Supporting Customized Views for Enforcing Access Control Constraints in Real-time Collaborative Web Applications

Patrick Gaubatz[1], Waldemar Hummer[2], Uwe Zdun[1] and Mark Strembeck[3]

[1]Faculty of Computer Science, University of Vienna, Austria
{firstname.lastname}@univie.ac.at
[2]Distributed Systems Group, Vienna University of Technology, Austria
{lastname}@infosys.tuwien.ac.at
[3]Institute for Information Systems, WU Vienna, Austria
{firstname.lastname}@wu.ac.at

Abstract Real-time collaborative Web applications allow multiple users to concurrently work on a shared document. In addition to popular use cases, such as collaborative text editing, they can also be used for form-based business applications that often require forms to be filled out by different stakeholders. In this context, different users typically need to fill in different parts of a form. Role-based access control and entailment constraints provide means for defining such restrictions. Major challenges in the context of integrating collaborative Web applications with access control restrictions are how to support changes of the configuration of access constrained UI elements at runtime, realizing acceptable performance and update behaviour, and an easy integration with existing Web applications. In this paper, we address these challenges through a novel approach supporting constrained and customized UI views that support runtime changes and integrate well with existing Web applications. Using a prototypical implementation, we show that the approach provides acceptable update behaviour and requires only a small performance overhead for the access control tasks with linear scalability.

1 Introduction

Real-time collaborative Web applications such as Google Docs[1], Etherpad[2], or Creately[3] aim to efficiently support the joint work of different team members, allowing them to collaboratively work on the same artifact at the same time. In addition to such popular examples, the real-time collaboration approach can also be used in typical business applications that often require multiple forms to be filled out by different stakeholders [7]. A crucial – though in the context of real-time collaborative Web applications often neglected – aspect of these business applications is access control.

[1] https://docs.google.com
[2] http://etherpad.org
[3] http://creately.com

In recent years, role-based access control (RBAC) [14] emerged as a standard for access control in software systems. In RBAC, roles are used to model different job positions and scopes of duty within an information system. These roles are equipped with permissions to perform tasks. Human users (subjects) are assigned to roles according to their work profile [17]. For example, in an e-health application only a doctor shall be allowed to file a report. Moreover, a second doctor needs to check and sign the same report (four-eyes principle). In this example the role *doctor* is equipped with both permissions, i.e., filing and signing a report. To prevent a single subject from performing both tasks on the same report (thus undermining the four-eyes principle) we have to constrain these two tasks with an entailment constraint. *Entailment constraints* (see, e.g., [3, 18, 20]) provide means for placing restrictions on the subjects who can perform a task x given that a certain subject has performed another task y. *Mutual exclusion* and binding constraints are typical examples for entailment constraints. For instance, a *dynamic mutual exclusion* (DME) constraint defines that two subjects must not perform two mutually exclusive tasks in the same instance of a Web document. This means, that the permissions to perform two DME tasks can be assigned to the same subject or role, but for each instance of a particular Web document, we need two distinct individuals to perform both tasks. Binding constraints, on the other hand, can be seen as the opposite of mutual exclusion constraints. For example, *subject binding* defines that the subject who performed the first task must also perform the bound tasks.

Ideally, realizing form-based business applications with a real-time collaborative Web application approach would enable us to enforce RBAC and entailment constraints directly as the users collaboratively work on the forms, i.e., by constraining (e.g. by disabling, locking, or hiding) certain control elements in the user interfaces (UI) for certain subjects. However, so far this topic has – to the best of our knowledge – not been addressed in the existing literature. Major open challenges in this context are how to support changes of the configuration of access constrained UI elements at runtime, realizing acceptable performance and update behaviour, and the easy integration with existing Web applications.

In this paper, we address these challenges that are inherent to enforcing access control constraints in the context of real-time collaborative Web applications. The client-side part of our approach follows the Model-View-ViewModel pattern [15]. Additional server-side components complement our service-based architecture. The resulting architecture enables us to support runtime changes and facilitates the integration our approach with existing applications (see Section 6.2). Furthermore, we show that the approach provides acceptable update behaviour and requires only a small performance overhead for the access control tasks. In our experiments, it shows linear scalability (see Section 6.1). The remainder of this paper is structured as follows: An example scenario motivates our approach in Section 2. In Sections 3 and 4 we propose a novel approach supporting constrained and customized UI views. In Section 5, we describe a prototypical implementation and revisit the motivating example. After comparing to related work in Section 7 we conclude in Section 8.

2 Motivating Example and Challenges

As a motivating example, consider a Web-based application where patient health records are maintained using forms for data entry. The data entry procedure is typically included in a business process with well-defined roles and responsibilities (see, e.g., [9]). In previous work, we presented *CoCoForm* [7], a real-time collaborative Web application framework in which several users can concurrently fill out HTML forms.

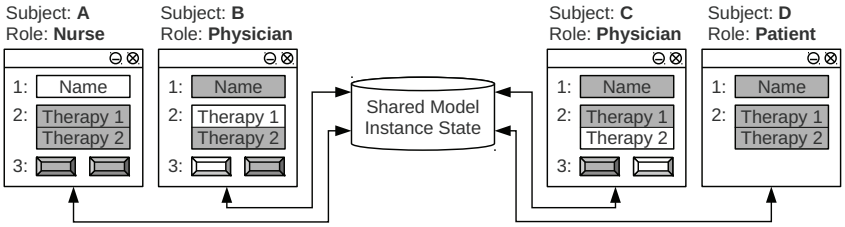

Figure 1: Form-based Collaborative Web Application with Customized Views

Figure 1 shows a simplified example of using *CoCoForm* in the e-health domain. It includes four subjects with shared access to the health record of a patient. The subjects take different roles (nurse, physician, patient) which define their permissions within the application. The nurse enters the name and other personal data of the patient into a textfield (identified by "1"), physician B adds "Therapy 1" to the list of therapies (field "2"), and physician C suggests an additional specialized therapy "Therapy 2". The entire form record is then confirmed by both physicians (buttons "3"). To enforce the four-eyes principle (DME constraint), after physician B clicks the first submit button, the second button is deactivated for physician B, but remains active for physician C. Moreover, each physician can only modify his own therapy suggestions (subject-binding constraint). Finally, the patient should have read-only access to the data. To enforce these constraints, each user has a customized view with partial access to the collaboratively shared model. In Figure 1, white elements can be accessed and modified by the respective user, whereas elements with gray background are subject to access limitations (e.g., read-only but not editable).

A major challenge to realize such customized views for access control constraints is that the *configuration of constrained UI elements* must be *computed server-side* and *effected client-side*. Moreover, this *configuration* might *change dynamically at runtime*. Other challenges are related to *performance and update behaviour*: This means, we immediately need to deliver customized views to all UIs that access the same instance of a Web document (e.g., in the example the UIs need to be updated immediately after one of the subjects changes a document). Such an *immediate update* is required to prevent users from performing actions that were either already performed by another user or that are con-

strained by an entailment constraint (which may have a direct impact on the subjects who are allowed to fill in certain form field for example, see Section 1). In order to be applicable in real-world application scenarios, the approach should *efficiently handle large numbers of simultaneously connected users*. Finally, the approach should allow for an *easy integration with existing Web applications*.

3 Approach Synopsis

The aim of our approach is to support access control and customized views in real-time collaborative Web applications. The *View* of a Web application represents the UI with all visible and invisible elements, form input fields, interactive content, and more. The elements and associated interactions in the UI are subject to constraints (e.g., actions that require a certain permission) which are encoded in well-defined (RBAC) models. Our approach maps the model elements to configuration properties, and clients request the runtime values of these configurations from a *View Service*. The user-specific configurations computed by the server-side *View Service* are then applied to the *View* on the client-side.

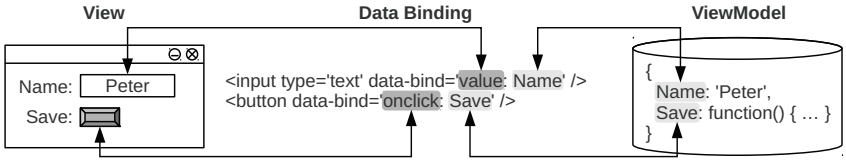

Figure 2: Data Binding between View and ViewModel

As the basic binding concept between the *View* and the *Model*, our approach applies the *Model-View-ViewModel* (MVVM) pattern [15]. The MVVM is a specific version of the Presentation Model pattern (see [6]). It relies on the data binding concept, which ensures that the *View* and the state of its components are bound to properties of a *ViewModel*. This means that changes of the *ViewModel* are automatically reflected in the *View*. For instance, in Figure 2 we can see that the `value` attribute of the `<input>` field is bound to the property `Name` in the *ViewModel*. Secondly, the `onclick` handler of the `button` is bound to the *ViewModel*'s `Save` property. In general, the *ViewModel* acts as a mediator between the Model and the View by encapsulating all logic (e.g., formatting and data type conversion) needed to expose the properties and functionalities of the *Model* to the bound *View*. Additionally, it is in charge of reacting to user commands (e.g., a user fills out an input field) and reflecting them by performing the corresponding *Model* state changes. In general, the MVVM pattern makes it easy to realize the client-side part of the required *View Customization* functionality. In particular, we can customize a client's *View* just by configuring its *ViewModel* properties.

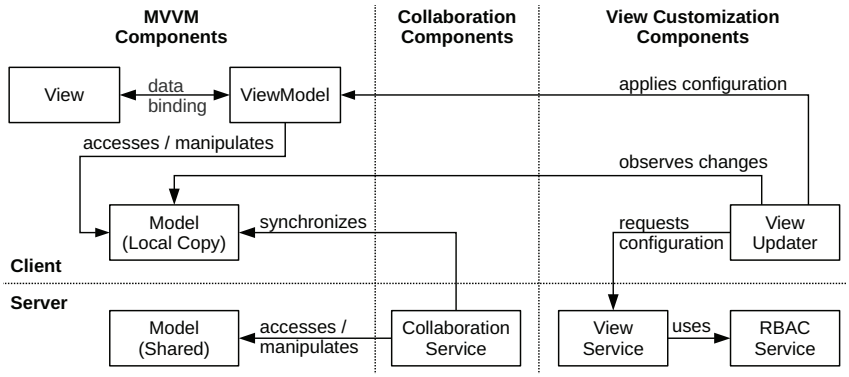

Figure 3: Architectural Overview

Figure 3 provides an architectural overview of the components (i.e., both server-side and client-side) and interactions in our approach that are needed to realize the required *View Customization* functionality. The left-hand column of the figure depicts the core components of the MVVM architecture. In contrast to the classic MVVM architecture, in our approach the *ViewModel* does not directly access/manipulate the shared *Model* (i.e., the shared application state). Instead, it accesses/manipulates only a local copy of the shared *Model*. That is, a *Collaboration Service*, which is the cornerstone of a real-time collaborative Web application, ensures that the server-side shared *Model* is constantly kept in sync with all client-side copies of it. While the *Collaboration Service* allows us to let users collaboratively work on the same Web document, it certainly does not provide means for constraining (e.g., disabling, locking, or hiding) certain control elements in the UI for certain users. Consequently, the *View Service* uses the central *RBAC Service* to compute *ViewModel* configurations. Although these *ViewModel* configurations are computed server-side, they need to be effected client-side, i.e., to constrain UI elements in the *Views* of each client. To account for this, the client-side *View Updater* component of each client actively requests (i.e., pulls) the computed *ViewModel* configurations from the *View Service*. Eventually, these configurations are then applied to the *ViewModel*, which in turn – through data binding – effectively constrain the *Views* of each client.

4 Supporting Customized Views

This section details how the different components of the architecture outlined in Figure 3 enable us to enforce access control policies and entailment constraints directly as the users collaboratively work on a shared *Model*, i.e., by constraining certain control elements in the UI for certain subjects.

Firstly, we want to exemplify our UI customization approach using Figure 4. The figure is divided in two parts, the client-side part and the server-side part.

The figure shows that the *Model* contains only a single property `Name` which is mapped ① to both, a `value` and a `label` property in the *ViewModel*. Next, by applying the basic MVVM pattern, the two properties are bound ② to concrete `<label>` and `<input>` HTML elements in the *View*.

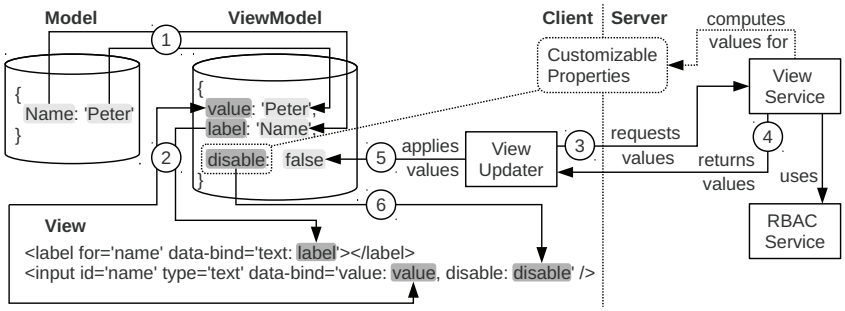

Figure 4: View Customization Example

Next, we assume that the `<input>` field (and the associated action in the RBAC model) is constrained by some RBAC policy. To customize the `<input>` and dynamically make it enabled or disabled, we add the `disable` property to our *ViewModel*. The name of the property (`disable`) is added to the set of *Customizable Properties*. That is, we do not want the client to decide about the value of the `disable` property on its own. Instead, the server has to compute the values for each *Customizable Property*. Thus, the client requests ③ the values from the server-side View Service. The View Service uses the RBAC Service to determine the concrete value for the `disable` property (`true` if and only if the client is allowed to change the `Name` property of the *Model*). The View Service returns ④ the list of *Customizable Properties* together with their customized values to the client-side View Updater. Next, the View Updater applies ⑤ these customized values to the *ViewModel*. Finally, the property value is automatically reflected ⑥ in the *View*, as we have bound the `disable` property of the *ViewModel* to the `disabled` flag of our `<input>` field.

Abstracting from the example in Figure 4, the basic idea of our approach is that the core *ViewModel* is augmented with additional *Customizable Properties*. These properties are used to easily implement customizations in the *View* (e.g., enabling/disabling an `<input>` field). While the property names are defined and processed on the client-side, the actual values for these properties are computed for each user separately on the server side. In summary, the purpose of the *Customizable Properties* is twofold:

1. **Enablement.** At the client-side, these properties have an enabling character, i.e. they allow for realizing the customization of the *View*.
2. **Contract.** Additionally, they can be considered as a contract between the *ViewModel* and the server-side View Service. That is, the client-side *View-*

Model defines the set of *Customizable Properties* and the server-side View Service provides the actual values for these properties. For instance, if the server returns a value of `true` for the `disable` property (see the example above), the client is responsible for actually disabling the `<input>` field in the client's *View*. Hence, the client and the server must have a common view of the semantics of each property.

4.1 Client-side Updates of the ViewModel

The View Updater is in charge of requesting and applying *ViewModel* configurations from the View Service. We propose a simple request/response style of communication between these two components.

```
1  var subject, role,
2      viewModel = {
3          value: 'Peter', label: 'Name', // core properties
4          disable: false, visible: true   // customizable properties
5      };
6
7  function requestView() {
8      var xhr = new XMLHttpRequest(),
9          uri = '/viewService?subject=' + encodeURI(subject) + '&role=' + encodeURI(role);
10     xhr.open('GET', uri);
11     xhr.onload = function() {
12         var configuration = JSON.parse(this.response); // e.g. {disable: true, visible: true}
13         for (var property in configuration) {
14             viewModel[property] = configuration[property];
15         }
16     };
17     xhr.send();
18 };
19
20 function onModelChange(property, value) { // called whenever the Model changes
21     requestView();
22     viewModel[property] = value;
23 };
```

Listing 1: A Simple View Updater Example

Listing 1 illustrates an excerpt of the corresponding exemplary client-side JavaScript code. After firing the request (line 17) we asynchronously process the response that contains the requested *ViewModel* configuration. In the example from Listing 1, the *Customizable Properties* consist of two properties `disable` and `visible` (line 4). Correspondingly, the *ViewModel* configuration returned by the View Service contains concrete values for these two properties, e.g., `{disable: true, visible: true}`. The next step is to apply this configuration to our *ViewModel*. To this end, the JSON-encoded result of the View Service is parsed, and each entry in the result is applied to the local `viewModel` variable (lines 12-15).

Having discussed how the View Updater requests and applies *ViewModel* configurations, we now draw our attention to the question when it should issue its requests. In general, we can say that this depends on the application's context.

However, in our context, i.e., RBAC and entailment constraints, we can also say that *Views* need to be updated exclusively after a *Model* change has happened. Whenever a property is changed in the shared *Model* (i.e., the application state), all *Views* need to be re-computed and (potentially) updated. This circumstance is also reflected in Listing 1 (lines 20-23), where we can see that a new request is triggered for every *Model* change that happens (via the `onModelSync()` callback).

4.2 Server-side Computation of ViewModel Configurations

The computation of *ViewModel* configurations is done server-side, i.e., by the View Service. Upon a request, the View Service returns a *ViewModel* configuration to the requesting client-side View Updater component.

```
1  function onRequest(subject, role) {
2      var property = 'Name', // there is just a single 'Name' property in our model
3          response = {
4              disable: !rbacService.canWrite(subject, role, property),
5              visible: rbacService.canRead(subject, role, property)
6          };
7      return response; // e.g. {disable: false, visible: true}
8  }
```

Listing 2: Basic View Service Example

For instance, in Listing 2 we can see an excerpt of the implementation of a very basic View Service[4] that is tailored to return a configuration for the set of *Customizable Properties* defined in the application code presented in Listing 1. In essence, the service has to compute values for the two *Customizable Properties*, i.e., `disable` and `visible`. As we can see (line 4), it "asks" the central RBAC Service if the provided `subject`/`role` combination has the permission to change (i.e., write) the application's *Model* property, i.e., `Name`. A positive answer (i.e., the user has the permission to change the *Model* property) is reflected with a `disable` value of `false`, which in turn enables the UI element and eventually allows this specific user to manipulate the *Model* property in her customized *View*. Similarly, the service uses the RBAC Service to determine a value for the `visible` property. Eventually, it returns the JSON-encoded configuration (line 7) to the requesting client. Note, that the required parameters of the service, i.e., `subject` and `role` could be supplied as URI parameters (as in line 9 in Listing 1).

5 Implementation – The CoCoForm Framework

This section discusses a prototype implementation of our approach, called Constrainable Collaborative Forms (CoCoForm)[5]. We used CoCoForm to implement and evaluate the e-health record case from Section 2.

[4] Note that we chose JavaScript solely for its well-known and concise syntax.
[5] A proof-of-concept demo is available at `http://demo.swa.univie.ac.at/cocoform2`

Our prototype is based on the OpenCoweb[6] framework, which consists of both, a Collaboration Service (as in Figure 3) and a (client-side) JavaScript API. The latter allows to subscribe to incoming *Model* change events, i.e., by registering a callback function which in turn enables us to trigger our View Updater component (as in Listing 1).

The View Updater issues simple XMLHttpRequests to obtain *ViewModel* configurations from the View Service. The View Service is implemented as a plain HTTP Service in Java, using the JAX-RS API[7], and the configurations are returned in JSON format. The central RBAC Service, which is utilized by the View Service, has been presented in previous work [7]. We use a model-driven approach for defining forms and securing them using access control constraints. Server-side we internally work with Ecore[8] model instances which are marshalled into JSON for the client-side JavaScript application.

Besides OpenCoweb's JavaScript API, we use the Knockout[9] library for realizing the MVVM pattern in the client-side application code. In particular, we also use Knockout's Mapping plugin which allows us to automatically transform the JSON-encoded *Model* into a *ViewModel*. The Mapping plugin also allows us to easily update the *ViewModel* whenever the *Model* changes. Additionally, we augment the *ViewModel* with additional `visible` and `editable` properties. We also use Knockout's template mechanism to (1) create the needed input fields and buttons on-the-fly and (2) establish data binding using corresponding `data-bind` attributes.

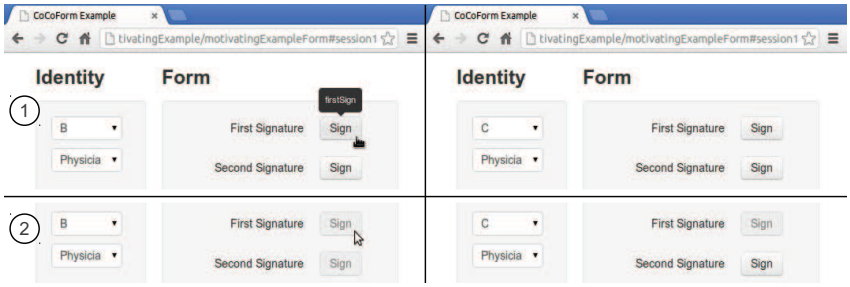

Figure 5: Customized Views and Dynamic Mutual Exclusion with *CoCoForm*

Motivating Example Revisited Now we want to revisit the dynamic mutual exclusion example from Section 2 and discuss a concrete implementation using *CoCoForm*. Figure 5 shows four screenshot excerpts of an example form with

[6] OpenCoweb, `http://opencoweb.org`
[7] JAX-RS, `http://jax-rs-spec.java.net`
[8] Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf`
[9] Knockout, `http://knockoutjs.com`

two dynamically mutual exclusive buttons. In particular, these buttons represent the first and the second signature on a patient record (as described in Section 2). Figure 5 is vertically split into two columns, i.e., the *View* of the first user (subject B) and the second user (subject C). Both subjects are concurrently working on this form. In the first row (indicated with ①) we can see that both buttons are available for both subjects. The mouse pointer in the upper left part indicates that subject B clicks the first signature button. This click results in a *Model* change which triggers the View Updater component of both clients. As a result, the View Updaters of both clients issue a request to the View Service, resulting in the updated Views in ②. While the first button has been disabled for both clients (which reflects the requirement that any form element can only be manipulated once), the second button is only disabled for subject B. This is due to the dynamic mutual exclusion constraint which demands that subject B, who has just clicked the first button, must be prevented from clicking the second button (see Section 2). However, subject C is still allowed to click the second button. In summary, this example illustrates how our approach enforces access control constraints in real-time collaborative Web applications by dynamically changing the UIs of each user at runtime.

6 Evaluation

In the following sections we discuss both, our lessons learned and the limitations of our approach and the findings of the conducted performance evaluation.

6.1 View Service Performance Evaluation

In the context of real-time collaborative Web applications, users typically expect instantaneous update behavior, which led us to study in how far our UI customization approach meets this requirement. We identify the View Service as a potential performance bottleneck. In particular, we anticipate that requests issued by a potentially large number of users (i.e., resulting from a *Model* change) need to be handled concurrently by *CoCoForm's* View Service.

All measurements have been conducted on a machine equipped with a 2.4 GHz dual core CPU, 8 GB RAM, running Ubuntu GNU/Linux 12.10. Both, the View Service and the testing tool, i.e., Apache's `ab` tool[10], ran on the same machine. Hence, the measurements are free from any network-induced effects such as latency, jitter and so on.

Figure 6 depicts the average response times of both, the actual View Service (solid line) and a "Null" (i.e., no computation at all) Service (dashed line), for a given number of concurrent requests. For instance, in the case of 600 concurrent requests, the average response time for all clients is roughly 200 ms while the response time of the Null Service is roughly 50 ms. This means, that in this case it takes roughly 150 ms to compute a single *ViewModel* configuration, while the rest of 50 ms accounts for the underlying communication and Web Service stack.

[10] Apache `ab` tool, `http://httpd.apache.org/docs/2.4/programs/ab.html`

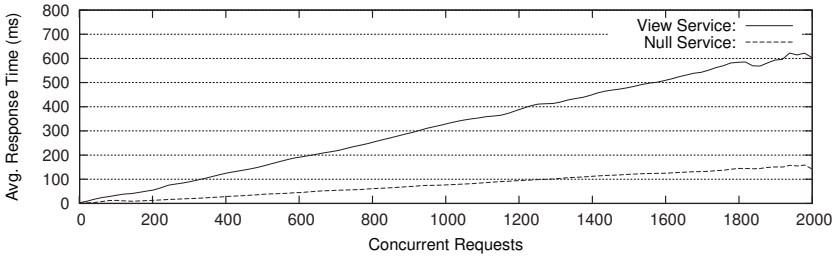

Figure 6: View Service Response Times

The evaluation results indicate that our View Service implementation has linear scalability. Even in the case of 2000 users working on the same form document collaboratively, the average response time remains well below a second. In our experiment, the View Service's response times amount to approximately four times the response times of the Null Service. As the Null Service represents the theoretical minimum that is possible for the given Web Service framework, we consider the performance overhead acceptable.

6.2 Lessons Learned

We implemented the *CoCoForm* prototype (see Section 5) to demonstrate the feasibility of our approach (see Section 4). We showed that access control policies and entailment constraints in the context of real-time collaborative Web applications can effectively be enforced by dynamically constraining UI elements for certain subjects. In the following paragraphs we want to discuss our lessons learned and the limitations of our approach.

Our approach is complementary to currently available frameworks and solutions that support the development of real-time collaborative Web applications such as Apache Wave[11], ShareJS[12] and OpenCoweb (see Section 5). This is due to the fact that it is completely decoupled from the collaborative aspects of the application. In essence, supporting customized views using our approach merely requires the deployment of a single, dedicated and self-contained View Service as well as hooking-in the View Updater code into the client-side application code.

Although our approach is built upon the MVVM pattern, it does not exclude other approaches (e.g., the classic Model-View-Controller pattern). Instead, we argue that our approach can coexist with others. In that case, the *ViewModel* is solely used to realize the customizable parts of the *View*. Hence, it just contains the set of *Customizable Properties*. The only requirement is that the corresponding DOM nodes (e.g., `<input>` elements) are augmented with additional data binding attributes (e.g., `data-bind`). Note that this even works in the case of

[11] Apache Wave, `http://incubator.apache.org/wave`
[12] ShareJS, `http://sharejs.org`

dynamically generated (i.e., generated using JavaScript code) DOM nodes, as long as it is possible to add the data binding attributes.

A major concern – especially in the context of real-time collaborative Web applications – is the ability to apply the View customization nearly instantaneously. In other words, the response times of the View Service must be kept low. Keeping the response time low with a growing number of simultaneously connected users, requires that the system is able to scale. Our View Service itself is completely stateless, as (1) each request contains all necessary information (e.g., subject and role) that is needed to compute a *ViewModel* configuration and (2) no information at all needs to be persisted. This stateless nature as well as the simple request/response style of communication between the View Updater and the View Service allows for scaling horizontally in a straightforward manner, i.e., the communication can be routed through a load-balancing proxy that distributes each request among multiple instances of the service.

However, the request/response communication style also comes with a couple of challenges. For example, there is the issue of "the needless request". This is the case when the View Service returns a *ViewModel* configuration that is not different from the currently active one. Hence, we could have saved both client-side and server-side computing resources (e.g., CPU time, network bandwidth, etc.) if we simply had not issued this "needless request" in the first place. This issue can be addressed using a push approach (instead of the presented pull approach). That is, the View Service would selectively push new *ViewModel* configurations to the clients only if it is necessary (i.e., at least one *ViewModel* property needs to be changed). However, this push approach introduces a certain amount of complexity to the View Service. For instance, it would require an explicit session handling, i.e., in a push scheme we have to maintain a list of connected clients to correctly update the corresponding *ViewModels*. Moreover, a push scheme would also require to keep track of each client's *ViewModel* to determine if we need to push a new *ViewModel* to a particular client. In summary, the push approach allows for avoiding "needless requests" (in fact, no requests are made at all) while the pull approach comes with a lower complexity, especially when scaling (i.e., when multiple instance of the View Service have to coordinate session with each client's *ViewModel* configuration). Another idea to – at least – mitigate this problem would be a more efficient client-side triggering logic. For instance, we could provide the clients with a list of *Model* properties that are not constrained by any access control constraint at all. Then, the clients would not need to request a new *ViewModel* configuration whenever a *Model* change event arrives that is contained in the list of unconstrained properties.

In our approach access control policies and entailment constraints are enforced client-side, i.e., by constraining UI elements. From a security perspective, however, we often cannot trust code that is executed on the client (i.e., the browser). The reason is that we can not prevent a potential attacker from modifying the code to be executed. For instance, an attacker might be able to change the *ViewModel* configuration to gain access to a constrained UI element and eventually pass a *Model* change event (i.e., concerning a constrained *Model*

property) to the Collaboration Service. However, we could contain the effects of such client-side code injections by preventing such unauthorized *Model* changes (1) from being applied to the server-side *Model* and (2) from being distributed to other session participants. This can be achieved by routing all incoming (i.e., coming from the clients) *Model* change events trough an enforcement proxy. This proxy uses the RBAC Service to decide if it should forward the event to the Collaboration Service (i.e., in case the client has the permission to change the *Model* property) or not. This guarantees that client-side code injections do not lead to server-side *Model* changes or impact session participants.

Finally, our approach assumes that the *Model* is being synchronized with all clients. That is, all clients "see" exactly the same *Model*. However, if this *Model* contains sensitive information, this might be an issue. We will address this problem as part of our ongoing research.

7 Related Work

In this section we discuss related work in the area of customized and shared application views, collaboration platforms as well as access control enforcement.

Customized and Shared Application Views. Similar to customized views in our approach, Koidl et al. [12] propose user-specific Web site rendering. However, their approach aims at user-centric personalization of Web experience, whereas the customized views in our approach result from RBAC policies and entailment constraints. An interesting aspect in their solution is that the personalization is cross-site, i.e., it spans the Web sites of multiple providers. Our approach currently does not implement cross-provider policies. However, we presented a related approach for cross-organizational access control in Web service based business processes in [9]. As part of our future work, we will integrate cross-site capabilities in our approach for real-time collaborative Web applications. Berry et al. [2] have applied role-based view control to desktop applications. Their approach captures the virtual framebuffer of application windows and applies blurring, highlighting, pixelizations, and other manipulations over the rendered view. Our approach benefits from the fact that manipulation of Web user interfaces is easier to achieve; using the path to the target DOM element, our client-side View Updater takes care of customized view manipulations.

Collaboration Platforms. The seminal work of Sun et al. [19] proposes the transparent adaptation (TA) approach to convert single-user applications into collaborative multi-user applications. The cornerstone of TA is operational transformation (OT) [4]. Our approach is orthogonal to OT: the RBAC policies and entailment constraints provide an application workflow with well-defined responsibilities, and we maintain document consistency by allowing only sequences of operations that comply with this workflow. Farwick et al. [5] discuss an architecture for Web-based collaborative metamodeling. Their framework allows multiple users to work on graphical meta-models collaboratively. Modifications of the (meta-)models are secured by basic access control measures, but in contrast to our work, they do not explicitly address customized views and dynamic

updates resulting from the enforcement of RBAC entailment constraints. Heinrich et al. [8] present a generic collaboration infrastructure aimed at transforming existing single-user Web applications into collaborative multi-user Web applications by synchronizing DOM trees. In other words, their approach makes sure that the DOM trees of all clients in a collaborative session is constantly kept in sync. As we strive for customizing the DOM tree for each client, this approach is completely at odds with ours. Consequently, we require synchronization to take place at the model-level instead of the view-level (as in [8]).

Security and Access Control Enforcement. A plethora of approaches have been presented for integrating security and access control in Web applications. Joshi et al. [10] provide an early study on generic security models for Web-based applications. Starnberger et al. [16] use smart card based security and discuss a generic proxy architecture to enforce authorizations. In [1], Belchior and colleagues model RBAC policies using RDF triples and N3Logic rules. Mallouli et al. [13] use extended finite state machines (EFSM) to model systems with OrBAC [11] (Organization Based Access Control) security policies. However, none of these approaches addresses the enforcement of access control policies and entailment constraints in dynamic real-time Web applications.

8 Conclusion and Future Work

In this paper, we demonstrate that access control policies and constraints – in particular entailment constraints – in the context of real-time collaborative Web applications can effectively be enforced by dynamically constraining UI elements for certain subjects. We show that our service-based approach can be used to realize the corresponding UI view configuration functionality and we provide evidence that it is potentially capable of meeting the – especially in the context of real-time collaborative Web applications important – requirement of nearly instantaneous update behavior, even for a large number of simultaneously connected users. Although the client-side part of the UI view configuration functionality is built upon the MVVM pattern, we show that it can easily coexist with others.

As future work we will look into privacy issues (see Section 6.2) and apply our approach to other types of collaborative processes. In particular, we are interested in establishing the concept of entailment constraints in more dynamic processes (e.g., text editing or modeling) where we will have to deal with completely dynamic (i.e., changing at runtime) access control and constraint models.

References

1. Belchior, M., Schwabe, D., Silva Parreiras, F.: Role-based access control for model-driven web applications. In: 12th International Conference on Web Engineering (ICWE). pp. 106–120 (2012)
2. Berry, L., Bartram, L., Booth, K.S.: Role-based control of shared application views. In: 18th ACM symposium on User interface software and technology (UIST). pp. 23–32 (2005)

3. Bertino, E., Ferraria, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Transactions on Information and System Security 2(1), 65–104 (1999)
4. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Record 18(2), 399–407 (1989)
5. Farwick, M., Agreiter, B., White, J., Forster, S., Lanzanasto, N., Breu, R.: A web-based collaborative metamodeling environment with secure remote model access. In: 10th International Conference on Web Engineering (ICWE). pp. 278–291 (2010)
6. Fowler, M.: Presentation model. Essay, July (2004)
7. Gaubatz, P., Zdun, U.: Supporting entailment constraints in the context of collaborative web applications. In: 28th Symposium On Applied Computing (2013)
8. Heinrich, M., Lehmann, F., Springer, T., Gaedke, M.: Exploiting single-user web applications for shared editing: a generic transformation approach. In: Proceedings of the 21st international conference on World Wide Web. pp. 1057–1066 (2012)
9. Hummer, W., Gaubatz, P., Strembeck, M., Zdun, U., Dustdar, S.: An integrated approach for identity and access management in a SOA context. In: 16th ACM Symposium on Access Control Models and Technologies (SACMAT) (2011)
10. Joshi, J.B.D., Aref, W.G., Ghafoor, A., Spafford, E.H.: Security models for web-based applications. Communications of the ACM 44(2), 38–44 (2001)
11. Kalam, A.A.E., Benferhat, S., Miège, A., Baida, R.E., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y., Trouessin, G.: Organization based access control. In: 4th IEEE Int. Workshop on Policies for Distributed Systems and Networks (2003)
12. Koidl, K., Conlan, O., Wade, V.: Towards user-centric cross-site personalisation. In: 11th International Conference on Web Engineering (ICWE). pp. 391–394 (2011)
13. Mallouli, W., Orset, J.M., Cavalli, A., Cuppens, N., Cuppens, F.: A formal approach for testing security rules. In: 12th ACM symposium on Access control models and technologies (SACMAT). pp. 127–132. ACM (2007)
14. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role- based access control models. Computer 29(2), 38 –47 (1996)
15. Smith, J.: WPF apps with the Model-View-ViewModel design pattern. MSDN magazine (2009)
16. Starnberger, G., Froihofer, L., Goeschka, K.M.: A generic proxy for secure smart card-enabled web applications. In: 10th International Conference on Web Engineering (ICWE). pp. 370–384 (2010)
17. Strembeck, M.: Scenario-driven Role Engineering. IEEE Security & Privacy 8(1) (January/February 2010)
18. Strembeck, M., Mendling, J.: Generic algorithms for consistency checking of mutual-exclusion and binding constraints in a business process context. In: On the Move to Meaningful Internet Systems (OTM). pp. 204–221 (2010)
19. Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., Cai, W.: Transparent adaptation of single-user applications for multi-user real-time collaboration. ACM Transactions on Computer-Human Interaction 13(4), 531–582 (2006)
20. Wainer, J., Barthelmes, P., Kumar, A.: W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. International Journal of Cooperative Information Systems (IJCIS) 12(4) (Dec 2003)