tabbed pages. The transformation list of each page changes the title of the document, sets the correct tab header (by setting a class), and sets the content of the page. The tab content is positioned inside of a tab-specific `div`, which allows us to rewrite links to use Ajax if that tab-specific `div` exists.

Status Feed We implemented a site to allow users of a fictional social network to update their status, which is combined on a single page. The transformation list consists of new status updates. These status updates are prepended to the current list of updates in response to an Ajax request or set as the current list of updates in response to a non-Ajax request.

Form Validation We implemented an example of form validation using SVC. Form validation is an Ajax pattern which must also be implemented in server-side code to prevent a malicious user from inserting invalid data into the application.

Each `input` element is given an associated `span` element to hold errors (e.g., Fig. 2(a)).

```
<input type="text" name="name"/>
  <span id="name_error"></span>
<input type="text" name="email"/>
  <span id="email_error"></span>
```

(a) Two inputs with associated error `spans`

```
$svclist->text('span#email_error',
               'Email error');
```

(b) Setting email error text

Figure 2: Setting an error message with SVC.

The form controller depends on a method which generates the form. The controller then sets the appropriate error message (e.g., Fig. 2(b)). If the form is submitted using Ajax, the error message will be updated in place. If it is submitted without Ajax, the error message will be properly set when the page reloads.

5.2.2 Filter Examples

We implemented three filters to illustrate the ease with which they allow developers to post-process pages. An example of filter code can be found in Appendix A.

Example not using SVC

(a) Controller

```
<?php
class Article extends Controller {

function index() {
    $data = array('title' => 'Brief Overview',
                  'showing_more' => FALSE);

    echo view('index.php', $data);
}

function more() {
    $data = array('title' => 'More Content',
        'content' => view('morecontent.php'),
        'showing_more' => TRUE);

    if (is_ajax()) {
        echo json_encode($data);
    } else {
        echo view('index.php', $data);
    }
}

}
```

(b) index.php

```
<html>
<head><title><?=$title?></title>
<script src="/js/jquery.js"></script>
<script src="/js/morecontent.js"></script>
</head><body>

A brief overview.
<? if (!$showing_more) { ?>
  <a href="/more" id="show_more">Show More</a>
  <div id="more_content"></div>
<? } else {?>
  <div id="more_content"><?=$content?></div>
<? } ?>

</body></html>
```

(c) morecontent.js

```
function more_content_clk(json) {
   $('a#show_more').remove();
   document.title = json.title;
   $('#more_content').html(json.content);
}

$(function() {

   $('#show_more').click(function() {
       $.getJSON('/more', more_content_clk);
       return false;
   });

});
```

(d) morecontent.php

```
Some <b>more</b> content
```

Example using SVC

(e) Controller

```
<?php
class Article extends Controller {

function __construct() {
    $this->svc->rewrite('more', 'index');
}

function index() {
    $this->svc->initial(view('index_svc.php'));
}

function more() {
    $this->svc->initial('index');
    $this->svc->text('title', 'More Content');
    $this->svc->remove('a#show_more');
    $this->svc->html('#more_content',
        view('morecontent.php'));
}

}
```

(f) index_svc.php

```
<html>
<head><title>Brief Overview</title></head>
<body>

A brief overview.

<a href="/more" id="show_more">Show More</a>

<div id="more_content"></div>

</body></html>
```

(g) morecontent.php

```
Some <b>more</b> content
```

SVC Output Example

(h) SVC generated index

```
<html>
<head>
<title>Brief Overview</title>
<script src="svc.js"></script>
</head>
<body>

A brief overview.

<a href="/more" id="show_more"
    class="svc_rewrite">Show More</a>

<div id="more_content"></div>

</body></html>
```

Figure 3: Two implementations of a web page containing the link "Show More". When clicked, additional content is loaded. Both Ajax and non-Ajax calls are supported. The left column ((a)-(d)) shows this page implemented without SVC. The right column ((e)-(g)) implements the page using SVC. The SVC output of the index page can be seen in (h). SVC has inserted its client-side script and added the class `svc_rewrite` to the appropriate links.

CoralCDN Filter CoralCDN [18] is a free peer-to-peer content distribution network. It provides an elegant URL structure that allows a resource to be turned into a "coralized" resource by simply appending `.nyud.net` to the hostname. Fig. 6 shows a complete SVC filter which rewrites all links and images in a page having the class "coralize". Fig. 7 shows the regular expressions used by a WordPress plugin [9] to perform the same search.

CSRF Form Filter Cross-Site Request Forgery vulnerabilities allows an adversary to perform actions on behalf of a user by maliciously submitting data to a trusted site [31]. A complete description of CSRF vulnerabilities is outside the scope of this paper, but a common defense against CSRF attacks is to insert a *CSRF token* into all `form` elements which will be submitted to the current site using the `POST` method. This token can be used by the server to verify that requests were sent by the user and not a malicious site. We wrote a filter that uses SVC to insert a token as a child of the appropriate `form` elements.

Script Coalescing Filter JavaScript can be compressed to reduce its size. A web service, called Reducisaurus [27], accepts URL arguments consisting of one or more JS files or JS code. Reducisaurus combines all scripts into a single, compressed file. We wrote a filter to convert all script tags to the appropriate URL arguments. The filter removes these scripts from the page and appends a link to the compressed JS file to the document's `head` element.

6 Performance Evaluation

Three aspects of SVC impose performance costs on the application. SVC needs to parse each snippet, run selectors, and perform actions. We evaluate these costs below and show the total performance cost of an example site. All tests were run on an desktop Dell XPS Dimension 9150 (Pentium D 2.8Ghz) machine with 2GB of RAM running Ubuntu 9.10 (Linux Kernel 2.6.31-16-generic). We used PHP 5.2.10 and Apache/2.2.12 (where applicable).

Parsing time To evaluate the parsing speed of our SVC implementation, we parsed the content of 10,000 popular websites according to Alexa [1]. The Alexa list of the most popular million sites was downloaded on 21-Nov-2009 and the first page of each site was downloaded on Nov 24, 2009. We parsed the content of the first 10,000 sites, skipping any site that did not return an HTTP 200 status. We compare our parsing speed to the speed of DOMDocument [14], which is an HTML parser built into PHP (we did not use DOMDocument for our implementation because it does not support selectors). DOMDocument does not parse JavaScript or CSS, while WebKit (used by our implementation) does. See Fig. 4 for the results of our parsing tests. The cost to parse a page is a cost not imposed on traditional PHP sites, which can output their content directly.

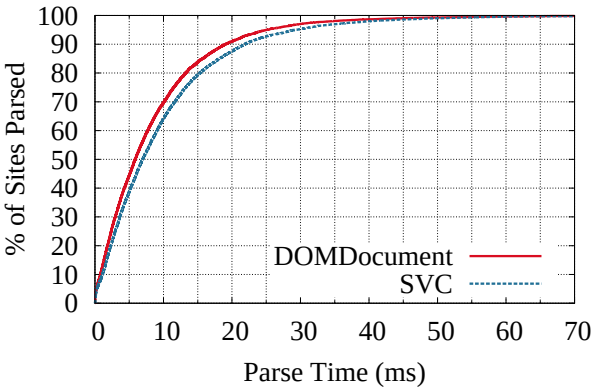

Figure 4: Parse time of 10,000 popular sites. SVC is compared to DOMDocument, an HTML parser that comes with PHP. This graph shows that, e.g., 80% of sites were parsed in 15ms.

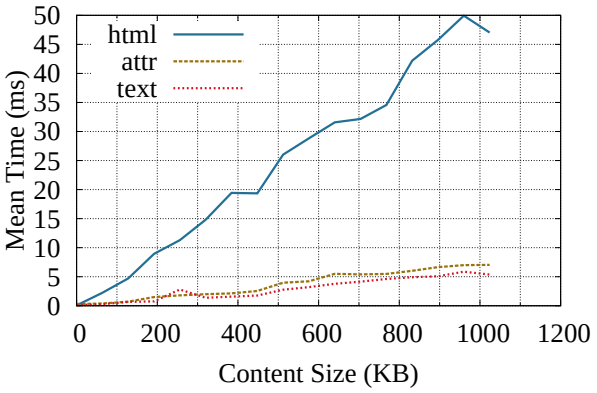

Figure 5: Mean DOM manipulation time as a function of content size. Each action (`html`, `attr`, and `text`) was called 100 times on each content size, which varied from 0 to 1 MB (incrementing by 64 KB).

Selector query time The cost of running a selector on a document depends on the complexity of the selector and complexity of the document. We test selector performance by implementing MooTool's SlickSpeed test suite [32] in SVC. The SlickSpeed test suite runs 40 selectors of varying complexity against a standard 108KB web page.

We ran the test suite 1000 times and calculated the total time spent running each test. The result was a mean

time of 59.180ms ($std.dev = 2.906$), giving a mean of 1.498ms per selector.

Actions In our implementation of SVC, actions are mapped to manipulations of WebKit's internal DOM tree. The cost of DOM manipulation is dependent on the type of action performed, which largely depends on WebKit's internal performance. We measured the performance of three actions (`html`, `attr`, and `text`) in Figure 5. Each action was called with a varying amount of random data, from 0 to 1024KB. Content was escaped to ensure that additional HTML tags were not introduced.

These costs are only imposed on non-Ajax requests. For the cost of client-side DOM manipulation, see Dromaeo [28, 16].

7 Related Work

We are aware of no tool that allows for automatic progressive enhancement of non-JS sites, which SVC allows. Various frameworks allow Ajax code to be automatically created (e.g., Cappuccino, GWT, RJS (a Ruby on Rails feature) and SproutCore [3, 20, 21, 33]). Although these frameworks provide a convenient means of generating JavaScript, they do not generate code that supports both JS and non-JS browsers. In fact, these frameworks make it difficult or impossible to support non-JS browsers when using their Ajax capabilities.

Various server-side frameworks allow DOM manipulation (e.g., Genshi, DOMTemplate, GWT [19, 15, 20]). We are not aware of any framework that allows manipulation to happen on either the client or server depending on the type of request, or any framework that uses DOM manipulation as a means to allow for automatic Ajax instrumentation.

FlyingTemplates [35] proposes a system where templates are sent to the client along with data to replace variables in the template. The key idea of FlyingTemplates is to allow templates to be static and to send the data that will be substituted in the template separately. If templates are static, they can be cached by the browser and served statically by the website. SVC differs significantly from FlyingTemplates. FlyingTemplates requires a template parser to exist on the client. Additionally, it has no notion of conditional template replacement (all replacement happens on the client). FlyingTemplates also only runs on an initial template. Once it has performed replacement on a template, it cannot operate on that replaced data. This makes it unable to assist in automatic Ajax instrumentation. Also, FlyingTemplate only works in JS browsers because it relies on the client to do template substitution.

Post-processing filters exist in a number of frameworks. The main advantages of SVC over filters in other frameworks is the ability to register a filter using selectors and the ability to run filters on pieces of the document. The most similar filtering mechanism found is Genshi's [19], which allows XPath expressions to be used in filters.

8 Conclusions

The paper presents SVC as a novel programming style that can reduce development costs related to web programming. SVC allows developers to create both Ajax and non-Ajax versions of a site by composing different pieces of a page together using selectors. Developers can create Ajax sites without writing JavaScript, while also supporting non-JS browsers. SVC can be integrated with existing sites and does not interfere with previously written controllers, JavaScript, or template systems.

SVC also provides a succinct filtering mechanism that allows post-processing to be expressed more clearly than existing solutions.

We implemented a prototype of SVC for PHP and the Code Igniter framework, but it could easily be extended to other languages and frameworks. The only requirement is an HTML parser and selector support. In Python, for example, the lxml library could be used directly, without the need to compile a separate HTML parser (also, lxml supports selectors directly) [17].

Since SVC manages Ajax calls, client-side JS plugins could be written which implement common Ajax patterns. For example, supporting the "back" button (which can be tricky in Ajax applications) could be handled automatically.

Acknowledgements

We would like to thank Joe Calandrino, Will Clarkson, Thorsten von Eicken, Ari Feldman, J. Alex Halderman, Jon Howell, Tim Lee, and the anonymous referees for their helpful comments and suggestions.

References

[1] Alexa Top 1,000,000 Sites (Updated Daily). http://www.alexa.com/topsites.

[2] BYRNE, S., HORS, A. L., HÉGARET, P. L., CHAMPION, M., NICOL, G., ROBIE, J., AND WOOD, L. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, Apr. 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[3] Cappuccino. http://cappuccino.org.

[4] ÇELIK, T., ETEMAD, E. J., GLAZMAN, D., HICKSON, I., LINSS, P., AND WILLIAMS, J. Selectors Level 3. W3C proposed recommendation, W3C, Dec. 2009. 3http://www.w3.org/TR/2009/PR-css3-selectors-20091215/3.

[5] CHAMPEON, S., AND FINCK, N. Inclusive Web Design For the Future. http://www.hesketh.com/publications/inclusive_web_design_for_the_future/, Mar. 2003.

[6] Chrome. http://google.com/chrome.

[7] Closure. http://code.google.com/closure/library/.

[8] Code Igniter. http://codeigniter.com.

[9] Coralize for wordpress (v.08b). http://theblogthatnoonereads.davegrijalva.com/2006/02/12/coralize/.

[10] DEROSE, S., AND CLARK, J. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[11] Django. http://www.djangoproject.com.

[12] Django Middleware. http://docs.djangoproject.com/en/1.1/topics/http/middleware/.

[13] Dojo. http://dojotoolkit.org.

[14] DOMDocument. http://php.net/manual/en/class.domdocument.php.

[15] DOMTemplate. http://www.domtemplate.com.

[16] Dromaeo. http://dromaeo.com.

[17] FAASSEN, M. lxml. http://codespeak.net/lxml/.

[18] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with coral. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 18–18.

[19] Genshi. http://genshi.edgewall.org.

[20] Google Web Toolkit. http://code.google.com/webtoolkit/.

[21] HANSSON, D. H. RJS. http://wiki.rubyonrails.org/howtos/rjs-templates.

[22] jQuery. http://jquery.com.

[23] JSON (JavaScript Object Notation). http://www.json.org.

[24] LIE, H. W., AND BOS, B. Cascading style sheets, level 1 recommendation. first edition of a recommendation, W3C, Dec. 1996. http://www.w3.org/TR/REC-CSS1-961217.

[25] MooTools. http://mootools.net.

[26] Prototype. http://prototypejs.org.

[27] Reducisaurus. http://code.google.com/p/reducisaurus/.

[28] RESIG, J. JavaScript Performance Rundow. http://ejohn.org/blog/javascript-performance-rundown/.

[29] Ruby on Rails. http://rubyonrails.org.

[30] Safari. http://apple.com/safari/.

[31] SHIFLETT, C. Security Corner: Cross-Site Request Forgeries. http://shiflett.org/articles/cross-site-request-forgeries.

[32] SlickSpeed (MooTools). http://mootools.net/slickspeed/.

[33] SproutCore. http://sproutcore.com.

[34] Struts. http://struts.apache.org.

[35] TATSUBORI, M., AND SUZUMURA, T. Html templates that fly: a template engine approach to automated offloading from server to client. In *WWW '09: Proceedings of the 18th international conference on World wide web* (New York, NY, USA, 2009), ACM, pp. 951–960.

[36] VAN KESTEREN, A., AND HUNT, L. Selectors api level 1. Tech. rep., W3C, Dec. 2009. http://www.w3.org/TR/2009/CR-selectors-api-20091222/.

[37] The WebKit Open Source Project. http://webkit.org.

[38] WebKitGTK+. http://webkitgtk.org.

[39] The YUI Library. http://developer.yahoo.com/yui.

[40] YUI Compressor. http://developer.yahoo.com/yui/compressor/.

A Filter Examples

Below is an example filter, described in Section 5.2.2.

```
<?php
// point URL to CoralCDN
function coralize($url) {
  $host = parse_url($url, PHP_URL_HOST);
  $s = '://'.$host;
  // append '.nyud.net:8080' to host
  return str_replace($s, $s.'.nyud.net:8080',
                     $url);
}

function c_img($s) {
  $url = coralize($s->attr('src'));
  $s->attr('src', $url);
}

function c_a($s) {
  $url = coralize($s->attr('href'));
  $s->attr('href', $url);
}

// Coralize all elements with class 'coralize'
$svcfilter->register('img.coralize', 'c_img');
$svcfilter->register('a.coralize', 'c_a');
```

Figure 6: A filter which rewrites all links and images (having the class "coralize") to use the CoralCDN web service.

```
$content = preg_replace('/(\<(img|a)\s+.*?cla
  ss\=["\'].*?coralize.*?["\'].*?(src|href)\=
  ["\']http\:\/\/.*?)(\/.*?["\'].*?\>)/i', '$
  1.nyud.net:8080$4', $content);
$content = preg_replace('/(\<(img|a)\s+.*?(sr
  c|href)\=["\']http\:\/\/.*?)(\/.*?["\'].*?c
  lass\=["\'].*?coralize.*?["\'].*?\>)/i', '$
  1.nyud.net:8080$4', $content);
$content = preg_replace('/(\<(img|a)\s+.*?cla
  ss\=["\'].*?coralize.*?["\'].*?(src|href)\=
  ["\'])(\/.*?["\'].*?\>)/i', '$1http://'.$_S
  ERVER['HTTP_HOST'].'.nyud.net:8080$4',
  $content);
$content = preg_replace('/(\<(img|a)\s+.*?(sr
  c|href)\=["\'])(\/.*?["\'].*?class\=["\'].*
  ?coralize.*?["\'].*?\>)/i', '$1http://'.$_S
  ERVER['HTTP_HOST'].'.nyud.net:8080$4',
  $content);
```

Figure 7: Taken verbatim from the Coralize for Wordpress plugin [9]. In addition to being difficult to read, these regular expressions would incorrectly match an `img` element with the class "donotcoralize" and would not match elements with irregular spacing.