The SWAC Approach for Sharing a Web Application's Codebase Between Server and Client

Markus Ast, Stefan Wild, and Martin Gaedke

Chemnitz University of Technology, Germany
{firstname.lastname}@informatik.tu-chemnitz.de

Abstract. A Web application's codebase is typically split into a server-side and a client-side with essential functionalities being implemented twice, such as validation or rendering. For implementing the codebase on the client, JavaScript, HTML and CSS are languages that all modern Web browsers can interpret. As the counterpart, the server-side codebase can be realized by plenty of programming languages, which provide facilities to implement standardized communication interfaces. While recent developments such as Node.js allow using JavaScript as a client-side programming languages outside the browser in a simple and efficient way also on the server-side, they lack offering a common codebase for the entire Web application. We present a flexible approach to enable sharing of presentation and business logic between server and client using the same codebase. Our approach aims at reducing development efforts and minimizing coding errors, while taking characteristic differences between server and client into account. We show the impact of our solution during an evaluation and in comparison to related work.

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 by moving large parts of their business and presentation logic from the server-side to the client-side[1]. This trend was accelerated by the Internet's increasing speed and coverage for mobile devices as well as advances in standards, which made the Web more dynamic in the last couple of years [1,5,13]. Development methodologies like progressive enhancement have additionally 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 semantic Hypertext Markup Language (HTML). Enhancements such as advanced

[1] While in our scenarios clients are mostly represented by browsers, other applications e.g., Firefox OS or WebView Components in Android and iOS are also valid clients.

Cascading Style Sheet (CSS) or JavaScript are layered unobtrusively on top of HTML [12]. Web-enabled devices like search engines or gaming systems do only provide limited or no support for further enhancements. By following this development methodology, they are also enabled to access corresponding Web applications. New devices like latest browsers benefit from this additional technological layer by improvements to style and interaction.

Progressive enhancement can be accomplished by duplication, i.e., realizing parts of the application for both server and client. Duplicating the business and presentation logic entirely, however, is inadequate as it decreases not only development efficiency, but also makes the application more error-prone and harder to maintain. Besides the problem of technically establishing a server/client compatible codebase[2], we identified four key differences between both sides that have to be treated separately: view, routing, data access and state transfer.

Unlike the view on the server-side that is commonly string-based and generated on-demand, the client-side view is based on the Document Object Model (DOM). This difference also affects the routing because the view on server-side is built from scratch on every request, i.e., on every route. In addition to this, the business logic of routes cannot be reused for the client-side without further regard. As the client-side is generally more vulnerable to malicious manipulations, unveiling data exchange logic to the client-side has potential security issues. The state established during the initial request consisting of data, view bindings and precompiled fragments acts as origin and as basis for all further user interaction. As a consequence, the state needs to be transferred from the server to the client in order to continue on the client where left off on the server.

In this paper we present a framework providing both dynamic functionalities and progressive enhancement without having to implement an application twice on server and client. We focus on coping with characteristic client/server differences using a technically compatible codebase, supporting client- and server-side generated views, implementing a server/client compatible routing as well as establishing mechanisms for data access and state transfer.

This paper is organized as follows: We begin in Section 2 with an example demonstrating the features of our framework. Section 3 provides an overview of our approach and describes the resulting framework. We detail the routing, the view, the data access and the state transfer. In Section 4 we evaluate our framework. We position our approach to related work in Section 5 and conclude our work in Section 6.

2 Example

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

[2] The term "codebase" refers to the whole source code of an application.

Consider a simple application for managing tasks and documents as shown in Figure 1. 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.

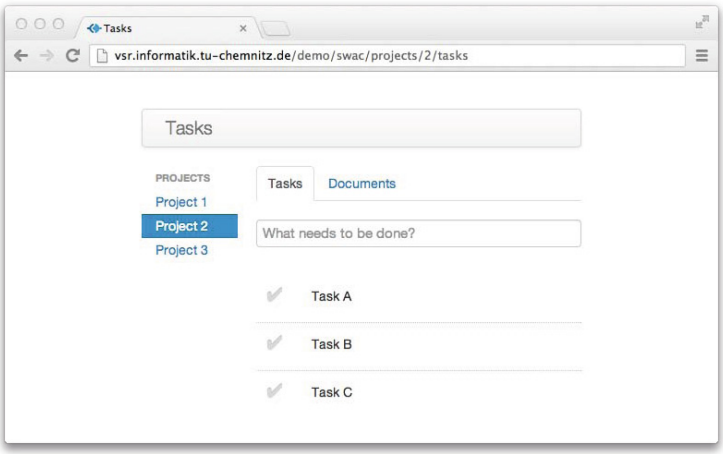

Fig. 1. Screenshot of Sample Web application

Our framework enables developers of such a task & document management application to combine these five distinct routes into one, which itself is split into five hierarchical pieces, as shown in Figure 2. Therefore, we 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).

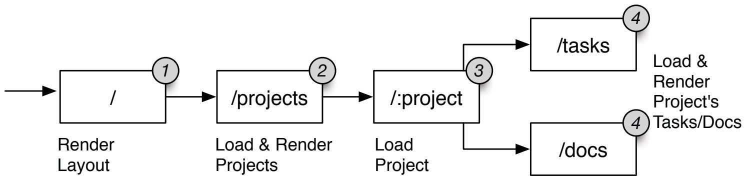

Fig. 2. Route Hierarchy of Sample Application

On the client-side, this separation relieves us from the need to execute the whole route once a project is selected. Furthermore, the separation enables moving through a route step by step to execute only the necessary parts of a route, e.g., projects, tasks or documents. The presentation logic is responsible for reflecting changes based on the business logic addressed by these routes. To achieve this without re-rendering, our 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 3, 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.

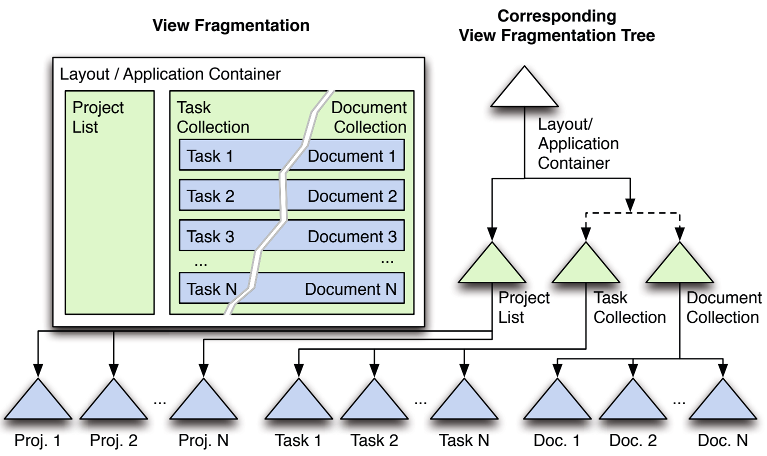

Fig. 3. View Fragmentation of Sample Application

Having implemented the routes and the view as described, requesting the list of projects from the server would be as follows: each route up to the projects route is executed automatically by our framework, and 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 precompiled templates[3] 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 decoupled from the server-side, i.e., only data access and manipulation operations have to involve server communication. It is important to note that in our framework, the logic responsible for the 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, our framework automatically provides an

[3] A precompiled template is a JavaScript function responsible for creating HTML for the data provided in the fragment.

appropriated 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. Additionally, the client requests the precompiled 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/`

3 Approach for Sharing a Web Application's Codebase

Our approach for Sharing a Web Application's Codebase (SWAC) establishes server/client compatible Web application codebases and addresses the differences between server- and client-side. SWAC is designed to execute only necessary parts of an application's business logic 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. These parts are automatically updated once their underlying data changes. It achieves data security by an additional layer between the business logic and the 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. To technically establish a server/client compatible codebase, SWAC is entirely implemented in JavaScript using Node.js on the server-side. The following subsections detail both the theoretical background and the actual implementation of the SWAC approach.

3.1 Routing

A route hierarchy is an essential part of the business layer as it defines the relationship of routes in an application to each other. SWAC utilizes such a 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 [4]. 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 the necessary parts to reflect the changes required to navigate from one route to another on the client-side. To handle scenarios of routes requiring logic that is incompatible to 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 framework covers three different routing schemes, where each schema is distinct in terms of handling the route hierarchy tree: 1) the route is executed on the server-side, 2) the client is initialized to take off the application and 3) 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 descendent 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 4.

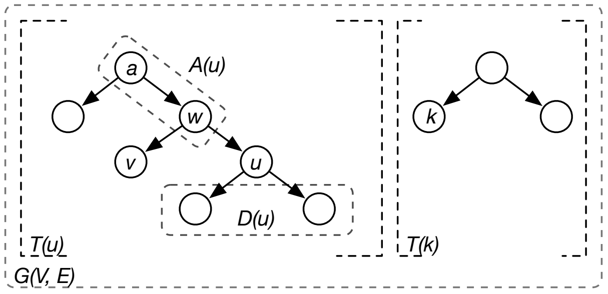

Fig. 4. Routing Terminology

Server-Side Execution. The server-side of our framework is stateless, i.e., 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 the execution of 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, our framework can execute most of the application's functionality decoupled from the server on the client-side. Therefore, the client-only parts of the current route are executed. This is done by applying the same method as used for the 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 the routes that are responsible for the changes between two user interactions are executed. There are four subscenarios for the client-side execution. They take different positions of the target route into account. If the target route v is not an element of the tree $T(u)$ of the starting position u, the execution works the same way as on the server-side. That is, v and all ancestors $A(v)$ are executed in their appropriate order. Otherwise, the 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 the 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:

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

Example. Consider the route from our sample application (Figure 2). There we use a parameter (`:project`). The SWAC framework handles each parameter as a distinct branch of the route hierarchy (cf. Figure 5). This is necessary for correct routing. For instance, switching from one project tasks (u) to the tasks of another project (v) requires executing the `/projects/:project` route again for the new project. The routes that have to be executed in this scenario are highlighted in green in Figure 5.

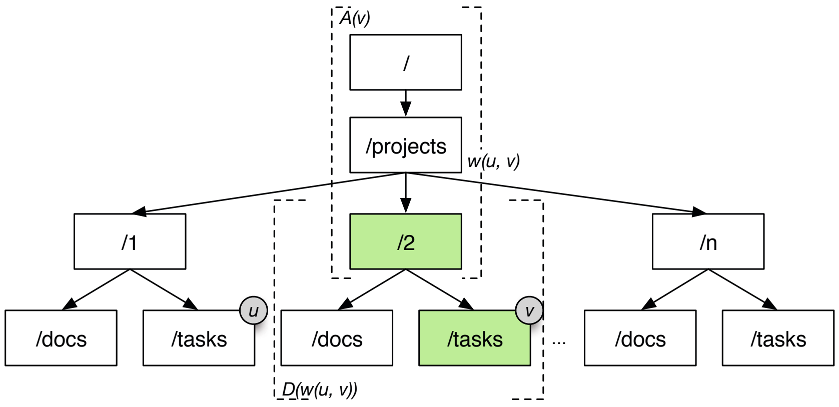

Fig. 5. Routing example $T(u)$ (1)

Figure 6 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.

3.2 View

The view on the client-side facilitates executing only the necessary presentation logic instead of re-rendering the entire view on every data change. While every

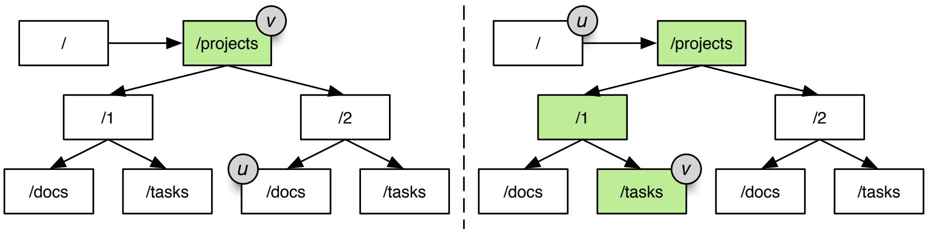

Fig. 6. Routing example $T(u)$ (2)

single part of a view can be directly updated on the client-side via DOM, this is impossible using 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 [10].

As we are interested in creating an efficient and compact partition similar to DOM for string-based templates on the server-side, we utilize embedded JavaScript. We achieve the fragmentation by wrapping parts of the template into an appropriated block expression, as exemplary shown below:

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

The re-rendering of 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 3 require knowledge about the position of the fragment. For this reason, the position of a fragment needs to be tracked. This could be easily accomplished by wrapping fragments into HTML elements, which are identified and accessed via IDs. However, this approach has several issues, e.g., HTML elements like `<title>` do not support child nodes [2]. HTML table rows are another example for elements, which 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, which is allowed to reside inside every HTML element, fits our purpose. Since a comment cannot wrap content that should be rendered, they have to act as start and end markers for a fragment, as exemplary shown below:

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

On the server-side, these comments are just parts of the rendered string and they only share their syntax with a DOM comment. This requires initializing the fragment positions once the DOM is built on the client-side. That is, as soon as the client builds the document, a method has to detect all relevant comments and assigns them to their 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
}
```

The bindings ensure that fragments update themselves on appropriate data changes. They have to be created once the data got accessed. To achieve this without making the API inconvenient, properties used inside a fragment have to be enabled to interact with the fragment they are accessed from. The SWAC framework uses the JavaScript function's caller property for this purpose. This property points to the object which called the function the property was accessed from [8]. We achieved this by making each data property a getter that binds itself 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
})
```

Although this Function.caller property is not part of the ECMA standard [3], it is currently supported by all major browsers (Firefox, Safari, Chrome, Opera and Internet Explorer) [16].

3.3 Security

Since the goal of this framework is to reuse an application's codebase between server and client every part of the application's logic is shared between server and client unless it is explicitly declared as server-only logic. Nevertheless, the communication between the business logic and the database is always executed on the server-side. We achieved this by splitting the business tier into two layers: the service layer and the business layer. The service layer provides the API for the communication 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 7. That is, authentication and authorization logic for data access is always executed in a privileged environment on the server-side.

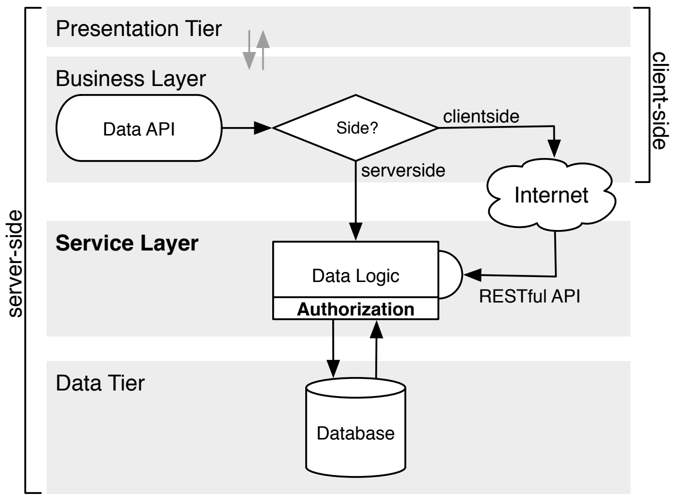

Fig. 7. Service Layer

Due to the fact that the actual authentication/authorization logic is not predefined, the SWAC framework provides hooks for injecting custom logic. This facilitates implementing such logic using already existing Node.js packages, e.g., for OAuth or OpenID. 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(request, user) {
      return request.user.id == user.id
    },
    post: function(request) {
      return true
    }
  })
})
```

There are two options for establishing route security. First, avoiding route sharing is the most secure way for routes referring to proprietary algorithms. We suggest only using this option if absolutely necessary because the benefit of the SWAC framework results from the ability of sharing code. Second, for shared routes, the SWAC framework provides hooks for both authentication and authorization logic. Executing this logic on the client-side is useless because of vulnerability to malicious manipulations. SWAC enables developers to divide applications into several areas, as shown in Figure 8. Since these areas are isolated from each other, navigating between them triggers requests to the server. A client requesting a route of such an area must pass the authentication/

authorization logic attached. This is necessary for the client to obtain the area's bundle (the JavaScript files that contain the business logic of this area) and to call the route at all.

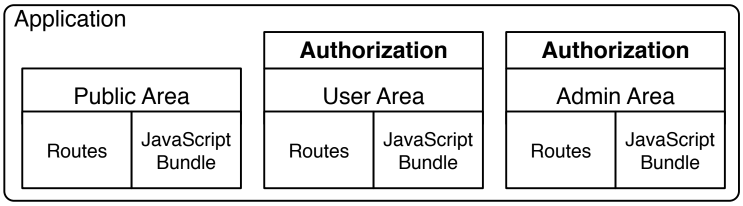

Fig. 8. Application Areas

An exemplary access control definition for such a route is listed below:

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

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.

3.4 State Transfer

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

- data contained in models and collections
- fragment positions
- events and their listeners
- precompiled 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 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 precompiled form.

Transferring the state requires serialization. Possible formats for this purpose are textual ones, like the JavaScript Object Notation (JSON) and the Extensible Markup Language (XML), or binary ones, like the MessagePack[4] and

[4] `http://msgpack.org/`

the Protocol Buffers[5]. There is no direct support for buffers in browser-based JavaScript [9]. Deserializing a buffer format on the client-side would be slow and error-prone. Therefore only textual formats are qualified for a server/client compatible serialization and deserialization. As JSON is supported by JavaScript directly [3], it is the textual format we use.

Regardless of this format choice, 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 [3]. 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. The framework replaces further occurrences of objects with a JSONPath[6] to their first occurrence. To serialize functions we avoid closures and use string representations of functions via their `toString()` method. For restoring objects created built using a constructor, we implement the service locator pattern. Constructors are registered to the service locator and all objects created with such a registered constructor are tagged appropriately. This enables restoring such objects on deserialization. These approaches in combination are a powerful toolkit to deserialize/serialize complex JavaScript objects.

4 Evaluation

In order to demonstrate the benefits of our solution, we made a small coding contest: the development of a simple single-page task application capable of adding, removing, editing and changing the state of tasks. For this purpose, we compared a combination of a common back-end and a common front-end framework with the SWAC framework. As a challenge for our approach, we chose Rails[7] as the back-end framework, which facilitates the development of back-ends due to its scaffolding functionalities. For front-end implementation, we used Backbone.js[8]. Both Rails and Backbone.js are necessary to provide the features the implementation with SWAC provides. Although a Rails-only implementation using its JavaScript Adapter would allow fast development, only a few functionalities are supported. This might be sufficient for a simple application like this, but is inappropriate for the use cases our framework is aiming at, e.g., execution of business logic on the client-side or automatic view binding.

While developing the application with Rails/Backbone.js required an average time of 52 minutes, the task was done on average in 22 minutes with SWAC. As shown in Figure 9, the development of this simple task application is about 60%

[5] `http://code.google.com/p/protobuf/`

[6] `http://goessner.net/articles/JsonPath/`

[7] `http://rubyonrails.org/`

[8] `http://backbonejs.org/`

faster when SWAC is used in comparison to the use of different frameworks for back- and front-end. As a result of Rails maturity and scaffolding functionalities, the back-end only development is indeed faster than using SWAC.

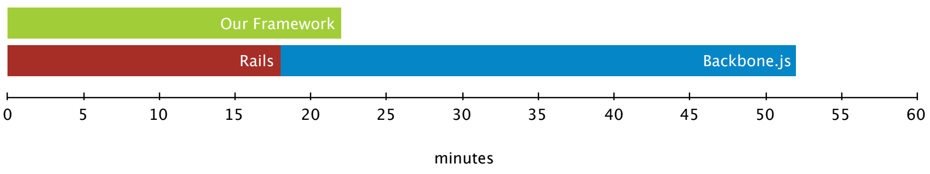

Fig. 9. Time comparison for Rails/Backbone.js vs. SWAC framework

The comparison of the amount of source lines of code (SLOC) necessary to implement the application shows a significant difference, as illustrated in Figure 10. While the implementation with SWAC only required about 60 SLOC, Rails and Backbone.js required at least 190 SLOC (Rails: 75, Backbone.js: 115).

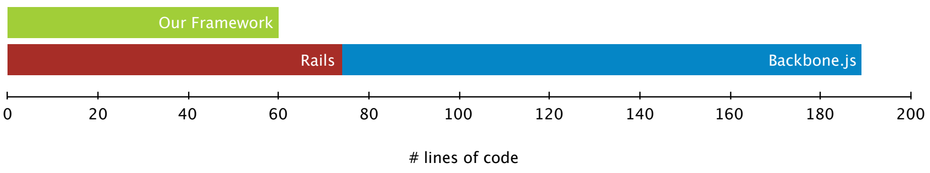

Fig. 10. SLOC comparison for Rails/Backbone.js vs. SWAC framework

The SWAC framework and Rails require nearly the same SLOC. However, Backbone.js' DOM based view updates needed some extra lines of code for their implementation. Although the evaluation shows that the SWAC framework improves the development efficiency of dynamic and "progressively enhanced" Web applications, both comparisons should only be considered as an orientation. This is because time consumed as well as lines of code required for the development depend on the knowledge and experience with a framework.

5 Related Work

In this section, we analyze the work related to our solution. This analysis focuses on frameworks that aim for creating dynamic Web applications through establishing a technically compatible client/server codebase. The key differences between server and client, we identified in Section 1, are used as main analysis criteria. The analysis is based upon the following criteria:

- Technically compatible codebase
- Progressive enhancements
- State transfer from server- to client-side

– Data logic separation and access control
– Business logic routing
– Intelligent presentation logic, e.g., automatically updating view fragments

Derby [14], as a WebSocket-based framework, focuses on real-time and collaborative applications. Despite its facility to completely render the result of the first request, it is not progressively enhanced. Since Derby uses WebSockets for all data manipulations, submitting data without WebSockets is impossible. A Web application's client-side implemented with Derby cannot be executed left-off from the server because the state remains on the server. However, WebSockets can be used to propagate state changes. Derby solves separation of data logic and access control similar to our solution [11]. While business logic is only executed on the server-side, Derby dynamically initiates routing from the client-side.

As another framework in this context, Meteor [6] can create technically compatible Web application codebases using JavaScript. However, Meteor does not achieve complete compatibility between server- and client-side because the framework lacks built-in routing functionalities, i.e., business logic is triggered through DOM events. 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 [7] taking Google's AJAX crawling specification into account. While this is beneficial for search engines, for common visitors Meteor renders the Web application only on the client-side. That is, Meteor is not suitable for developing progressively enhanced Web applications. As Meteor Web applications are executed on client-side only, there is no state to be transferred from the server-side.

Compared to the frameworks analyzed so far, the API of Yahoo! Mojito [15] is quite different from familiar back-end frameworks in the sense that it does not provide a homogenous data API for both the 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., HTML5 History [2] is not used. Mojito presentation logic allows updating view fragments on data changes.

Although the analyzed frameworks allow sharing parts of the codebase between server and client, they do not offer facilities to automatically create progressively enhanced Web applications.

6 Conclusion

With the SWAC approach we provided a solution for sharing a 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. As a consequence of this action, we had to adjust the presentation logic to be compatible, too. Therefore, we added a mechanism to split the view into fragments, which are updated automatically once underlying data changes. In addition to these contributions, we integrated a facility into our framework allowing the client to seamlessly take over the application after the state was transferred automatically from the server to the client. While the data exchange logic is not shared by the SWAC framework because of security concerns, we enabled the client to unobtrusively proxy its database calls through the server.

In future work, we intend to perform an evaluation with a larger set of frameworks. We assume that the evaluation results will help to identify advantages as well as shortcomings of our current solution that need to be addressed in further contributions. We plan implementing modularity improvements, e.g., making fragments compatible to third party template engines or enabling authorization per data property. In addition to these enhancements, we will investigate collaborative editing scenarios asking for facilities such as data push.

Acknowledgment. This work was funded by the European Commission (project OMELETTE, contract 257635).

References

1. Belson, D., Leighton, T., Rinklin, B.: The State of the Internet, vol. 5(3) (2012)
2. Berjon, R., Leithead, T., Doyle Navara, E., O'Connor, E., Pfeiffer, S.: HTML5 Specification, Editor's Draft 6 (October 2012)
3. Ecma International: ECMA-262 ECMAScript Language Specification 5.1 Edition (2011)
4. Masinter, L., Berners-Lee, T., Fielding, R.T.: Uniform Resource Identifier (URI): Generic Syntax (2005), http://tools.ietf.org/html/rfc3986
5. Meeker, M., Wu, L.: 2012 Internet Trends (2012)
6. Meteor Development Group: Meteor, http://docs.meteor.com/
7. Meteor Development Group: Meteor - Search engine optimization, http://meteor.com/blog/2012/08/09/search-engine-optimization
8. Mozilla Developer Network: Function.caller https://developer.mozilla.org/de/docs/JavaScript/Reference/Global_Objects/Function/caller
9. Mozilla Developer Network: JavaScript typed arrays, https://developer.mozilla.org/en-US/docs/JavaScript_typed_arrays
10. 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)
11. Noguchi, B., Smith, N.: Racer Access Control, https://github.com/codeparty/racer/tree/master/lib/accessControl
12. Parker, T., Jehl, S., Wachs, M.C., Toland, P.: Designing with Progressive Enhancement: Building the Web that Works for Everyone. New Riders Publishing (2010)
13. Smith, A.: Cell Internet Use 2012 (2012)
14. Smith, N., Noguchi, B.: Derby, http://derbyjs.com/
15. Yahoo! Inc.: Yahoo! Mojito, http://developer.yahoo.com/cocktails/mojito/
16. Zaytsev, J.: ECMAScript extensions compatibility table, http://kangax.github.com/es5-compat-table/