KevoreeJS: Enabling dynamic software reconfigurations in the Browser

Maxime Tricoire, Olivier Barais, Manuel Leduc, Johann Bourcier
INRIA, IRISA, Université de Rennes 1
Rennes, France
firstname.name@irisa.fr

François Fouquet, Grégory Nain, Ludovic Mouline
SnT, Luxembourg
firstname.name@uni.lu

Gerson Sunyé
INRIA, Université de Nantes
gerson.sunye@inria.fr

Brice Morin
Sintef
Oslo, Norway
bmorin@sintef.no

***Abstract*— The architecture of classic productivity software are moving from a traditional desktop-based software to a client server architecture hosted in the Cloud. In this context, web browsers behave as application containers that allow users to access a variety of Cloud-based applications and services, such as IDEs, Word processors, Music Collection Managers, etc. As a result, a significant part of these software run in the browser and accesses remote services. A lesson learned from development framework used in distributed applications is the success of pluggable architecture pattern as a core architecture concept, i.e., a Software Architecture that promotes the use of Pluggable Module to dynamically plug. Following this trend, this paper discusses the main challenges to create a component-based platform supporting the development of dynamically adaptable single web page applications. This paper also presents an approach called KevoreeJS based on models@runtime to control browser as component platform which address some of these challenges. We validate this work by presenting the design of a dashboard for sensor based system and highlighting the capacity of KevoreeJS to dynamically choose the placement of code on the server or client side and how KevoreeJS can be used to dynamically install or remove running components.**

Keywords. *Web Engineering, Dynamic component model, Single page application*.

I. Introduction

Traditional desktop-based productivity software are moving to the Cloud. Nowadays, the browser is essentially an application container that allows users to run a single page application to access a variety of Cloud-based applications and services, such as an IDE, a Word processor, a Music Collection Manager, etc. A large offer of generative framework now proposes to create the skeleton of such modern web applications, such as cite JHipster[1], Mean.js [2], Ionic[3], or KeystoneJS[4]. These stacks generally use frameworks for developing the client part following a family of MVC pattern such as AngularJS [1], EmberJS [2], Backbone [3], Durandal [4], and React [5].

A lesson learned from classical application development frameworks used for building IDEs, word processor,s operating systems, and music or video players is the success of the use of the pluggable or composite architecture pattern [6], [7] as a core architecture concept, i.e., a Software Architecture enabling dynamic module plugging to adapt the application functionalities to project requirements. In this trend, the OSGi framework specification [8] has been widely adopted by the Eclipse community and forms the basis of the Eclipse Runtime since Eclipse 3.X. The OSGi framework is a system module and service platform for the Java programming language that implements a complete and dynamic component model. Components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot.

The design of highly configurable web applications requires the support of such pluggable architecture based on Component Based Software Engineering within the Browser. Current frameworks such as AngularJS, Ember, Backbone, Durandal, or Eeact focus on a clear architecture of the web applications following a single page application principle [4] but does not provide a solution to dynamically reconfigure a running application. Due to the increasing complexity of Web Applications and based on the experience of other application containers, it is now required to support the evolution of a software artefact or the installation of a new software artefact without reloading the web page.

This paper highlights the challenges that arise when supporting a pluggable architecture pattern to support dynamic reconfigurations of a single page application. It also presents KevoreeJS in details, our approach to provide such a platform, and discusses its current limitations. We validate this work by showing how KevoreeJS can help to dynamically change Client/Server code partitioning in a dashboard for sensor-based system. Through this use case, we motivate the need for a dynamic module system for the browser, similar to OSGi for the JVM.

The remainder of this paper is the following. Section II presents the main challenges for designing a module system for the JavaScript programming language that implements a complete and dynamic component model. Section III shows an overview of KevoreeJS and illustrates the main concepts through a motivating example of a dashboard for sensor based systems. Section IV and V discuss related work and

[1]https://jhipster.github.io/
[2]http://meanjs.org
[3]http://ionicframework.com/
[4]http://keystonejs.com/

present ongoing work.

II. Main challenges for the use of the browser as a container for reconfigurable single page web applications

A single-page application (SPA) is a web application that fits on a single web page, providing a more fluent user experience similarly to a desktop application. This architecture style is now a standard way of designing modern Web applications, but still lacks support for dynamic reconfigurability. This section discusses eight important challenges to enable dynamic reconfigurability in a SPA.

A. Automatic provisioning of component implementation and third-party libraries

Component-based distributed systems are known to be hard to deploy for two main reasons: the complexity of their structure and the complexity of the deployment tasks. Most of the current tools are not able to properly address these challenges because the underlying component dependency descriptions lack expressiveness. Web modules does not avoid this drawback. Indeed, if most of them use tools to manage dependencies at design time such as node package manager (npm[5]) or Bower[6], these tools are generally not used to enable the dynamic loading of third-party libraries at runtime. Besides, deploying components in the web is generally done when the user accesses a web page. In Web applications, updates are made by downloading new software artefacts when the user refreshes the whole page. In a SPA, all necessary code–HTML, JavaScript, and CSS– is retrieved with a single page load, usually in response to user actions. The page does not reload at any point in the process, nor does control the transfer to another page, although the location hash can be used to provide the perception and navigability of separate logical pages in the application. SPA Frameworks provide solutions to limit the time required to load the whole page by providing support for dynamic updates of page fragments. However, this mechanism does not support the update of existing libraries. The first challenge to build dynamically adaptable single page web applications is to manage the dynamic deployment of new components and their third-party libraries without perturbing the running applications. It includes also a correct management of libraries dependencies based on standard tooling used by developers such as Bower and npm.

B. Type System

The second main challenge is to make web components contract-aware [9]. If the CBSE community agrees to trust a component, we must be able to determine how this component will behave. The Web domain is using radically different development method than the one used for mission-critical applications. Web developers generally use JavaScript as a programming language without providing a clear interface for their components. A component model for Web Application must force the developer to declare component interface and propose at minimum a basic level of contract to support duck typing verification when assembling components [9]. These contracts can also be used to check that the component interface fully conforms to its implementation if developers use programming language such as TypeScript [10] or Dart [11] to implement their components.

C. User Interface composition

The next challenge is to provide a way to compose the User Interface (UI). If HTML-based interface technologies enable end-users to easily use various remote Web applications, it is difficult for end-users to express complex compositions. Currently, the main Web actors provide frameworks to express complex compositions, such as React [5] from Facebook, which provides an adaptation algorithm to compute the minimum diff in the DOM. Google's Polymer [7] is a similar approach with stronger emphasis on reusability. RiotJS [8] also provides a lightweight framework for UI composition, however limited to static composition. None of them aims at providing dynamically reconfigurable user interface composition mechanisms for the deployed components.

D. Security handling

Providing reconfigurations support from an external manager leads to several security issues. If the SPA can receive a new configuration, automatically download components, and update the running ones, it becomes easy to disseminate malware to any running application. The model which acts as a blueprint of the deployed runtime system should embed mechanisms to ensure authentication and the data integrity of the system. This kind of issue is not new to the world of web development and existing principles can be applied such as https protocols, Role-based access control...

E. Search engine optimization

Because of the lack of JavaScript execution on crawlers of some popular Web search engines, SEO (Search engine optimization) has historically presented a problem for public facing websites wishing to adopt the SPA model. In fact, modern Web search engine such as Google support the fact that a SPA can dynamically generate new contents [12]. However, in the case of a dynamic component model, the crawler cannot know in advance all the components that can be installed in a web application. Consequently, search engines that only know the first configuration of the application cannot correctly index such a dynamically

[5]https://www.npmjs.com/
[6]http://bower.io/
[7]https://www.polymer-project.org/1.0/
[8]http://riotjs.com/

reconfigurable SPA. This require to integrate new metadata to define, for example, all the components that can be installed on a running web app. In that case, a search engine can do the hypothesis of the closed world to know all the potential configurations of a Web application (even if it can become combinatorial).

F. Client/Server code partitioning

A dynamic component model for SPA must enable dynamic client/server code partitioning. In that sense such framework must provide abstractions for templating definition and rendering, abstraction for streams queries, to let a developer dynamically decide if the component should be executed within the browser or on the server side. As it exists lots of technical stack on the server side which leverage various programming languages, the component model must provide abstractions to define the component behavior that can be generated on this different technical stacks. This problem might be simplified by making the hypothesis that the server also used JavaScript. In that case, initial solutions exist[9] to share the same code for the client and the server side.

G. Browser history

With a SPA being, by definition, "a single page", the model breaks the browser's design for page history navigation using the Forward/Back buttons. The traditional solution for SPAs has been to change the browser URL's hash fragment identifier in accord with the current screen state. The HTML5 specification has introduced *pushState* and *replaceState* providing programmatic access to the actual URL and browser history. For a dynamic component model, the main challenge is to defined how to aggregate the states of all deployed components. The second challenge is to specify whether a user request for coming back to a previous state should include or not the configuration history. There is in fact several histories in a dynamically reconfigurable SPA: history of actions, history of the cached data, etc.

H. Analytics

Analytics is a common concern for web applications. The goal is to be able to track and report website traffic and user activity. Tools such as Google Analytics rely heavily upon entire new pages loading in the browser, initiated by a URL change. As SPAs do not work this way, and in particular if we dynamically load new components, analytics packages are not able to take these actions into account. The new HTML5 history API allows to add page load events to an SPA. As the component web developers is in charge of using correctly this API, the challenge is then to avoid missing reports and double entries.

Analytics, browser history, and Search engine optimization are clearly challenging due to the dynamic requirements.

[9]http://isomorphic.net/ references a set of solutions for client/server partitioning

I. Speed of initial load (overhead@runtime)

The last challenge is to guarantee that the flexibility offered by the use of a pluggable architecture does not introduce too much overhead when we start the SPA. Such an approach must minimize the download of unused features.

III. KevoreeJS: A runtime for reconfigurable single page applications in the browser

This section present a module system for the Browser called KevoreeJS . In particular, KevoreeJS implements a dynamic component model for SPA. This component model currently addresses only a part of the challenges discussed in Section 2.

A. Motivating Scenarios

To illustrate the framework, we consider a simple dashboard for sensor-based system in which, it is required to install/uninstall a new web widget when a new sensor appears/disappears. In such system, three kinds of reconfiguration have to be managed: i) the installation and retrieval of software package (javascript code), the instantiation of components, the components parametrization (to bind components through ports, the setup of parameters, ...), the component life-cycle management. ii) the client/server code partitioning to select if some components managing complex event queries must be executed on the server side or within the browser. iii) the selection of a specific deploy unit depending on the browser type, its devices and its screen layout. A screenshot of the results of such an application is presented in Figure1. This dashboard provides information regarding values that can come from a set of nodes such as the one described in Figure 2.

B. KevoreeJS overview

KevoreeJS is built on top of the models@runtime paradigm. Models@runtime denotes model-driven approaches aiming at taming the complexity of dynamic adaptation. It basically pushes the idea of reflection [13] one step further by considering the reflection layer as a real model that can be used to drive the system deployment and (re) configuration: "something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality". In practice, component-based (and/or service-based) platforms like Fractal [14], OpenCOM [15] or OSGi [8] offer reflection APIs which make it possible to introspect the system (which components and bindings are currently in place in the system) and dynamic adaptation (by applying CRUD operations on these components and bindings). While some of these platforms offer rollback mechanisms [16] to recover after an erroneous adaptation, the idea of models@runtime is to prevent the system from actually enacting an erroneous adaptation. In other words, the "model at runtime" is a

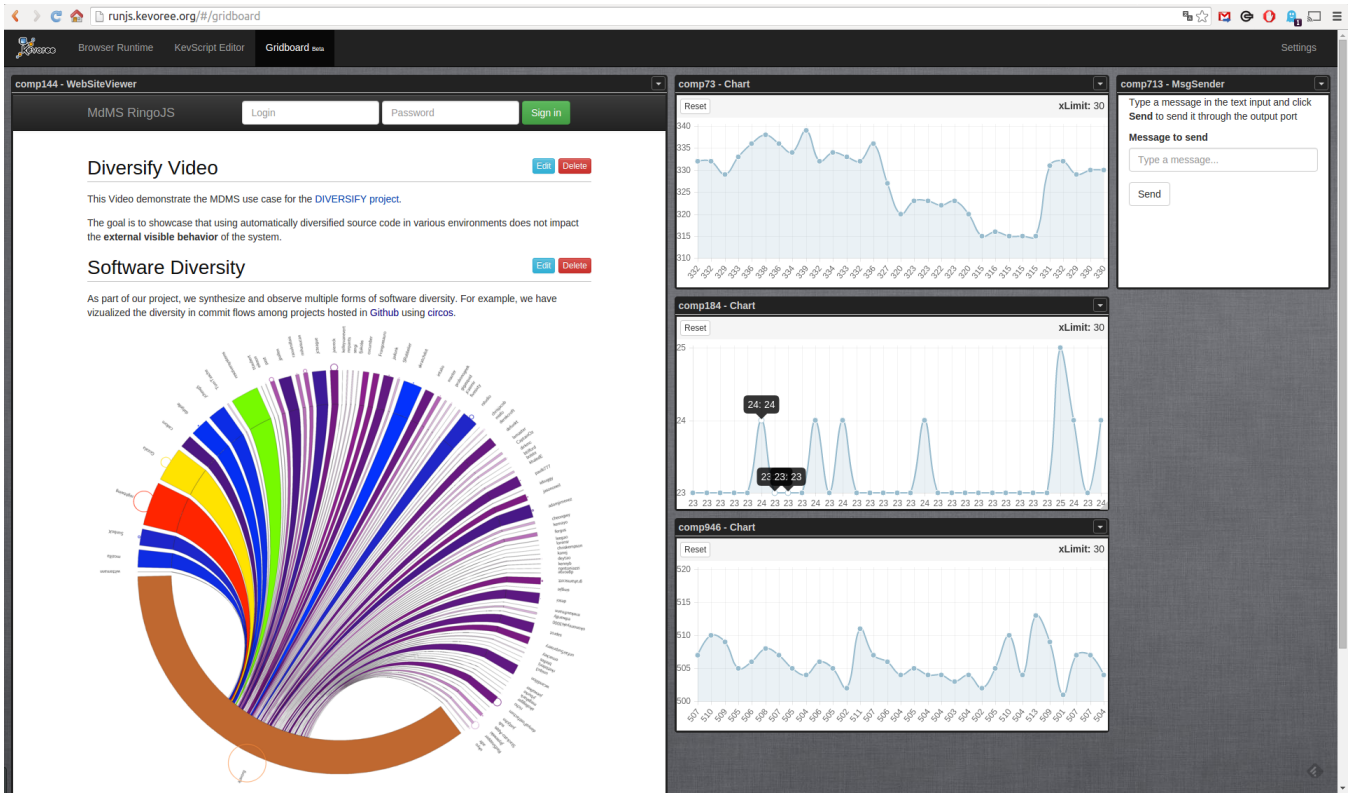

Figure 1. An example of dashboard for sensor-based applications

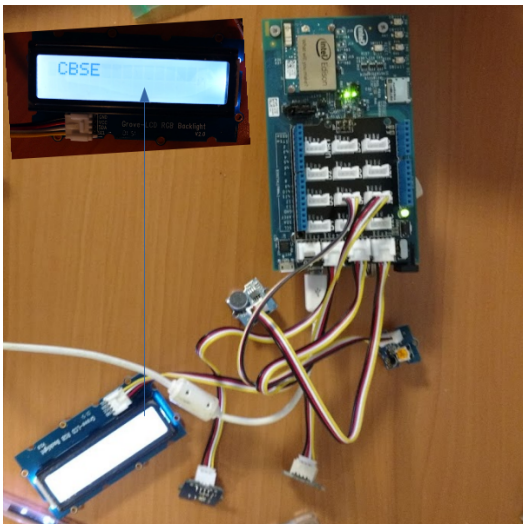

Figure 2. An example of sensor node based on Intel Edison

reflection model which can be uncoupled (for reasoning, validation, simulation purposes) and automatically resynchronized. This modelling layer provides a common abstraction to describe the system configuration. This model can be interpreted to decide which packages (component package and third parties libraries) must be installed or removed and which component must be instantiated and started. This modelling layer can also be modified and pushed to peers to trigger distributed reconfigurations.

KevoreeJS[10] implements the Kevoree component model. Kevoree is a dynamic component-based framework for distributed systems that follows the models@runtime paradigm and embeds a structural model of the distributed system. This model is used for two main purposes: (i) it represents a snapshot of the heterogeneous and distributed application state and (ii) it provides a language to drive the reconfigurations of this application. The Kevoree model embodies the following four main concepts of a distributed system.

1) The software **components** represent software units that provide the business value of the distributed system.
2) The **connectors** (called channels in Kevoree) are in charge of inter-component communications. A channel encapsulates and provides a particular communication semantic (e.g. synchronous or asynchronous, unicast or multicast, and may provide different contracts for synchronization and quality of services).
3) The **nodes** represent execution hosts for all other software entities such as components and channels. A node may represent a physical node or a virtual

[10]http://runjs.kevoree.org

machine. Nodes are application containers and they are in charge of the dynamic adaptations of its system part when a new model@runtime is received.

4) The **groups** are responsible for inter-node communications. In particular, a group provides semantics of dissemination and ensures consistency of models among nodes. On top of these abstractions, Kevoree provides a development model to design new components, channels, groups and nodes using different programming languages. It also comes with a set of tools for building dynamic applications (a graphical editor to visualize and edit configurations, a textual language to express reconfigurations, several checkers to validate configurations). Kevoree supports multiple execution platforms (e.g., Java, JavaScript, .NET, Android, LXC, Docker, FreeBSD, Arduino). For each target platform it provides a specific runtime container as a specific node type.

For supporting the SPA within the browser, we mainly reused the core of Kevoree component model which is written in Kotlin. This core contains the component model entities, tooling for loading and saving configuration models, tooling for detecting abstract actions (install a library, instantiate a component, bind a component to a channel, ...) and tooling to achieve runtime adaptations when the platform receives a new configuration model. As Kotlin provides a code generator for JavaScript, we can reuse this core library directly. As a consequence, to create KevoreeJS we mainly provide the concrete implementations of abstract actions in order to achieve concrete tasks within a running Browser core. We also provide a basic UI composition mechanism based on mashup. Each component comes with its own view that can be composed on the full SPA using mashup. To enable this mashup mechanism, we reuse framework such as AngularJS and angular-gridster. KevoreeJS reuses Bower for its static Web part, but it handles the dynamicity by downloading browserified[11] modules directly from the npm registry.

Finally we provide a simple development model in JavaScript and in TypeScript for developing components, channels, groups and nodes (see Section III.C).

KevoreeJS comes with a set of tools:

- a web-based architecture model editor,
- a Yeoman generator,
- a set of Grunt tasks to fully automate the component packaging and publishing
- and a runtime container to manage the dynamic deployment of third-party libraries.

The Yeoman generator creates skeletons to implement new components. It mainly provides six Software artefacts:

- *package.json*, which defines the set of dependencies for a component.
- *Gruntfile.js*, which provides a set of tasks to create component's deployment unit, to start a development environment and to publish the component.
- *browser/kevoree-comp-mycomp.html*; which is a HTML file used to describe the component's UI (it will be injected in a DOM node that is mapped to the component's AngularJS controller)
- *lib/MyComp.js*; which contains the model and the controller for the component.
- *browser/ui-config.json*; which stores the styles parameter, the layout parameters, the AngularJS module dependencies parameter, and the other JavaScript files to include.
- *kevs/main.kevs*; which contains an initial configuration for the application startup.

[11] http://browserify.org/

C. Development model

To illustrate this development model, we discuss the code of the last four artefacts for a simple component that provides the following graphical behavior (see Figure 3). This component simply shows its current state (Kevoree property) and lets the user change some properties (pure AngularJS UI properties).

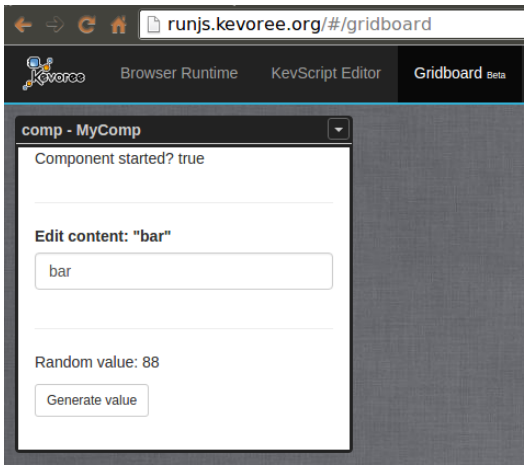

Figure 3. An example of KevoreeJS component

The view contains the following HTML code. It can access the controller's *$scope* properties such as the lifecycle state (*started*) or the business values (*foo* or *value* l.2 and l.6-11). It can also act on the component internal behavior in calling specific function declared in the component implementation (*genValue* l.12).

```
<div class="col-xs-12">
  <p>Component started? {{ started }}</p>                    2
</div>
<div class="form-group col-xs-12">                           4
  <hr />
  <label for="foo">Edit content: "{{ foo }}"</               6
    label>
  <input type="text" class="form-control" id="foo"
    data-ng-model="foo">
```

```
</div>
<div class="col-xs-12">
  <hr />
  <p>Random value: {{ value }}</p>
  <button class="btn btn-sm btn-default" data-ng-
    click="genValue()">Generate value</button>
</div>
```

Listing 1. Excerpt of the component view

The component implementation contains the following JavaScript code in the file *lib/MyComp.js*. Implementing a component consists in defining its life-cycle methods (*start* and *stop*, l.20-35), defining some parameters that can be configured from the configuration model (l.7-10), and defining an *uiController* function that will be called from the browser runtime to handle the view behavior using AngularJS. The binding between the view and the component model is obtained by sharing the *$scope* object, which refers to the application's model in the AngularJS technological stack. The developers can also define input and output ports for the component[12]). We propose an alternate development model using TypeScript as a main programming language.

```
var AbstractComponent = require('kevoree-entities').
    AbstractComponent;

var MyComp = AbstractComponent.extend({
  toString: 'MyComp',

  /* This is an example of dictionary attribute that you
     can set for your entity */
  dic_yourAttrName: {
    optional: false,
    defaultValue: 'aDefaultValue',
  },

  construct: function () {
    this.scope = {};
  },

  /**
  * this method will be called by the Kevoree platform
     when your component has to start
  * @param {Function} done
  */
  start: function (done) {
    this.log.debug(this.toString(), 'START');
    this.scope.started = true;
    // ...
    done();
  },

  /**
  * this method will be called by the Kevoree platform
     when your component has to stop
  * @param {Function} done
  */
  stop: function (done) {
    this.log.debug(this.toString(), 'STOP');
    this.scope.started= stop;
    done();
  },


  /**
  * this method is called by the Browser Runtime in order
     to retrieve
  * this component AngularJS UI controller
  */
  uiController: function () {
    return ['$scope', '$timeout', 'instance', function (
     $scope, $timeout, instance) {
      // this is your UI controller function
      // $scope content is available directly within the
     browser/kevoree-comp-foocomp.html file
      instance.scope = $scope;
      $scope.started = instance.started;
      $scope.foo = 'bar';
      $scope.value = parseInt(Math.random()*100);

      $scope.genValue = function () {
        $scope.value = parseInt(Math.random()*100);
      };
    }];
  }
});

module.exports = MyComp;
```

Listing 2. Excerpt of the component implementation

Finally, the development model contains a very simple configuration language to tweak the component's view. Developers can:

- provide local and/or external style sheets;
- provide local and/or external JavaScript files;
- specify a list of AngularJS modules to be bootstrapped;
- modify the layout metadata such as the default *width* and *height* for the component view.

This configuration is defined using a JSON formatted file named: *browser/ui-config.json* such as the one presented below.

```
{
  "scripts": [
    "my-local-script.js",
    "//cdn.scripts.foo/my-lib.js"
  ],
  "styles": [
    "my-local-style.css",
    "//cdn.styles.bar/my-style.css"
  ],
  "depModules": [
    "myNgModule"
  ],
  "layout": {
    "width": 2,
    "height": 1
  }
}
```

Listing 3. Excerpt of the component configuration

Finally, the last artefact contains the configuration model used to define the architecture of a distributed and heterogeneous application. A full documentation of this configuration is available on our website[13]. The following example illustrates a configuration with two nodes: the first one starts a Node.js runtime and the other one runs in the Browser runtime. Figure 4 presents a visual representation of this running configuration using the Kevoree Editor.

```
add node0, browser : JavascriptNode
add browser.comp : cbse.MyComp
add sync : WSGroup

attach node0, browser sync
```

[12]https://github.com/kevoree/kevoree-js

[13]http://kevoree.org/doc/

```
set sync.master = 'node0'

set browser.logLevel = 'DEBUG'
set browser.comp.yourAttrName = 'nonDefaultValue'
```

Listing 4. Excerpt of the application initial configuration

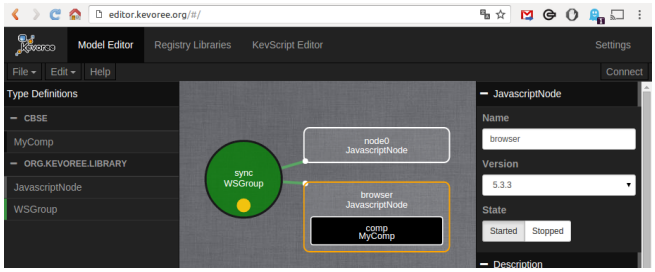

Figure 4. Visual editor that displays an excerpt of configuration

A complete description of the development model is available online[14,15].

D. Evaluation

To validate the proposed approach, we follow two ways. First we provide some figures on KevoreeJS in terms of line of code, number of reusable components available online, time penalty to initiate KevoreeJS when loading the pages. Then, we evaluate the approach regarding the challenges discussed in Section 2.

1) Quantitative evaluation: To implement the KevoreeJS core and the sensor based dashboard, we create 8 components, 3 channels, 3 groups. The core of KevoreeJS and these 14 software artefacts contain 56,671 LoC (15,584 has been manually written and 41,177 are generated from the model to automatically manage model entities, model load and serialization, ...The Figure 5 illustrates the configuration model of the applications using the Kevoree Web Editor [16]. The Figure 6 focuses in particular on the configuration of the SPA presented in Figure 1. All the code for this application is available on github. We provide a companion web page[17] to provide the links to all the software artefacts we use in this experiment.

Starting the sensor based application on a Chrome browser, running on top of a HP EliteBook 820 with Intel i7 processor, SSD hard drive and 16Gbytes of memory, takes 1533 ms from scratch. It takes 677ms when the components are available in the cache. The load time is mainly the time to download software modules on the npm registry server. Deploying the sensor based dashboard consists in writing a simple configuration.

[14]https://github.com/HEADS-project/training/tree/master/2.Kevoree_Basics
[15]https://github.com/kevoree/kevoree-js.d.ts
[16]http://editor.kevoree.org
[17]http://github.com/kevoree/CBSE16KevoreeJS

2) Qualitative evaluation : To discuss the strength and the limitations of KevoreeJS, we present the current implementation choices regarding the challenges discussed in Section 2.

1. KevoreeJS provides an initial solution to automatically provision component implementations and third-party libraries. It reuses npm's registry to download the component implementations. KevoreeJS does not provide isolation between components, such as sandboxing mechanism. As a result, a faulty KevoreeJS component may impact the rest of the SPA. However, components' UIs are displayed using iframes which helps sandboxing the CSS.

2. KevoreeJS provides a basic type system inherited from the Kevoree component to check component assembly description. The provided development environment also support TypeScript to ensure interface compatibility between components.

3. KevoreeJS provides an initial UI composition mechanism based on mashup. If this UI composition mechanism is sufficient for a sensor based dashboard, currently KevoreeJS does not provide advanced UI composition mechanism.

4. KevoreeJS provides an initial security solution to restrict peers access to the model configuration. This mechanism is currently limited in its ability to manage role based access rules on the configuration models. KevoreeJS relies on registries, which act as providers of the model characteristics. On this aspect, KevoreeJS has to deal with the same issues of trustability than the Linux distribution providers (e.g debian's apt, arch's linux pacman...) have encountered in the past (i.e. what happened if an attacker corrupt a registry and reference a malware?). In its current implementation, by default, Kevoree is unsecured.

5. Search Engine Optimization is still an open problem. We do not provide new concepts in the KevoreeJS to solve this issue.

6. We support client/server partitioning. Among 14 modules that have been developed for the motivating scenario, 11 are cross-platform Java-JavaScript components, consequently they can run on the server side or on the client side. One of the component, which is in charge of doing complex event processing is generated from ThingML behavioral description. ThingML [17] provides code generators for Java, JavaScript, and C. Consequently this component can be deployed dynamically on the Browser or within a Java or JavaScript Kevoree runtime on the server. The following code illustrates an example of a component behavior that performs a complex event processing on a stream of float values and provides every five second an average of this stream of values. Thanks to ThingML code generators, we could obtain C, Java or JavaScript implementation of this behavior. Consequently, an architect can dynamically decide if the query must be deployed closed to the sensors, within a gateway, or within the browser.

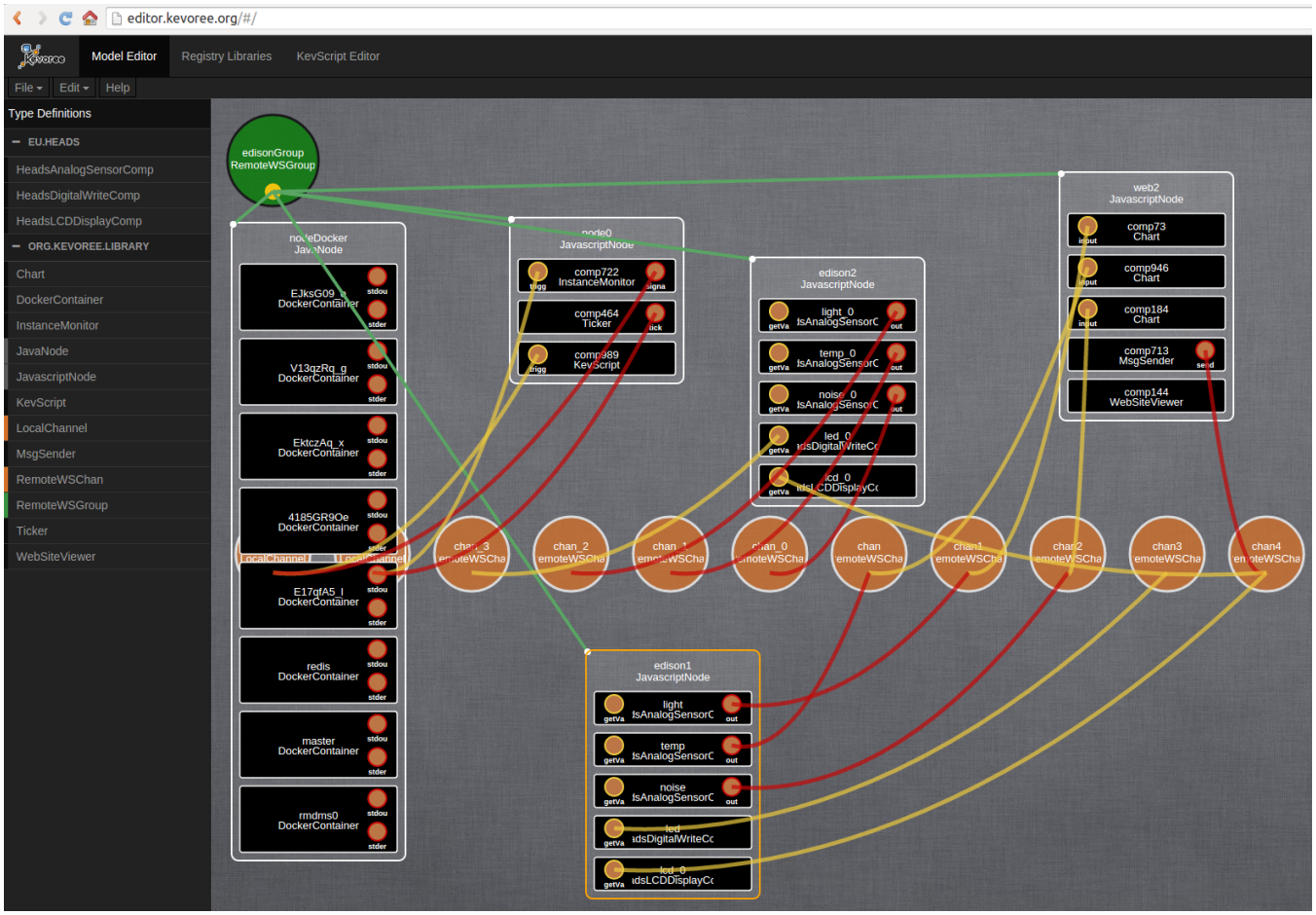

Figure 5. An example of dashboard for sensor-based applications

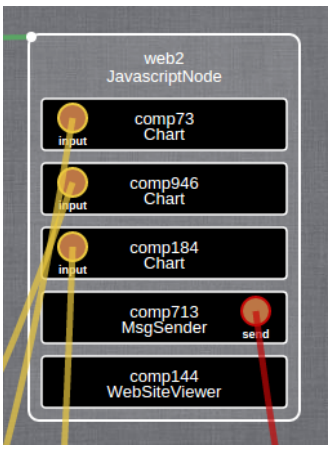

Figure 6. An example of sensor node based on Intel Edison

```
thing CepComp {
  message input(v : Float);
  message output(avg : Float);

  required port cep {
    receives input
    sends output
  }

  function avg(values : Float[]) : Float do
    var i : Integer = 0
    var appen : Float = 0
    while(i < values.length) do
      appen = appen + values[i]
      i = i+1
    end
    return appen / i
  end

  stream computeAvg do
  from evt : [cep?input]::timeWindow(5000, 5000)
  select avg : avg(evt.v[])
  action cep!output(avg)
  end

  statechart behavior init Init {
    state Init {
      on entry print("Starting")
    }
  }
}
```

Listing 5. Excerpt of a component implementation defined using ThingML

7. We take the design decision in KevoreeJS that the browser history only affect the component states. We do not use Browser history API to go back to a previous SPA configuration.

8. Improving analytics in dynamically adaptable SPA is still an open problem in our current implementation of KevoreeJS

9. Finally, the KevoreeJS core takes 20kbytes without any minification. The use of KevoreeJS does not have any real impact on the SPA performance using standard browser and hardware.

IV. Related Work

Several component model exist for building dynamically adaptable web application. OSGi [8] provides an initial RFP for providing an OSGi for web applications. Eclipse provides an initial solution within the Orion project to write plugins for its online IDE [18]. However, this approach does not support dynamic reconfiguration without reloading the web pages. ComponentJS [19] is a stand-alone MPL-licensed Open Source library for JavaScript, providing a powerful run-time Component System for hierarchically structuring the User-Interface (UI) dialogs of complex SPA. It provides a rich component model for UI composition based on the concepts of Event, Service, Hook, Model, Socket, and Property. However, it does not manage the dynamic reconfiguration of running applications. Lerner et al. [18] present C3, an implementation of the HTML/CSS/JS platform designed for web-client research and experimentation. C3 explores the role of extensibility throughout the web platform for customization and research efforts. C3 proposes an interesting component model for the Web browser (the application container) itself. It proposes an extensible architecture to let developers to evolve the browser. Escoffier et al. [19] developed a service-oriented component framework, named H-ubu. Its purpose is to bring modularity to applications and to ease their runtime adaptation. H-ubu is based on the notion of components with provided and required services and on a hub, a specific component in charge with runtime components bindings. H-ubu follows a dynamic service approach. Main adaptations consists in reacting when a component becomes unavailable but the framework does not provide specific mechanism to automatically deploy and remove required components. The configuration model is not explicit, as in a service oriented architecture, each hub manages dynamically the bindings between component services.

V. Conclusion

This paper highlights the motivations, challenges, and main requirements to build a dynamic component model for single page applications. It shows how a distributed system running on top of various browsers relying on heterogeneous hardware can be considered as a common service. This leads to the need of managing the configuration of such a service from a common and abstract view. This paper presents the requirements for such a system and KevoreeJS, an implementation of the Kevoree Component Model for the Browser. It evaluates KevoreeJS by building a dashboard for sensor-based applications that can be dynamically reconfigured. In particular, it supports dynamic client/server code partitioning and dynamic component installation without refreshing the web page.

We are currently extending this approach to improve security, to support analytic services and search engine optimizations. From a technical point of view, we are working on a pattern to support other SPA frameworks such as AngularJS 2 or React. In this direction, we are working on a development model that decreases the coupling between the component implementation and the SPA application framework used to provide a clean MVC framework.

[18]http://wiki.eclipse.org/Orion

[19]http://componentjs.com/

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n611337, the HEADS project (www.heads-project.eu)

References

[1] B. Green and S. Seshadri, *AngularJS*. " O'Reilly Media, Inc.", 2013.

[2] J. Cravens and T. Q. Brady, *Building Web Apps with Ember. js*. " O'Reilly Media, Inc.", 2014.

[3] A. Osmani, *Developing Backbone. js Applications*. " O'Reilly Media, Inc.", 2013.

[4] F. Monteiro, *Learning Single-page Web Application Development*. Packt Publishing Ltd, 2014.

[5] A. Fedosejev, *React. js Essentials*. Packt Publishing Ltd, 2015.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0201633612

[7] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[8] R. Hall, K. Pauls, S. McCulloch, and D. Savage, *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[9] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[10] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris, "Safe & efficient gradual typing for typescript," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015, pp. 167–180.

[11] K. Dhiman and B. Quach, "Google's go and dart: parallelism and structured web development for better analytics and applications," in *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2012, pp. 253–254.

[12] J. Ahn, "Demystifying seo with experiments." [Online]. Available: https://engineering.pinterest.com/blog/demystifying-seo-experiments

[13] B. Morin, O. Barais, G. Nain, and J. Jézéquel, "Taming dynamically adaptive systems using models and aspects," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 122–132. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070514

[14] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software-Practice and Experience*, vol. 36, no. 11, pp. 1257–1284, 2006.

[15] P. Grace, G. S. Blair, and S. Samuel, "A reflective framework for discovery and interaction in heterogeneous mobile environments," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 1, pp. 2–14, 2005.

[16] P.-C. David, T. Ledoux *et al.*, "Safe dynamic reconfigurations of fractal architectures with fscript," in *Proceeding of Fractal CBSE Workshop, ECOOP*, vol. 6, 2006.

[17] F. Fleurey, B. Morin, A. Solberg, and O. Barais, "MDE to manage communications with and between resource-constrained systems," in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. Kühne, Eds., vol. 6981. Springer, 2011, pp. 349–363. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24485-8

[18] B. S. Lerner, B. Burg, W. Schulte, and H. Venter, "C3: An experimental, extensible, reconfigurable platform for html-based applications," in *2nd USENIX Conference on Web Application Development*. USENIX, June 2011. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=150010

[19] C. Escoffier, P. Lalanda, and N. Rempulski, "h-ubu: An Industrial-Strength Service-Oriented Component Framework for JavaScript Applications," in *FSE 2013 - ACM SIGSOFT Symposium on the Foundations of Software Engineering*, M. M. Bertrand Meyer, Luciano Baresi, Ed. Saint Petersburg, Russia: ACM, Aug. 2013, pp. 699–702, industrial track: Effective Industry Use of Software-Engineering Tools. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00854339