Journal of Web Engineering, Vol. 13, No. 3&4 (2014) 223–242

EFFICIENT DEVELOPMENT OF PROGRESSIVELY ENHANCED WEB APPLICATIONS BY SHARING PRESENTATION AND BUSINESS LOGIC BETWEEN SERVER AND CLIENT

MARKUS AST
Technische Universität Chemnitz
Chemnitz, Germany
markus.ast@informatik.tu-chemnitz.de

STEFAN WILD
Technische Universität Chemnitz
Chemnitz, Germany
stefan.wild@informatik.tu-chemnitz.de

MARTIN GAEDKE
Technische Universität Chemnitz
Chemnitz, Germany
martin.gaedke@informatik.tu-chemnitz.de

A Web application's codebase is typically divided into a server side and a client side with essential functionalities being implemented twice, such as validation or rendering. While developers can choose from a rich set of programming languages to implement a Web application's server side, they are bound to JavaScript for the client side. Recent developments like Node.js allow using JavaScript in a simple and efficient way also on the server side, but lack offering a common codebase for the entire Web application. In this article, we present the SWAC approach that aims at reducing development efforts and minimizing coding errors in order to make creating Web applications more efficiently. Based on our approach, we created the SWAC framework. It enables establishing a unified Web application codebase that provides both dynamic functionality and progressive enhancement by taking characteristic differences between server and client into account.

Keywords: Development Tools, HTML5 and Beyond, Web Standards and Protocols

1 Introduction

More and more of today's dynamic Web applications imitate behavior, look and feel of desktop applications in order to improve the experience users have when browsing the Web. They do this by moving large parts of their business logic and presentation logic from the server side to the client side[a]. While business logic is about an application's functionality and information exchange between database and user, presentation logic is about handling interactions between user and application [9]. The trend of making greater use of a Web application's client side was not only accelerated by the Internet's increasing speed and coverage for mobile devices, but also by advances in standards, which made the Web more dynamic in the last couple of years [1, 13, 19]. Development methodologies like progressive enhancement have additionally

[a]While throughout the article clients are mostly represented by browsers, other applications, e.g., Firefox OS or WebView Components in Android and iOS, are also valid clients.

blurred the line between desktop and Web applications.

Progressive enhancement focuses on Web applications that are universally accessible, intuitive, and usable by realizing all Web content and functionality only using Hypertext Markup Language (HTML). Enhancements such as advanced Cascading Style Sheet (CSS) or JavaScript are then added unobtrusively to the application [17]. Some Web-enabled devices, e.g., search engines or gaming systems, provide no or only limited support for such enhancements. By following this development methodology, they are also enabled to access corresponding Web applications. New devices like latest browsers benefit from additional technological enhancements to style and interaction.

Problem. Developing progressively enhanced Web applications requires to take presence and absence of technology support into account. This is typically accomplished by duplicating essential business and presentation logic on both server and client, which is an inefficient solution. Not only does it slow down development, but it also makes the resulting Web application more error-prone and harder to maintain.

Objective. To tackle the problem described, we propose the SWAC approach and corresponding SWAC framework. With SWAC, we enable developers to create Web applications that provide both dynamic functionalities and progressive enhancement without having to implement them twice on server and client. While progressively enhanced Web applications do not present a new kind of architecture, we aim at combing their advantages, such as search engine optimization, with the features known from single page, client-centric and Rich Internet Applications [7]. SWAC makes this possible through creating a codebase that is - in its design - compatible to the characteristic differences between server and client.

This article is organized as follows: We begin with analyzing the problem in Section 2 and use the results to discuss the suitability of existing technologies in Section 3 Taking all requirements into account, we propose the SWAC approach in Section 4 and then implement the corresponding SWAC framework in Section 5. We showcase the features of this framework in Section 6 and evaluate it in Section 7. Finally, we conclude our work in Section 8.

2 Analyzing the Development of Progressively Enhanced Web Applications

This section describes the aspects we must consider when creating Web applications that provide both dynamic functionalities and progressive enhancement. We analyze them to derive requirements that have to be satisfied for an efficient development of such Web applications.

To achieve progressive enhancement, developers have to ensure that both the content and the functionality of a Web application are accessible by all user agents, i.e., not only by the latest, most feature rich Web browsers. Paying attention to the user's Web browsing preferences, both enhanced layout and behavior are then to be made available in an optional and unobtrusive way. That is, when user agent support for layout and behavior enhancements is missing, content as well as functionality still have to remain accessible.

Consider a Web application that solely has a server side codebase[b]. This server side codebase is progressively enhanced and created using only one programming language. There are neither declarative bindings to connect parts of the user interface with the data model, nor dynamic functionalities to accomplish interactivity. While it is common to implement essential parts of a Web application on the server side, imitating desktop applications requires

[b] The term *codebase* refers to the whole source code of an application.

to have rich functionalities also on the client side. In accordance with progressive enhancement and separation of concerns, unobtrusive JavaScript allows adding dynamic functionalities, but is limited in its features, e.g., requesting a partial view and adding it to the DOM. As an example, unobtrusive JavaScript does not cover requesting plain JSON data and then building a partial view from it. Tasks like this would require creating an additional client side codebase written in JavaScript or in a language that can be compiled to JavaScript, like GWT[c].

There are many frameworks for Web application development - Rails, Django, ASP.NET MVC[d], to name a few - that rely on server side programming languages such as Ruby, Python or C#. Such frameworks are completely executed on the server side and called through requests triggered by each user interaction. Adding dynamic functionalities to them could be achieved by using unobtrusive JavaScript or writing additional client side code directly.

As a programming language for the Web application's client side, only JavaScript is available. Toolkits like jQuery[e] accelerate the client side development by providing useful helper methods for recurring actions, e.g., document traversal, event handling, and AJAX. In order to structure a client side codebase and to reduce boilerplate manipulation and event bindings, a developer can utilize several client side frameworks, e.g., Backbone, AngularJS or Knockout.[f] Client side frameworks provide means to establish declarative bindings and exchange data with Web services and, thus, ease creating dynamic functionalities [11].

With such frameworks it is also possible to create a single-page application (SPA). Single-page applications enable to retrieve all necessary sources for running the Web application with a single page load [14]. Consequently, the task of a Web application's server side is trimmed to serving static files and providing a data Web service the client side communicates with. A single-page application has two codebases, the big client and the server side API, that do not intersect in their functionality with each other, except for features like data validation. By moving all logic to the client, the Web application is not progressively enhanceable anymore.

A Service-Oriented Front-End Architecture (SOFEA) focuses on interactions with Web services via AJAX calls and therefore relies on JavaScript or proprietary logic, e.g., Flash, for providing the user interface [18]. The Web application's front-end is completely separated from the back-end. Similar to single-page applications, SOFEA-based applications are not progressively enhanced due to their dependency on JavaScript etc. on the client side.

Apart from SPA and SOFEA, Rich Internet Applications (RIA) intend imitating several aspects of desktop application to improve the user experience [7]. In the past, RIAs usually required features provided by third-party extensions like browser plug-ins. Nowadays, HTML in its version 5 is capable enough to build RIAs without having to rely on third-party extensions [5]. As it still requires JavaScript, it is not progressively enhanced.

Findings. Many of today's Web applications implement either a specific type of architecture or a combination of SPA, SOFEA and RIA. Without overlapping functionalities between the client side and the server side codebase, the client side provides functionality that the server side does not. As there is no fallback on the server side, the Web application is not progressively enhanced. Such fallbacks require writing functionalities twice. But implementing features for both codebases - on server side and on client side - makes creating and maintaining

[c] GWT `http://www.gwtproject.org/`
[d] Rails `http://rubyonrails.org/`, Django `http://djangoproject.com/`, ASP.NET MVC `http://asp.net/mvc`
[e] jQuery `http://jquery.com/`
[f] Backbone `http://backbonejs.org/`, AngularJS `http://angularjs.org/`, Knockout `http://knockoutjs.com/`

Web applications more difficult. This is because changes must be integrated using different programming languages or frameworks. Aiming at establishing a server/client compatible codebase necessitates addressing characteristic differences between server and client:

While a view on the server side is commonly string-based and generated on-demand, the client side view is based on the Document Object Model (DOM). This difference also affects navigation because the view on server side is built from scratch on every request, i.e., on each new navigation. In addition to this, it is not possible to reuse the business logic behind navigation actions for the client side without further consideration. Due to the fact that the client side is generally more vulnerable to malicious manipulations, unveiling data exchange logic to the client side has potential security issues. Only using a JavaScript front-end would negatively affect the initial loading time of the Web application [22]. Since a short initial loading time is important for keeping users from leaving the page [24], the Web application should be ready right after the initial request. As a consequence, the state established on the server side during the initial request needs to be transferred to the client to directly continue where left off on the server. This state consisting of data, view bindings, and pre-compiled fragments acts as origin and basis for all further user interaction.

Based on the analysis above, we have extracted the following requirements for efficiently developing dynamic Web applications that are progressively enhanced and take characteristic differences between server and client into account:

- Achieve progressive enhancement
- Create unified codebase for both server and client
- Enable partial business logic execution
- Allow declarative bindings
- Establish state transfer from server to client
- Implement data logic separation & access control

3 Suitability of Existing Technologies

This section discusses how existing technologies fulfill the requirements we gained as part of the findings of the analysis conducted in Section 2.

Derby is a JavaScript framework focusing on real-time and collaborative Web applications. Initially, it was completely based on WebSockets. That is, it was not possible to use an application based on Derby without WebSockets being supported and enabled. However, it recently switched to operational transformation and data synchronization using ShareJS and Racer[g] that enable server communication through AJAX [20]. Derby also provides a unified server and client interface for working with data. By moving to ShareJS, they rewrote most parts of their data layer and have not yet published an appropriate access control[h]. Despite

[g]ShareJS `http://sharejs.org`, Racer `http://racerjs.com/`
[h]Racer Features `https://github.com/codeparty/racer/blob/master/README.md#features`

the fact that it is possible to define the transition between routes[i], the routes are not meant to be hierarchically. Derby utilizes a state transfer to continue on the client where the server has left off. It also supports creating Web applications that make use of dynamic functionalities and declarative bindings while still being progressively enhanced.

As another framework in this context, Meteor[j] can create technically compatible Web application codebases using JavaScript. However, Meteor does not achieve complete compatibility between the server side and the client side because the framework lacks built-in routing functionalities, i.e., business logic is bound to DOM events directly. Meteor utilizes Fibers, i.e., one thread per request, to create synchronous APIs. Even though this is good for simplicity, it breaks with Node.js event-based characteristic. For protecting the data logic, sensitive functions can be executed in a privileged environment on the server side. Meteor supports rendering via DOM simulation on the server side taking Google's AJAX crawling specification into account [6]. While this is beneficial for search engines, Meteor usually renders the Web application only on the client side. That is, Meteor is not suitable for developing progressively enhanced Web applications. As Meteor-based Web applications are executed on client side only, there is no state to be transferred from the server side.

Rendr adapts Backbone to allow Web applications to run seamlessly on both server and client [3]. This enables creating progressively enhanced applications with Backbone. Since Rendr does not support hierarchical updates, there is no state to be transferred to the client. Not providing a data layer on its own, it leaves the problem of a unified data API and access control unsolved. Like Backbone, Rendr lacks missing support for declarative bindings [15].

Compared to the frameworks analyzed so far, the API of Yahoo! Mojito[k] is quite different from conventional back-end frameworks as far as it does not offer a homogeneous data API for both server- and client side. Although Mojito can be used to build technically compatible Web application codebases, it lacks native support for client side routing, i.e., via HTML5 History [2]. For data query and manipulation, Mojito relies on existing services accessible through HTTP[l] . While Mojito's presentation logic can partially update the view, declarative bindings are not directly provided. It needs a boilerplate view and event handling.

Even though all analyzed frameworks allow sharing codebase parts between server and client, they do not offer all facilities that would be required to automatically create progressively enhanced Web applications by paying attention to server/client differences. Our assessment results are summarized in Table 1, where *"+"* implies good support and *"-"* does not.

4 The SWAC Approach for Sharing a Web Application's Codebase

To remedy the shortcomings of existing technologies and completely fulfill the requirements associated to the development process of dynamic, progressively enhanced Web applications, we propose SWAC - an approach for Sharing a Web Application's Codebase. SWAC establishes a unified server/client compatible Web application codebase. It is designed to execute partial

[i] While routes might be considered as similar to navigational paths and links, we omit using this terminology here. We think that applying navigation terminology is limited to express that a user *intentionally navigates* from one Web page to another. In contrast, routes address business logic that can and often should be executed unnoticed in the background, i.e., without the user's attention.

[j] Meteor `http://docs.meteor.com/`

[k] Mojito `http://developer.yahoo.com/cocktails/mojito/`

[l] Mojito Overview `http://developer.yahoo.com/cocktails/mojito/docs/intro/mojito_overview.html`

Table 1. Assessment of Related Work

	Progressive Enhancement	**Unified Codebase**	**Partial Business Logic**	**Declarative Bindings**	**State Transfer**	**Access Control**
Derby	+	+	-	+	+	-
Meteor	-	-	-	+	-	+
Mojito	+	-	-	-	-	-
Rendr	+	+	-	-	-	-

business logic of a application by defining routes as a route hierarchy. To update only the affected parts of a view on data changes, the SWAC approach supports fragmentation of views into parts. Through declarative bindings, these parts are automatically updated once their underlying data changes. It achieves access control by an additional layer between business logic and logic responsible for communicating with the database. For a seamless transition from server- to client-execution of the Web application, SWAC enables to automatically transfer the state from the server to the client. The following subsections detail how the SWAC approach takes characteristic server/client differences into account.

4.1 Partial Business Logic Execution

A route hierarchy is an essential part of the business logic. It defines the relationship of routes in an application to each other. SWAC utilizes such route hierarchy to determine the necessary parts to be executed for reflecting changes between two user interactions. That is, it expects the URL to be hierarchical, which is also considered a best practice [12].

There is no standalone business logic for each complete route. Instead, the business logic is separated into parts, where each part reflects the changes necessary to move from one route to an immediately following one. This allows executing only necessary parts required to navigate from one route to another on the client side. To handle scenarios of routes requiring logic that is incompatible with both sides, e.g., logic provided by third-party frameworks like jQuery and Dojo, every route can consist of an additional client-only part. While the client-only part is optional, the server/client compatible part is always required. The SWAC approach supports three different routing schemes, where each schema is distinct in terms of handling the route hierarchy tree:

- server side execution, i.e., the route is executed on the server side,
- client side initialization, i.e., the client is initialized to take off the application, and
- client side execution, i.e., the client side navigates through the route hierarchy tree.

The following terminology is applied to describe the routing algorithm: We define $G(V, E)$ as a directed graph representing the route hierarchy, $u \in V$ as a route and $(x, y) \in E$ as a directed edge connecting dependent routes. Additionally, we define $A(u)$ as a subset of G, with each node $x \in A$ being an ancestor of u. A node $a \in A$ is called a *common ancestor* of u and v if a is an ancestor of both of them. $w(u, v)$ is called the *lowest common ancestor* of u and v. In analogy to $A(u)$, we define $D(u)$ as a subset of G, with each node $x \in D$ being a descendant of u. We define $T(u)$ as a tree inside G with $u \in T$. Other trees like $T(k)$ can also exist within $G(V, E)$. These definitions are illustrated in Figure 1.

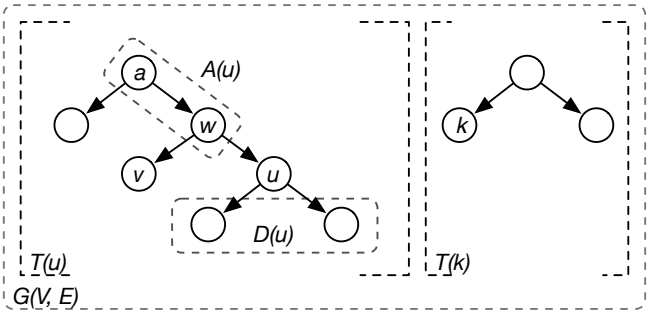

Fig. 1. Routing Terminology

Server Side Execution. The server side of a Web application that follows the SWAC approach is stateless. That is, a request to the server always requires executing the whole logic responsible for providing the desired result. Calling a route v on the server side results in executing v and all ancestors of v, i.e., all nodes being an element of $A(v)$. These routes are executed in the order they are specified in.

Client Side Initialization. The initial request is always completely processed on the server side. Afterwards, SWAC allows executing most of an application's functionality decoupled from the server on the client side. Therefore, client-only parts of the current route are executed. This is done by applying the same method as used for server side execution with the difference of executing the client-only and not the server/client compatible part of the route.

Client Side Execution. On the client side, only routes that are responsible for changes between two user interactions are executed. There are four sub-scenarios for client side execution that take different positions of the target route into account. If target route v is not an element of tree $T(u)$ of starting position u, it is executed the same way as on the server side. That is, v and all ancestors $A(v)$ are executed in their appropriate order. Otherwise, target route v is an element of $T(u)$. If $v \in T(u)$, v could be an ancestor of u, i.e., $v \in A(u)$, v could be a descendant of u, i.e., $v \in D(u)$ or otherwise, v is inside another branch of $T(u)$. In case v is an ancestor of u, only v is executed. If v being a descendant of u, every route from u down to v is executed. Otherwise, every route from lowest common ancestor $w(u, v)$ down to v is executed. $R(u, v)$, as the set of routes to be executed, is built using the following method:

$$R(u,v) = \begin{cases} A(v) \cup \{v\} & \text{if } v \notin T(u), \\ \{v\} & \text{if } v \in T(u) \wedge v \in A(u), \\ [A(v) \cap D(u)] \cup \{v\} & \text{if } v \in T(u) \wedge v \in D(u), \\ [A(v) \cap D(w(u,v))] \cup \{v\} & \text{if } v \in T(u) \wedge v \notin A(u) \wedge v \notin D(u). \end{cases} \tag{1}$$

4.2 Declarative Bindings

The view is the part of a Web application generating the visual representation of the application for the user based on the data provided. It additionally acts as the interface between user and application. This makes it necessary to react on user interactions and update the visual representation accordingly. While every single part of a view can be directly updated on the client side using DOM, this is impossible with a string-based template as normally done on the server side. Due to the fact that parsing the whole template is expensive, having a

full-fledged DOM on the server side would negatively affect the performance [16]. Therefore, the SWAC approach allows dividing the view into smaller pieces using fragments on the server side. Unlike re-rendering the entire view on every data change, this enables to re-render parts of the view independently. Such fragments can also be used to achieve declarative bindings to reduce boilerplate DOM manipulations and event bindings. Therefore, fragments establish bindings between themselves and data they render. These bindings automatically trigger the fragment re-rendering once their underlying data changes, as shown in Figure 2.

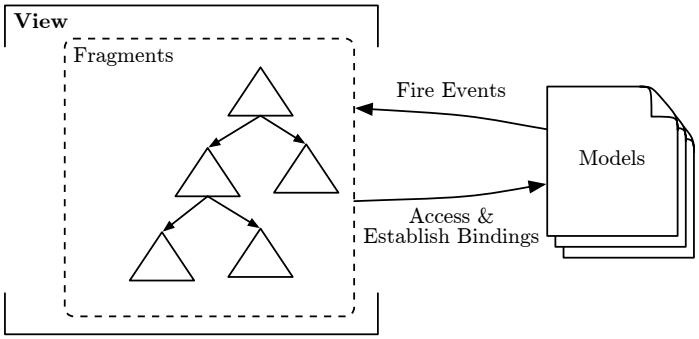

Fig. 2. View Bindings

4.3 Data Logic Separation and Access Control

To reuse an application's codebase, every part of the application's logic is shared between server and client, unless it is explicitly declared as server-only logic. There is one exception: communication between business logic and the database is always executed on the server side.

A typical three tier architecture in the Web has two tiers for presentation logic and business logic, and a third one for the database. By splitting the business tier into two layers, i.e., the service layer and the business layer, we achieved keeping execution of logic for the database communication always on the server side. The service layer provides an API for communicating with the database and is never shared with the client, i.e., the client side cannot directly access the database. However, both sides share the same API. Data API calls executed on the client side are proxied through an automatically provided RESTful API on the server, as illustrated in Figure 3. That is, authentication and authorization logic for data access is always executed in a privileged environment on the server side.

There are two options to establish route security. First, avoiding route sharing completely, which is the most secure way for routes referring to proprietary functionality. As the benefit of SWAC results from sharing code, we suggest using this option only if necessary. Second, for shared routes, SWAC provides so-called hooks for both authentication and authorization logic.

Executing authentication or authorization on the client side is not that useful because of the client side's vulnerability to malicious manipulations. SWAC therefore enables developers to divide applications into several areas, as shown in Figure 4. These areas are isolated from each other, i.e., navigating between them triggers requests to the server. This allows the execution of authentication and authorization logic in a privileged environment on the server side. Passing such authentication/authorization logic is necessary for the client to obtain the

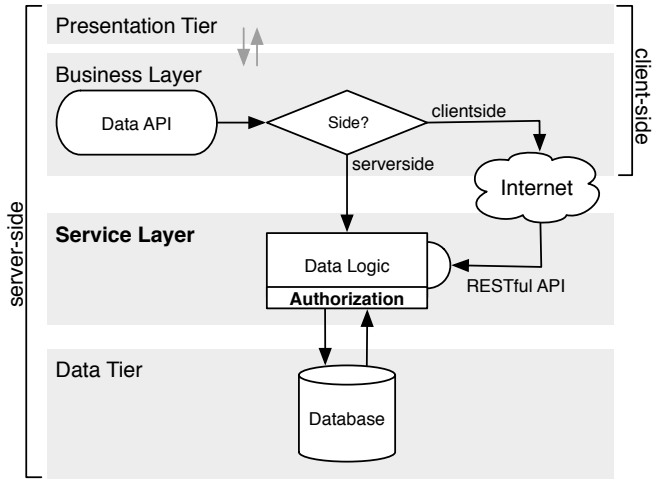

Fig. 3. Service Layer

area's bundle, i.e., business logic of an area, and to call the route at all. These areas provide a way to support the separation of an application into different security levels and enable responding to users who try to access application parts they are not authorized to.

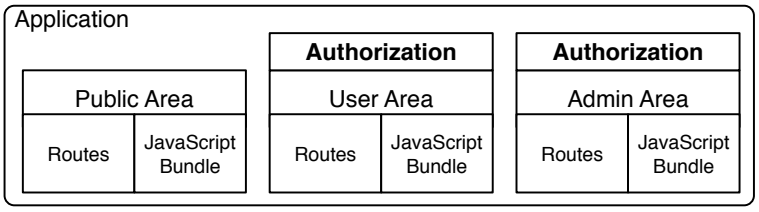

Fig. 4. Application Areas

4.4 State Transfer from Server to Client

The initial request is processed and rendered completely on the server. Enabling the client to take off the application's execution requires making the state available to the client. The state includes following information:

- data contained in models and collections,
- fragment positions,
- events and their listeners, and
- pre-compiled templates.

Such state information is necessary to update the view on the client side on data changes caused by user interactions. Although the client can always retrieve data from the server, it would be unnecessary to retrieve data twice - once on the server side and once the client

takes off the Web application. Fragments can update themselves on certain events, e.g., data changes. Bindings between fragments and events are established once a fragment is rendered for the first time, i.e., on the server side. To automatically reflect data changes, bindings must be transferred to the client. Since a fragment's position and template are required for fragment re-rendering, associated data is also transferred to the client. To avoid compiling templates again on the client side, they are transferred in a pre-compiled form.

5 The SWAC Framework

For utilizing the SWAC approach when developing progressively enhanced Web applications, we created the SWAC framework. The SWAC framework is completely implemented in JavaScript. It only relies on JavaScript and Node.js on the server side. This, however, does not imply that Web browsers have to support JavaScript for using Web applications developed with the SWAC framework. Even though focusing on JavaScript enables to establish technically compatible server/client codebase, the other requirements addressed by the SWAC approach in Section 4 need to be implemented separately, as shown in the following subsections.

5.1 Partial Business Logic Execution

To make it easy to apply the traversing approaches of the different routing schemes, all routes of a SWAC-based Web application are managed as a tree structure. The paths for every node of this tree structure are computed by combining the partial paths of their associated ancestors. These paths are then used to define the complete routes in a Sinatra-style approach [10].

Accomplishing progressive enhancement requires reflecting user interactions, i.e., route changes, also on the client side. This allows both sharing and bookmarking links to the current position in the application. To achieve this, we utilize the HTML5 History API that allows to manually adjust the browser location [2]. That is, the SWAC framework detects user interactions with the Web application and intercepts each link clicked and each form submitted as shown in Figure 5. Once intercepted, the framework tries matching the link or form target with an application's route. If there is no matching route to be triggered, the browser's default behavior is used, i.e., requesting the server. By making the application thereby crawlable for search engines, this contributes to create progressively enhanced Web applications.

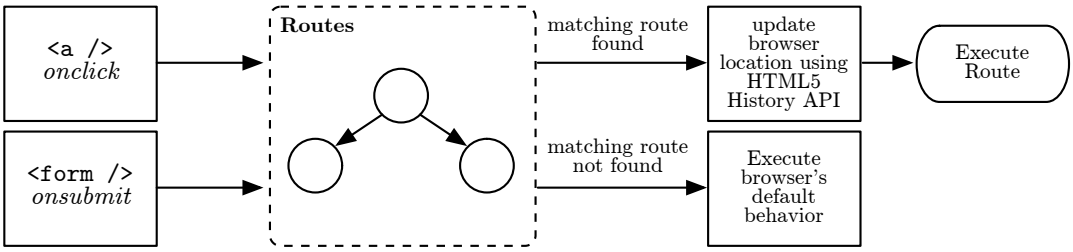

Fig. 5. Client Side Routing

In contrast to this client side routing approach, the server side tries to find a match between the requested path and an application route and then responds accordingly.

5.2 Declarative Bindings

For reflecting data changes by only re-rendering necessary view parts, the SWAC framework allows creating an efficient and compact partition similar to DOM for string-based templates on the server side by utilizing embedded JavaScript. It achieves the fragmentation by wrapping parts of a template into an appropriate block expression, as exemplary shown below:

```
<div>
  @fragment(function() { Self-updating fragment })
</div>
```

Templates are just HTML files with inline JavaScript. There is no problem using responsiveness frameworks like Bootstrap or Foundation in addition to the SWAC framework. Re-rendering fragments as described by the SWAC approach requires tracking their position. Position tracking of fragments could be accomplished easily by wrapping them into HTML elements, which are identified and accessed via IDs. However, this procedure has some drawbacks, e.g., HTML table rows do not allow container tags. Considering a collection, where each item is represented through two HTML rows, there is no valid way to wrap each of these two `<tr>` rows into their own container [2]. Only the comment node fits our purpose. It is allowed to reside inside every HTML element, except for the `title` and the `textarea` element. As comments cannot wrap content to be rendered, they must act as start and end markers for a fragment, as also done in Derby[m], as an example, and shown below:

```
<div>
  <!-- -{1 --> Self-updating fragment <!-- -1} -->
</div>
```

On the server side, these comments are parts of the rendered string. This requires initializing the fragment positions once the DOM is built on the client side. As soon as the client builds a document, a method must detect all relevant comments and assigns them to their corresponding fragments. Such a simplified method is listed below:

```
var walker = document.createTreeWalker(
                    start, NodeFilter.SHOW_COMMENT)
while(walker.nextNode()) {
 if (!isRelevantComment(walker.currentNode)) continue
 // assign comments to their fragments
 fragment[isStartNode(walker.currentNode.nodeValue)
  ? 'startNode' : 'endNode'] = walker.currentNode
}
```

It would be also possible to mimic this approach via Web Components [4]. There, the template tag could be used to transfer templates from server to client. However, as we already compile each template on the server side to a function ready to be called with some data, it would not be beneficial to repeat this step on the client side.

To implement self-updating view fragments without making the API inconvenient, properties used inside a fragment must be enabled to interact with the fragment they are accessed from. The SWAC framework utilizes JavaScript Function.caller[n] property for this purpose. It

[m] Derby `http://derbyjs.com/`

[n] `http://developer.mozilla.org/docs/JavaScript/Reference/Global_Objects/Function/caller`

points to the object that called the function the property was accessed from. Although this Function.caller property is not part of the ECMA standard [8], it is currently supported by all major browsers (Firefox, Safari, Chrome, Opera and Internet Explorer) [25]. In the SWAC framework, each data property is a getter that binds to the fragment it is called from:

```
Object.defineProperty(model, prop, {
 get: function get() {
  if (typeof get.caller.fragment !== 'undefined')
   this.on('change.' + prop, get.caller.fragment.refresh)
  return value
 }, set: [...], enumerable: true
})
```

Re-rendering a fragment consists of three steps:

1. delete the fragment's content,
2. re-render the fragment, and
3. reinsert it into the DOM.

Step 1 and step 3 require knowledge about the fragment's position. Hence, it needs to be tracked. Declarative bindings ensure that fragments update themselves on appropriate data changes. They are created automatically once data is accessed.

5.3 Data Logic Separation and Access Control

The SWAC framework provides hooks for injecting custom authentication/authorization logic. To avoid the need for developing custom logic, we facilitate reusing already existing Node.js packages, e.g., for OAuth or OpenID. Since the authentication/authorization logic has to be executed in a privileged environment, changing the security level must cause a request to the server. That is, security-related logic can be attached to different areas of the Web application. Switching between these areas always entails a request to the server. Not only allows this securing the routes of an area, but also the application bundle intrinsically prevents users from requesting application logic they are not authorized to. An exemplary access control definition for such an area, which restricts access to authenticated users only, is listed below:

```
swac.area(__dirname + '/app.js', {
  allow: function(req) { return req.isAuthenticated() }
})
```

For authorizing data read and write operations, SWAC facilitates securing the read and write operations for data even down to their properties. An exemplary API usage, which only allows update, delete and read access to the user model by the owner, is shown below:

```
swac.Model.define('User', function() {
  this.allow({
    all:  function(req, user) { return req.user.id == user.id },
    post: function(req) { return true }
  })
})
```

5.4 State Transfer from Server to Client

To transfer the state, it needs to be serialized. Serialization of complex objects asks for additional logic to cope with circular references, functions as well as closures. There is no built-in mechanism available that allows serialization of all kinds of JavaScript objects [8]. The SWAC framework achieves sufficient object serialization by resolving circular references, avoiding closures and utilizing the service locator pattern to restore object instances.

SWAC resolves circular references by tagging an object as visited on its first occurrence. This allows identifying references to objects that are already part of the state. Further occurrences of objects are replaced with a JSONPath[o] to their first occurrence. To serialize functions, the framework avoids closures and uses their string representations created via `toString()`. For restoring objects created by a constructor, we implement the service locator pattern. The service locator registers the constructors and all objects created by them are tagged appropriately. This enables restoring such objects on deserialization. These techniques in combination are a powerful toolkit to deserialize/serialize complex JavaScript objects.

Appending a serialized object to the response requires an appropriated textual representation. Being directly supported by JavaScript, using JSON is the simplest way to achieve this [8]. JavaScript objects can easily be encoded and decoded into JSON, as shown below:

```
var encoded = JSON.stringify(obj)
var decoded = JSON.parse(encoded)
```

While serializing JSON produces smaller string sizes compared to XML, it still comes with a significant overhead. This is because of redundant data for recurring property names and values. Besides the textual specifications of JSON and XML, there are several other binary specifications available to serialize JavaScript objects, e.g., MessagePack[p] or Protocol Buffers[q].

To choose the best candidate, we compared the plain JSON format with both binary formats in terms of time required for both encoding and decoding, and the resulting string size. The encoding and decoding for both binary formats also includes the time necessary for converting between buffer and Base64 encoding as a textual representation. Since the objects to be serialized have no strict structure, Protocol Buffers - requiring such struct definition - is not an appropriate candidate for this. The result of the binary format MessagePack was not as satisfying as expected. So, we also considered candidates for compressing JSON. That is, we added the Lempel-Ziv-Welch (LZW) compression, which generates a dictionary for recurring strings, and PSON[r], which claims to combine the best of JSON, BJSON, ProtoBuf and ZIP. Table 2 lists all results.

Table 2. State Transfer - Compression Comparison

	Average Time [ms]	**Size [KB]**
plain	3,107	67
MessagePack	38,784	70
BSON	32,586	57
LZW	37,996	23

[o] JSONPath `http://goessner.net/articles/JsonPath/`
[p] MessagePack `http://msgpack.org/`
[q] Protocol Buffers `http://code.google.com/p/protobuf/`
[r] PSON `https://github.com/dcodeIO/PSON`

The results show that LZW generates the smallest result and requires an acceptable time for both encoding and decoding. Additionally, the LZW library is much smaller than for the binary formats. Therefore, we choose this compression for transferring the state.

6 Exemplary Application of the SWAC Framework

In this section, we present an example for developing a simple task & document management application. We apply the proposed SWAC framework and best practices to implement a single codebase for the entire Web application and realize progressive enhancement.

Consider a simple application for managing tasks and documents as shown in Figure 6. There are tasks, documents and projects. Tasks and documents are both assigned to projects and each project can contain several tasks and documents. There are five routes associated to these elements, i.e., one to the project list, a second to a specific project, a third to the tasks associated to a project, a fourth to the documents within a project, and another route acting as an entry point for the application. Four separate views are rendered on the basis of these routes, i.e., the layout, the project, the tasks and the documents.

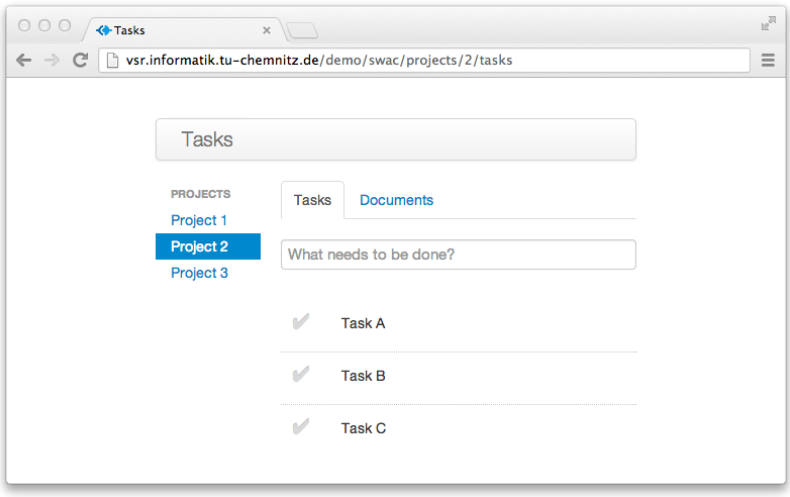

Fig. 6. Screenshot of Sample Web application

The SWAC framework enables developers of such a task & document management application to combine these five distinct routes into one, as shown in Figure 7. This route is

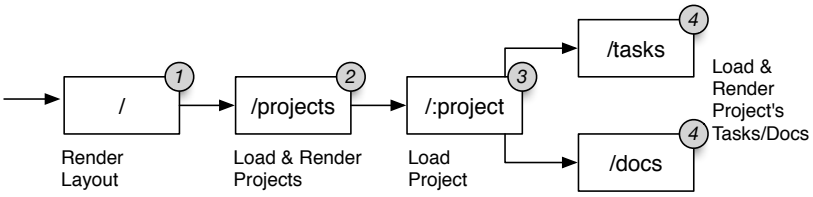

Fig. 7. Route Hierarchy of Sample Application

further split into five hierarchical pieces. We therefore have consolidated the business logic of these routes: (1) the root route rendering the layout, (2) the projects route loading and rendering the list of projects, (3) the project route loading a specific project and (4) tasks and documents routes to render all tasks and documents of the selected project. That is, the tasks and documents routes reuse logic introduced with the project route (1-3). This separation enables moving through a route step by step to execute only the necessary parts of the business logic, e.g., projects, tasks or documents.

The route hierarchy illustrated by Figure 7 contains a parameter (`:project`). The SWAC framework handles each parameter as a distinct branch of the route hierarchy (cf. Figure 8). This is necessary for correct routing. For instance, switching from the tasks of one project (u) to the tasks of another project (v) requires executing the `/projects/:project` route again for the new project. Routes to be executed in this scenario are highlighted green in Figure 8.

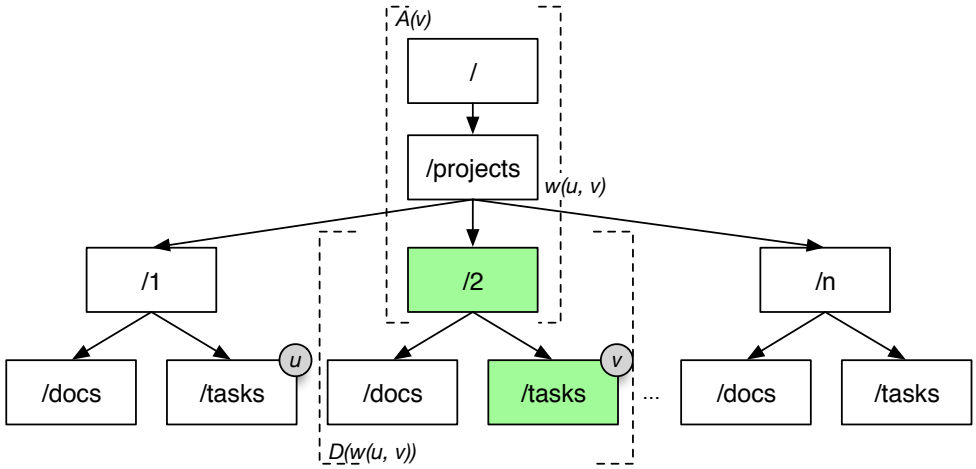

Fig. 8. Routing example $T(u)$ (1)

Figure 9 covers two additional scenarios for this route hierarchy. On the left hand side, target route v is an ancestor of the current route u. On the right hand side, target route v is a descendant. As in the previous example, the highlighted routes are executed.

The presentation logic is responsible for reflecting changes based on the business logic

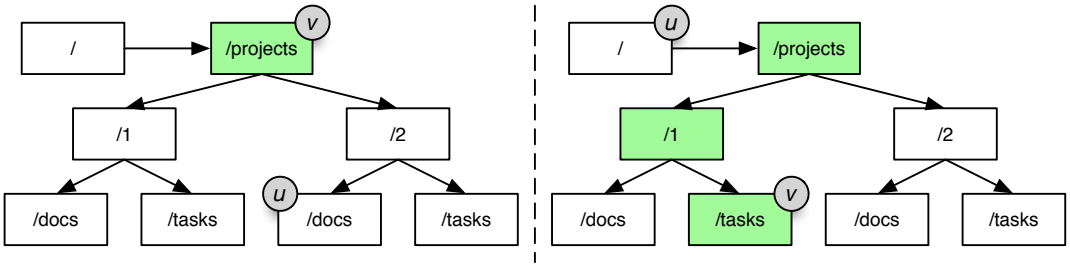

Fig. 9. Routing example $T(u)$ (2)

addressed by these routes. To achieve this without re-rendering, the SWAC framework allows splitting the view into pieces called fragments. These fragments are used to construct the view step by step or to update parts of the view once underlying data changes. As shown in Figure 10, the sample application consists of a fragment for the project list, the task collection, the document collection and one for each project, task and document.

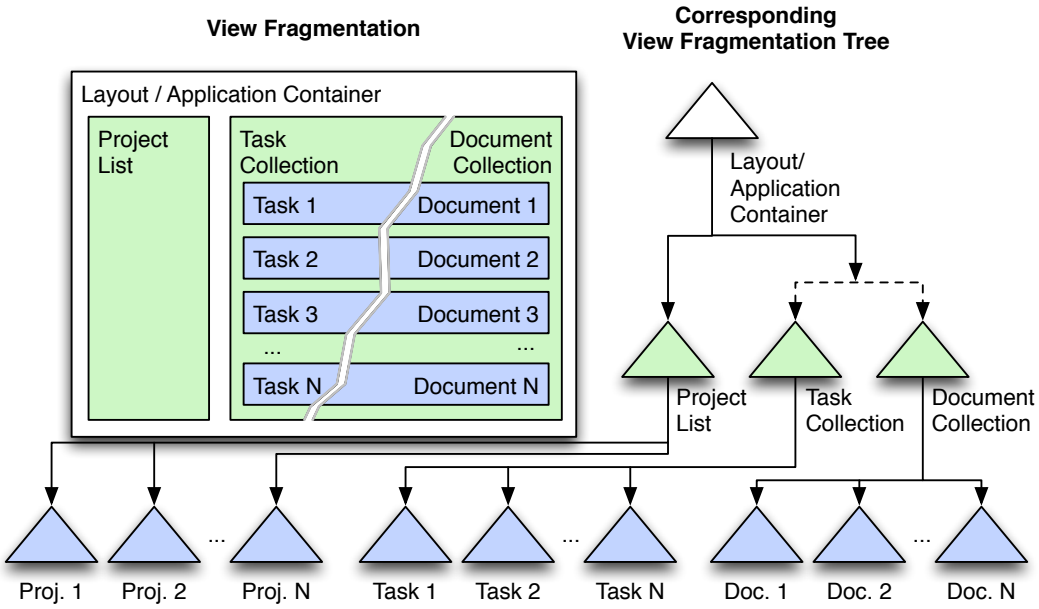

Fig. 10. View Fragmentation of Sample Application

Having implemented routes and views as described, requesting the list of projects from the server would be as follows: The SWAC framework automatically executes each route sequentially down to the projects route in the route hierarchy, i.e., (1) and then (2) in Figure 7. The layout and the project list are rendered accordingly. The result of this request not only contains the rendered view composed of the layout and the project list. It also contains the state consisting of the underlying data, the bindings, the fragment's pre-compiled templates[s] and their positions inside the view. The bindings are established to re-render fragments on data changes like creating, renaming or removing projects.

At this point in time, most of the application's functionalities can be executed from the server side in a decoupled way, i.e., only data access and manipulation operations have to involve server communication. It is important to note that in the SWAC framework, the logic responsible for data exchange with the database is not shared with the client because of potential security concerns. As the data logic remains on the server all the time, the SWAC framework automatically provides an appropriate API allowing client side data access to be proxied through. In the example, selecting a project on the client side would work as follows: The client makes an AJAX request to the server to get the tasks of the selected project.

[s] A pre-compiled template is a JavaScript function responsible for creating HTML for the data provided in the fragment.

Additionally, the client requests the pre-compiled fragments of the tasks template, which are used for the appropriate rendering. The resulting fragments ensure that the task items can be re-rendered once their underlying data changes, e.g., selecting another project results in the execution of the associated route part. This would cause re-rendering view fragments of the task items.

Demonstration: This sample application created with the SWAC framework is available at: `http://vsr.informatik.tu-chemnitz.de/demo/swac/`

7 Evaluation

This section discusses the evaluation of SWAC. We claim that SWAC allows efficiently developing and maintaining progressively enhanced Web applications and that this increase in efficiency, compared with conventional development concepts and other Web development frameworks, results from sharing presentation and business logic between server and client. We further expect that the following aspects were not exclusively decisive for devising SWAC, but taking them into account was important to realize the sharing of presentation and business logic between server and client, and to support our claim:

- Web application development simplified and made more efficient through establishing a unified codebase for both server and client
- Changes reflected between two interactions through executing business logic partially
- Views updated on data changes through creating declarative bindings
- Execution continued on the client side where left off on the server side through transferring the state from server to client
- Data security achieved through separating data logic and implementing access control

Having described our expectations, we outline the evaluation environment and process next.

7.1 Evaluation Environment and Process

There are many different settings - use cases - where the SWAC approach and framework could be utilized in, i.e., almost all scenarios of developing Web applications that consist of a server side and a client side. Some evaluation methods would allow inspecting SWAC's capabilities in such setting, e.g., field experiment or case study. However, efficiency, effectiveness and quality of written source code highly depends on the individual developer, and the available knowledge and experience in utilizing the approach, framework and the underlying programming language JavaScript. Due to the fact that such dependency would make the assessment significantly more difficult[t], we decided to conduct the evaluation as a controlled, theoretical and objective-based study to determine the extent to which SWAC's objectives have been achieved. As Stufflebeam described in [21], the "objectives-based approach is especially applicable in assessing tightly focused projects that have clear, supportable objectives" and that "such studies can be

[t] With regard to finding eligible developers; comparing their skills and experiences; selecting suitable criteria for comparing Web application frameworks (Microsoft ASP.NET MVC vs. Derby) and generated code; preparing, instrumenting and measuring within a test environment etc.

strengthened by judging project objectives against the intended beneficiaries' assessed needs, searching for side effects, and studying the process as well as the outcomes".

An objective-based study is the most prevalent approach in program evaluation and is applicable to be performed internally by program developers [21]. It is therefore well-suited to be applied here. We have used the requirements gained from the analysis on developing progressively enhanced Web applications in Section 2 as objectives for creating the SWAC approach. They are also applied as evaluation criteria in this objective-based study. We interpret the information collected while developing and utilizing SWAC, e.g., in [23], to determine how well each objective has been achieved.

After outlining the evaluation setup, we discuss the findings in the following subsection.

7.2 Discussion of Findings

In order to substantiate our assertion about an efficiency increase through SWAC, we pay particular attention to the objectives we consider as important in this context:

Unified codebase. With SWAC, developers can establish a unified codebase for the entire Web application. This relieves them not only from the need to implement two separate codebases, but also from dealing with different programming languages, e.g., C# on the server side and JavaScript on the client side. As a consequence, the entry barrier and initial effort of making oneself familiar with a programming language is reduced. We expect that it is simpler to acquaint oneself with only one programming language and an open source framework, which is transparently using this programming language than becoming familiar with two different programming languages and optional frameworks.

Apart from language advantages, methods encapsulating certain functionality, e.g., for validation and rendering, only need to be implemented as part of one codebase with SWAC. Such methods are available to both client side and server side and therefore enable, for instance, to use the *same* method for validating user inputs on the client side and after submission on the server side, which is necessary for the sake of security. This avoids creating redundant methods and, thus, reduces the development effort and saves time. Thereby the efforts and costs for maintaining and evolving Web applications are also minimized compared to conventional approaches, which rely on two separate programming languages or codebases.

Partial business logic execution. Reflecting the changes required between two user interactions instead of completely applying the final state does not only decrease the amount of data to be transferred, but also avoids introducing code redundancies. That is, in place of developing and utilizing a standalone business logic for each route, only necessary business logic parts are considered and executed when moving from one route to another. Implementing and maintaining a partial business logic is less expensive than implementing and maintaining it completely because common parts, recurring in several routes, only need to be created once. This contributes to making Web application development more efficient.

Declarative bindings. Similar to the advantages introduced by SWAC's partial business logic execution, the approach also allows partial view rendering rather than rendering it completely. This is accomplished by declarative bindings. Due to this feature, Web application developers do not have to manually implement complex operations for loading and re-rendering views, e.g., via redirects and AJAX calls. They are enabled to bind view elements to data. Once the data changes the view elements are updated automatically. This decreases the manual coding

effort during development and saves view data that would need to be transferred at runtime.

State transfer and access control. Transferring the state from the server to the client and controlling data access only has a minor impact on the efficiency of Web application development with SWAC. The state including all required information needs to be transferred for continuing on the client side where left-off on the server side, and to utilize partial business logic execution and declarative bindings. This enables data pre-fetch in the sense that data necessary to run the Web application decoupled from the server side is already available on the client side. That is, the Web application remains operational even though JavaScript is disabled - at any time - on the client side. While the state could be obtained at any time from the server side, SWAC transfers it on the first request, i.e., developers do not have to care about how to access the state on the client side and how to keep the state consistent. It is managed by SWAC and, thus, simplifies the development process. SWAC allows dividing the application into isolated areas and applies access control to them. Navigating between such areas is handled by SWAC through triggering appropriate requests. Developers do not need to manually decide when to do authentication/authorization, but can rely on the framework by adding routes and JavaScript bundles to a particular area they expect to be suited.

Concluding the evaluation, all objectives that contribute to a simpler and more efficient Web application development have been achieved with SWAC. Particularly noteworthy are the efficiency benefits that result from establishing a unified codebase and executing partial business logic. While other approaches and frameworks consider some objectives that are also incorporated in SWAC, they lack in addressing all of them in a holistic manner (cf. Section 3).

8 Conclusion

With SWAC we provided an approach for efficiently developing progressively enhanced Web application through sharing the Web application's codebase between server and client. Although we accomplished the technical compatibility of the codebase on both server- and client side using JavaScript, the characteristic differences between server and client made it necessary to create a business and presentation logic compatible to string- and DOM-based views. This was realized by splitting routes into hierarchical parts. For making the presentation logic compatible to this change, we added a mechanism to split the view into fragments. The fragments are automatically updated once underlying data changes. In addition to these contributions, we integrated a facility into SWAC that allows the client to seamlessly take over the application after the state was automatically transferred from the server. While data exchange logic is not shared by the SWAC framework because of security concerns, we enabled the client to unobtrusively proxy database calls through the server.

In future work, we intend to increase the modularity of the SWAC framework in order to make modules not only usable on their own, but also exchangeable with third party modules. We also plan investigating collaborative editing scenarios asking for facilities like data push. As another topic of interest, we want to assist in storing data by adding a server/client compatible local/session storage solution. In addition to these enhancements, we will research on providing a smart component on top of SWAC that enables detecting the best execution target for specific parts of a Web application. That is, we will aim at optimizing a Web application's response time by automatically delegating execution of functions to either server or client based on certain criteria like technology support.

References

1. Belson, D., Leighton, T., Rinklin, B.: The State of the Internet 5(3) (2012)
2. Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., Pfeiffer, S., Hickson, I.: HTML5 - A Vocabulary and Associated APIs for HTML and XHTML (2013), `http://www.w3.org/TR/html5/`, [Online; accessed Mar 23, 2014]
3. Brehm, S.: Introducing Rendr: Run your Backbone.js Apps on the Client and Server (2013)
4. Cooney, D., Glazkov, D.: Introduction to Web Components (2013), `http://www.w3.org/TR/components-intro/`, [Online; accessed Mar 23, 2014]
5. David, M.: HTML5: Designing Rich Internet Applications. Focal Press, 2nd edn. (2012)
6. DeBergalis, M.: Meteor - Search engine optimization (2012), `http://www.meteor.com/blog/2012/08/08/search-engine-optimization`, [Online; accessed Mar 23, 2014]
7. Duhl, J.: Rich Internet Applications (2003)
8. Ecma International: ECMA-262 ECMAScript Language Specification 5.1 Edition (2011)
9. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc. (2002)
10. Harris, A., Haase, K.: Sinatra: Up and Running. O'Reilly (2011)
11. Kumar, V.: HTML5 and JavaScript Apps with MVVM and Knockout. In: Beginning Windows 8 Data Development, pp. 13–27. Apress (2013)
12. Masinter, L., Berners-Lee, T., Fielding, R.T.: Uniform Resource Identifier (URI): Generic Syntax (2005), `http://tools.ietf.org/html/rfc3986`, [Online; accessed Mar 23, 2014]
13. Meeker, M., Wu, L.: 2012 Internet Trends (2012)
14. Mesbah, A., van Deursen, A.: Migrating Multi-page Web Applications to Single-page AJAX Interfaces. In: Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on. pp. 181–190 (March 2007)
15. Mirgorod, V.: Backbone.js Cookbook. Packt Publishing Ltd (2013)
16. Nicola, M., John, J.: XML Parsing: A Threat to Database Performance. In: Proceedings of the 12th International Conference on Information and Knowledge Management. pp. 175–178. ACM Press (2003)
17. Parker, T., Jehl, S., Wachs, M.C., Toland, P.: Designing with Progressive Enhancement: Building the Web that Works for Everyone. New Riders Publishing (2010)
18. Prasad, Ganesh Taneja, R.T.V.: Life above the Service Tier (10 2007)
19. Smith, A.: Cell Internet Use 2012 (2012)
20. Smith, N., Noguchi, B.: Derby v0.5.0 (2013), `http://blog.derbyjs.com/2013/06/04/derby-v0-dot-5-0/`, [Online; accessed Mar 23, 2014]
21. Stufflebeam, D.: Evaluation Models. New directions for evaluation 2001(89), 7–98 (2001)
22. Webb, D.: Improving performance on twitter.com (2012), `https://blog.twitter.com/2012/improving-performance-twittercom`, [Online; accessed Mar 23, 2014]
23. Wild, S., Ast, M., Gaedke, M.: Towards a Context-Aware WebID Certificate Creation Taking Individual Conditions and Trust Needs into Account. In: Weippl, E., Indrawan-Santiago, M., Steinbauer, M., Kotsis, G., Khalil, I. (eds.) Proceedings of the 15th International Conference on Information Integration and Web-based Applications & Services (iiWAS2013). pp. 532–541. ACM (2013), isbn: 978-1-4503-2113-6
24. Work, S.: How Loading Time Affects Your Bottom Line (2011), `http://blog.kissmetrics.com/loading-time/`, [Online; accessed Mar 23, 2014]
25. Zaytsev, J.: ECMAScript Compatibility Table (2013), `http://kangax.github.com/es5-compat-table/`, [Online; accessed Mar 23, 2014]