UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology

Liisi Haav

Role-Based Enterprise Mashups with State Sharing, Preservation and Restoration Support for Multi-Instance Executions

Master's thesis (30 ECTS)

Advisor: Peep Küngas

Author: …..... “.....” May 2013

Advisor: …..... “.....” May 2013

Approved for defence

Professor: …..... “.....” May 2013

Tartu 2013

Table of Contents

Abstract

Recent hype on consumer web mashups has resulted in many general-purpose mashup frameworks and tools. These tools aim at simplifying the creation of mashups targeted to mainstream Internet users. At the same time, mashups are also used for solving specific business-related tasks. Such mashups are called enterprise mashups and more sophisticated frameworks and tools have been developed to support their creation. However, similarly to traditional web application development tools, the complexity of these frameworks is hindering the main benefits associated with mashup development – agility and simplicity.

This thesis aims at extending a general-purpose mashup framework to support development of enterprise mashups while still preserving the simplicity and agility of development. More specifically, this thesis describes a solution for role-based decomposition of mashups for multi-instance executions with state sharing, preservation and restoration.

In this thesis, a general-purpose mashup framework is extended with the concept of roles to support multi-user interaction and decomposing complex enterprise mashups with rich interactions into role-based views. In the context of this thesis, a view is defined as a subset of widgets a mashup is made of. Hence, through views an effective mechanism is provided for decomposing enterprise mashups to mashups as simple as general-purpose mashups.

Additionally, this thesis proposes a generic solution for multi-instance mashup executions. In this thesis, each workflow instance is associated with an instance of a mashup. Since situational applications target at solving users day-to-day tasks, it is necessary to support multiple instances of a mashup. Furthermore, support for multiple mashup instances leverages users' ability to participate in multiple workflow instances and to initialize new ones. Such mashup instances are in this thesis also referred to as mashup sessions.

Finally, a solution is proposed for supporting mashup state sharing, preservation and restoration. Sharing states with other users is the key mechanism for facilitating interaction and collaboration between multiple users. State preservation and restoration are needed to allow a user to stop using the mashup and to resume to the same state at a later time.

The proposed solution is also validated through a proof of concept application.

1 Introduction

With the emergence of Web 2.0 the software industry has shifted towards web-based software. O'Reilly [1] described this as entering a period of user interface innovation, since web developers are finally able to build web applications as rich as local PC-based applications. Taivalsaari and Mikkonen [2] pointed out two important characteristics of such applications: collaboration and interaction. By collaboration, they refer to the way users can share data, applications, and services over the Web. The other aspect, interaction, describes the way Web 2.0 technologies allow building web sites that support direct manipulation while updating only certain page elements at a time instead of updating the entire page. Such Web 2.0 applications also leverage the potential of combining content from multiple web sites to create new aggregated web sites called mashups. The individual building blocks that a mashup is made of are called widgets in this thesis.

There are a lot of tools available targeted towards mainstream Internet users for creating general-purpose mashups. An example use case for such tools is creation of a simple mashup for calculating the distance between an office and a list of houses for sale in that area to search for a property within the walking distance. Other examples entail applications combining a list of sushi restaurants with their health ratings and organizing a night out. The latter will for instance combine information from Yellow pages to find a movie theater and a restaurant, Google Maps for displaying the locations, and a component for buying tickets for a movie.

However, in addition to mainstream usage, mashups could be used as situational applications for solving specific business user's day-to-day problems as well. Such mashups are called enterprise mashups. Hoyer and Fischer [3] define enterprise mashups as follows: "An Enterprise Mashup is a Web-based resource that combines existing resources, be it content, data or application functionality, from more than one resource in enterprise environments by empowering the actual end-users to create and adapt individual information centric and situational applications." Such mashups are driven by resource composition style by reusing building blocks in different contexts. Furthermore, Gurram et al. [4] argue that the mashup concept provides the flexibility that organizations need to quickly adapt to changes in requirements and processes.

Daniel and Matera [5] argue that although the value-adding combination of components is important for the success of a mashup application, it is also important to understand that mashup application can only be as good as its constituent parts, the components. This is especially critical for enterprise mashups that, with respect to mainstream mashups, require more complex components. The authors propose a set of guidelines for building mashup components by focusing on modularity and reusability. The main principles are the following:

- the complexity should be hidden inside the components and not be exposed to the composer or user;
- components with their own UI should run inside a `<div>`, `<span>`, or `<iframe>` HTML element without impacting other components or the mashup application;
- components' executability and benefit should not depend on whether the component is integrated into the mashup or not;
- components should be developed using standard technologies.

Furthermore, with respect to mainstream mashups, enterprise mashups require more complex inter-component interactions. One technology that supports integration of multiple components within a mashup is OpenAjax Hub 2.0 [6]. The hub provides means for components to safely interact with each other by isolating them into secure sandboxes and allowing communication through a publish-subscribe system. The publish-subscribe features support broadcasting messages with two parameters: `topic` parameter that specifies a string name describing the published message, and `data` parameter that holds the data payload of the message. Subscribers can subscribe to a specific topic by specifying the `topic` parameter or a range of topics by the use of wildcards.

However, the main concern with enterprise mashups is that available mashup frameworks and tools are either too simple to meet all the requirements that these more sophisticated mashups propose, or too complex to support end-user development. Additionally, situational enterprise mashups, being targeted at solving business users' day-to-day problems, need to provide functionality for sharing mashup state between users and also functionality for preserving and restoring the mashup state. Furthermore, for solving everyday business tasks it is important for situational mashups to support multiple

instances of the same mashups so that users can either select an existing mashup instance or initialize a new one.

Dheap and Gladd [7] have proposed a mashup session manager to maintain the mashup session ensuring consistency across mashup execution environments of all mashup session participants. In the proposed solution, a user first creates a mashup and then selects participants for that mashup. This way a mashup session can be initialized and collaboratively used across selected participants. However, this solution lacks the support for role-based mashup decomposition. Inspired by the proposed solution, this thesis also uses mashup instance and mashup session definitions interchangeably.

Hence, the aim of this thesis is to provide an extension to an existing general-purpose mashup framework for supporting development of enterprise mashups while still preserving the main benefits associated with mashup development – agility and simplicity. More specifically, this thesis proposes a solution for role-based decomposition of enterprise mashups with rich interactions to mashups as simple as general-purpose mashups while allowing communication between those roles. Furthermore, a solution is proposed for supporting initializing multiple instances of the same mashup that can be preserved and restored in case user stops mashup usage to resume it at a later time.

In this thesis, role-based mashup decompositions are described in form of views that encapsulate sets of widgets a particular role is associated with. This means that different user roles will have different views of the same mashup while still being able to exchange messages between those mashup views. Another novelty, compared to existing mashup platforms, is support for multiple instances of mashups. Namely, the developed extension allows initializing, preserving and restoring multiple instances of the same mashup. This generic functionality is especially essential for situational mashups where mashup users perform everyday business tasks. The approach proposed in this thesis allows mashup users to create a session for each task, such that task execution can be effectively stopped and resumed whenever needed. Preserving the state of task execution is accomplished without task-specific database support. Furthermore, the mashup state can be shared between the users in different roles, such that multiple users can collaborate on the same session by sharing data, which is supported by the view associated with a particular role.

Inspired by the role partitioning used by Daniel and Matera [5], this thesis aims at keeping the roles of component developer and mashup composer separately. From mashup composer side, the thesis targets at easing the mashup construction process. An extension is proposed to an existing XML-based mashup description language to specify the component model and also which components are going to be used in which mashup view and how they interact. Based on the mashup description file, the mashup generator composes the specified mashup views. Such approach eases applying new business requirements to existing mashups because if a view of some role changes or a new role needs to be added then one would not need to re-do the entire mashup but instead simply change the description file accordingly. Thus, if needed, the component developer can develop new components for an existing mashup which can be accomplished by someone with basic knowledge of JavaScript, CSS and HTML. New components can then be easily integrated into the mashup by the mashup composer by modifying the mashup XML description file accordingly. From mashup developer side this thesis follows the principles proposed by Daniel and Matera [5] for developing the set of components for the framework. The components are built using JavaScript, CSS and HTML, they run inside their own containers, and are designed to be reusable by encapsulating domain-specific knowledge into separate components.

Finally, the proposed approach is validated through a proof of concept application based on a credit management workflow. Components are developed for mashup state preservation and restoration, state sharing between multiple users, and coordinating application-specific data filtering, transformations and aggregations. Implementation includes various inter-widget communication patterns and uses event-based publish-subscribe communication of OpenAjax Hub 2.0.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of alternative solutions and related research. Chapter 3 describes the details of an existing general-purpose mashup framework followed by proposed alterations and extensions to the framework in Chapter 4. Chapter 5 explains in detail the architecture of the solution and implemented components. An overview of the proof of concept application is given in Chapter 6. Finally, Chapter 7 concludes the thesis and discusses future work.

2 Related work

The approach in this thesis combines the key aspects of different mashup approaches – reusability, integration, collaboration, and the mashup state preservation and sharing. To better understand the background of these aspects, this chapter describes how the mashup technology has involved over time.

2.1 Mashup providers

Early mashup tools focused on providing easy and intuitive user interfaces for building mashups. Many such tools took a spreadsheet-like approach for visualizing and manipulating data. Karma [8] retrieves data from web pages by letting users provide a small set of examples. Vegemite [9] uses a similar data extraction approach but also has the ability to augment the dataset with additional data from other web sites. Mashroom [10], on the other hand, provides an interactive presentation of data services. It extends the traditional two-dimensional array of cells to support relational data. While retrieving data from different sources and supporting different means of data manipulation and aggregation, the preceding approaches all use spreadsheet-like presentation to shorten the learning curve of building mashups by providing a work environment already familiar to most end-users. However, these systems only allow displaying the mashup results in a table, hence not providing a way to combine multiple visualization widgets.

In contrast to the spreadsheet-based approaches, Marmite [11] and Intel MashMaker [12] use a data-flow-like model for building mashups. Marmite combines data from existing web content and services, and then allows users to process the data in a flow-like manner. Marmite still uses a spreadsheet view for displaying intermediate states of data between operations, however, the spreadsheet is read-only and thus not meant for data manipulation. MashMaker uses a similar web-based approach. Mashups are created by using browsed web pages as information sources. The created data view can then be extended by live editing and mixing existing data view with computed values from surrounding data. MashMaker also supports collaboration and sharing by allowing users to package their queries as widgets and share them with other users in the community.

With the emergence of Web 2.0 widget-based mashups became even more popular. One such tool is Ousia Weaver [13]. Similarly to MashMaker, Ousia Weaver supports not only creating mashups but also publishing them online for sharing purposes. It automatically assigns a URL to each mashup making the mashup accessible to external users. However, both MashMaker and Ousia Weaver use their own mashup servers making access to existing mashups dependent on third-party software. While still using its own editor for creating dataflows for data collection, aggregation and processing, Ousia Weaver uses widgets to visualize the mashup results. Mashlight [14] is a framework that uses widget-like components already during data collection and processing. These blocks are implemented using Web 2.0 technologies but they include additional information about their inputs and outputs. Furthermore, the inlinks and outlinks are specified for all blocks to indicate in which order they are executed. This determines the mashup process flow. Although Mashlight uses Web 2.0 applications as building blocks, it requires them to be accompanied by a manifest file to be compatible with the framework. Also, while providing support for conditional branching and looping, all possible orders in which Mashlight blocks can be executed need to be specified in the process manifest.

Damia [15] is an enterprise-oriented mashup tool provided by IBM for creating situational enterprise mashups focusing on data aggregation and transformation. Damia enables secure access to data from a variety of desktop and web sources both inside and outside the corporate firewall. It supports various operations on received data including filtering, standardizing, joining, and aggregating. Damia mashups are comprised of a flow of operators which is presented in XML form and fed to the integration engine for compiling and executing the data mashup. Such usage of strongly connected components is a shortcoming of Damia. The publish-subscribe approach taken in this thesis is mentioned as future work for Damia.

One challenge of enterprise mashups is to provide flexibility with respect to adoption to changing requirements of businesses. Daniel and Matera [5] argue that existing mashup tools often either assume the existence of ready-to-use components or only allow creation of new components that work well only within their mashup platform. Thus, to leverage reusability and extensibility, enterprise mashup composition should be based on open standards. Gurram et al. [4] present a proposal for inter-widget communication based on OpenAjax. Widgets in OpenAjax framework are connected by means of loosely coupled

events using publish-subscribe mechanism. While using the same technology for exchanging messages between widgets, this thesis also confronts the problem of data aggregation that is not addressed by OpenAjax framework.

Another challenge with collaborative business-oriented mashups is preserving the widget-based mashup state. In traditional applications, this is achieved by saving the state to an application-specific data model. Preserving the state of a mashup is important to allow users to discontinue their work at any given time and later resume to the state where they stopped using the application. This has been a challenge because the widgets being loosely coupled means that they themselves are unable to capture the state of the mashup. Inspired by the solution proposed by Tammik [16], a similar approach is used in this thesis. However, instead of using a Wookie Widget and Wookie Engine to record, store and later reproduce messages exchanged by widgets, the messages are stored in a separate server-side database providing more flexibility for message handling. Furthermore, this thesis proposes the same widget-based approach for general mashup session handling that includes initialization of new mashup instances and selection of mashup instance amongst existing instances. As this thesis focuses on collaborative mashups then the limitations of Wookie Widget become more evident. The Wookie Widget overwrites messages with identical topics which is a big disadvantage for collaborative mashups that repeatedly need to publish the data they enrich. The approach in this thesis addresses this problem by storing all captured messages in the order they are received. As mashup state is stored by using messages exchanged between widgets then this provides a generic solution for state preservation without needing support for task-specific database. In addition to mashup state preservation and restoration, situational applications targeted at solving users day-to-day problems also need to support multiple instances of the same mashup so that users can either continue working in an existing mashup instance or initialize new ones. This functionality is integrated into the widget that handles mashup state preservation and restoration; and accomplished by assigning unique identifiers to each mashup instance. Whenever a mashup state is restored, only messages from that mashup session are republished.

The final challenge that emerges from collaborative real-time mashups is awareness. Since business-oriented mashups allow multiple users to manipulate the same data then support for workspace awareness is needed. Heinrich et al. [17] present a generic solution for this

in a form of a separate reusable awareness widget. The goal of the solution is to avoid application-specific solutions and instead propose a generic awareness infrastructure that captures information about user interactions at the standardized W3C API layer. The main steps in that approach are initializing registered awareness widgets, pushing collected awareness information from awareness widgets to the server; and receiving, interpreting, and visualizing awareness information from other clients. To record the modifications in the shared workspace, the awareness widgets register to a certain set of DOM events. The exchanged awareness messages are in JSON format. The solution in this thesis uses a similar approach for distributing awareness information among different clients using a central server. However, instead of registering DOM event listeners, this thesis relies on widget publish-subscribe approach provided by OpenAjax Hub to support awareness. The synchronization is done by publishing messages to a common server where other clients can pick them up by being subscribed to certain messages.

Another solution for supporting collaboration between multiple mashup users has been proposed by Dheap and Gladd [7]. Namely, the authors proposed a mashup session manager for supporting distributed multi-user mashup sessions. In their proposed solution, a user who initializes new mashup creation is referred to as a mashup designer. The components the mashup is made of and how the components are wired is defined according to the need contemplated by the mashup designer. Next, the mashup designer can define which other users may participate in the created mashup session. Finally, a unique identifier is associated with the mashup session for populating a mashup application with data associated with that identifier. The mashup session manager also supports asynchronous interactions within the mashup meaning that users can modify their own view of the mashup application by adding new components to it without synchronizing the changes to the other users. Although this means that different users have different views of the mashup, the mashup session manager still keeps the global perspective of the mashup application. This thesis borrows from the solution proposed by Dheap and Gladd [7] for supporting multi-user mashup usage. However, with respect to enterprise mashup applications that target at solving specific business user's day-to-day problems, this thesis decomposes a mashup application into pre-defined role-based views while still supporting collaboration between those views.

2.2 Mashup languages

In addition to tools analyzed in Section 2.1, there has been research about model-based mashup description approaches. There are several mashup tools that use domain-specific programming languages (DSL) developed for solving certain problems in a specific domain. These languages are often built on top of existing programming languages and their goal is to reduce end-user programming efforts.

The Enterprise Mashup Markup Language (EMML) [18] is an XML markup language for describing processing flows of mashups. The language provides features to connect different web services or sites and mash responses into new results. This way mashups written in EMML can produce new data that, in turn, can be used in other EMML mashups. Mashup scripts must be processed by an EMML engine that interprets the EMML statements to run the mashup. Swashup [19], another DSL for mashups, is based on Ruby-On-Rails and is designed for web service mashups (e.g. REST, SOAP, RSS, and Atom). The language unifies the most common service models and facilitates service composition and integration into end-user-oriented web applications. Although both above-mentioned DSLs reduce end-user programming efforts, they still remain too complex for non-programmers due to the extensive vocabulary and need for understanding programming concepts.

Prutsachainimmit et al. [20] propose another DSL to enable cooperation of mobile devices. The authors propose an XML-based description language called C-MAIDL to specify the mashup components and their integration. The specifications are first compiled into Java source code for further processing, then manually compiled into an Android package file, and finally manually installed to a target device. The generated mashup will be executed when the installed application gets invoked by a user. The set of mashup components in that approach is limited to 6 types: Web Application Component for interaction with web pages, Web Service Component for communication with RESTful web services, Mobile Application Component for integrating other mobile applications to the mashup, Arithmetic Component for providing support for pre-defined mathematical operations on results from other components, Cooperation Component to specify cooperation between multiple devices, and Output Component for displaying mashup results as points on a map or as a web page. This model describes mashup components in terms of their names, roles (either subscriber or publisher), and specific types from one of the six described above.

Considering the goal of this thesis to support various mashups components in different views, this approach is too limited as it specifically aims at supporting composition of native mobile applications of specific mobile platforms.

2.3 Mashup meta-models

There has also been research about model-based mashup description approaches. One such approach was proposed by Pietschmann et al. [21]-[22]. The designed meta-model consists of a component model and a composition model. The component model describes components on a non-technical level in terms of properties, events, and operations. Properties are attribute name-value pairs representing components' visible state. Examples of properties include a graph's type (line, bar, pie) and a map's type (normal, satellite, hybrid). Events are used for publishing state changes to other components and they encapsulate data in attribute name-value pairs. Finally, operations are implemented as component methods that are triggered by specific events. Components are described in XML-based Mashup Component Description Language (MCDL). The proposed composition model consists of five sub-models: conceptual model (containing concepts referenced by other sub-models), communication model (defining data and control flow), layout model (for arranging UI components), screen flow model (containing different views of an application), and adaptivity model (defining context-aware behavior). However, due to being focused on completeness rather than simplicity, the proposed approach requires relatively much effort to describe the components.

Imran et al. [23] argue that mashup platforms developed so far have either exposed too much functionality and technicalities to be suitable for non-programmers, or only allow compositions that are too simple and thus of little use for most practical applications. With increased flexibility that is required in both business and personal life, they recognize the need for situational applications. Claiming that it is impractical to design tools that contain powerful and generic functionalities to suit wide range of application domains while being simple enough to be accessible to non-programmers, the authors decided to narrow the focus of their design tool to one well-defined domain, which is related to evaluation of productivity of scientists. The authors propose a generic mashup meta-model for describing a mashup as a set of components, a set of data pipes for carrying data between components, a set of view ports as placeholders inside HTML template; and a layout defining which component is to be rendered in which view port of the template.

Descriptions of components consist of their type, general description, descriptions of input-output ports for carrying data between components, and descriptions of configuration ports for additional configuration of the components. While still being limited when it comes to defining loosely coupled interactions using publish-subscribe system, and defining views for different mashup user roles, the proposed approach is simple enough for non-programmer end-users to model their own mashups.

Finally, Yu et al. [24] aim at reducing efforts required for UI development by maximizing reuse. The authors propose a presentation integration framework, similarly to Pietschmann et al. [21]-[22], in a form of a component model and a composition model. The components are characterized by a state, a set of events, a set of operations, and a set of properties for component configuration. The composition model consists of event subscriptions, data mappings, references to additional integration logic, and layout information. The authors propose an XML-based declarative composition language called Extensible Presentation Integration Language (XPIL). While also not supporting loosely coupled interactions in a publish-subscribe format and views for different mashup user roles, this thesis borrows from the proposed framework for describing the component and the composition models, and extends them to overcome the above-mentioned limitations. Additionally, in this thesis, the XPIL language is extended for describing the mashup models.

3 Background

Yu et al. [24] propose a mashup framework in a form of a component model and a composition model as briefly introduced in Section 2.3. In order to better understand the limitations of the proposed framework in the context of enterprise mashups, this chapter describes in more detail their component and composition models, as well as their XML-based mashup description language. Additionally, to introduce inter-widget communication methods, this chapter gives an overview of component communication patterns proposed by Pietschmann et al. [26]. Finally, the shortcomings of both approaches are summarized forming a set of issues tackled in this thesis.

3.1 Conceptual model

Yu et al. [24] propose an abstract component model to specify the characteristics and behaviors of presentation components, and an event-based composition model to specify the composition model. When describing the presentation integration, there are certain key aspects that the authors point out – simplicity, formality, readability and modularity. Guided by these principles, the authors propose a presentation integration framework as illustrated in Figure 3-1.

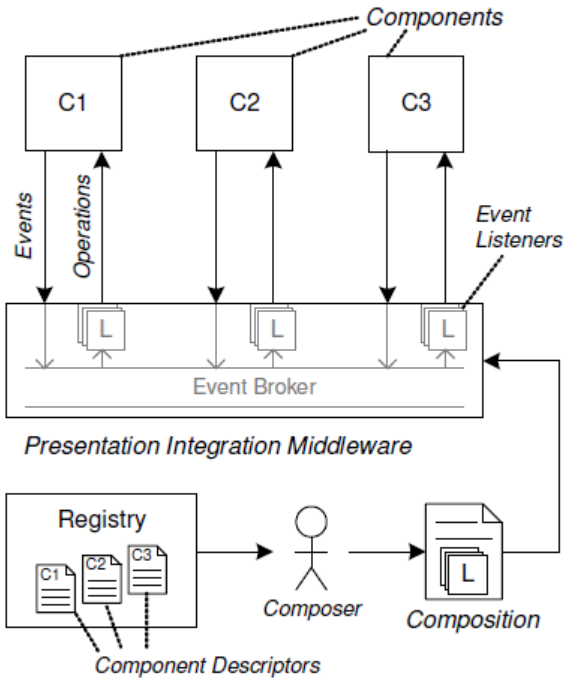

Figure 3-1 *The architecture of the presentation integration framework from [24].*

The composite application consists of the following parts: definitions of one or more components, a specification of the composition model, which describes the integration logic that coordinates the components at runtime, and a middleware to execute the composition. This model has been extended [25] to include not only UI components but also application components (non-visual components communicating with server-side databases, other data services, etc). The following sections describe the proposed framework in more detail.

3.1.1 Component model

The four main elements of the component model are: the state defining what the composite application can see and control in terms of changes in the presentation level, events to express components' state changes, operations to allow events to trigger state changes in other components, and configuration properties to define components' setup information.

Presentation components have a user interface which is the graphical front-end of the component that is rendered to the user. The UI enables the user to interact with the component, upon which the component reacts according to its own internal behavior which may result in state changes. At this point, the component may need to inform the other components in the composition of its state change so that the other components could update their UI accordingly. The attributes of the component's state are generally high level and conceptual, for example, the zoom level of a map.

Events communicate state changes and other relevant information to the composition environment. Events can be initiated either by user actions on the UI or by requests from other components. Event data includes the new state of the component. Other components can then subscribe to these events so that they can update their own state accordingly. The components internally define and implement which native UI events (such as mouse clicks, keystrokes, etc) are handled internally and which trigger component-defined events that will be exposed externally. Only state changes that contain relevant information to other components should be published by component-specific events. In the proposed model, event parameters are presented as attribute name-value pairs.

Operations are methods invoked as a result of events often representing the state change requests. The parameters that the operations consume are also represented as attribute

name-value pairs. In addition to modifying the state changes, operations can be used for querying components' states by defining a return value for a caller to retrieve the result.

The configuration parameters in the proposed model are also presented as attribute name-value pairs including components setup information. These parameters can vary from configuration filters such as user inputs for presentation components to connection options for service components such as the user name and password for login. The semantics of these parameters is entirely component-specific. Hence, the definition is flexible enough to support various usages.

In addition to the four component model elements described above, components can also have attributes that are attribute name-value pairs (i.e. component's type and name).

3.1.2 Composition model

The composition model, proposed by Yu et al. [24], defines all relevant aspects of a mashup, including the components that are used, the communication between them, data transformation logic, and also the layout information for UI components to be positioned properly.

The communication between components is defined in terms of event subscriptions. Events are exchanged through an event broker that facilitates loose coupling of the components. The composition model supports a one-to-many publisher-subscriber relationship between components, which means that one component publishes an event that multiple other components can subscribe to. The publisher-subscriber relationship is defined via event listeners. Each listener has the following attributes: event publisher, event type, event subscriber, and operation of the subscribing component. This means that the defined event type in the event publisher component will trigger the defined operation in the subscriber component. To specify one-to-many relationship, multiple event listeners would need to be defined with multiple event subscribers for a single event type from the event publisher.

The composition model also specifies additional data mappings and transformations inside event listeners in case direct mappings between event parameters and operation parameters are not possible. In the proposed model, the data transformation logic is accomplished by defining inline or external XSLT style sheets in the event listeners to map the event parameters to operation parameters. The additional integration logics can also be described

in the event listeners in the form of simple inline scripts or references to external code. The supported scripting depends on the implementation of the middleware.

Finally, the composition model includes the layout information. However, it does not define any layout mechanism itself but instead supports the notation of external layout managers. This means that the layout information is not interpreted by the middleware but instead simply passed to the external layout manager at runtime. Using the presentation component properties, such as width and height, the middleware will combine them with the layout specification and pass this information to the external layout manager, which will then be able to properly position the presentation components at runtime.

3.1.3 Language representation

The mashup meta-model language representation proposed by Yu et al. [24] contains two sets of XML elements: one for describing the component model, and the other for describing the composition model.

The component model consists of a list of component descriptors which are the `<component>` elements. Each component can have multiple `<event>` and `<operation>` elements identified by name attributes as illustrated in Example 3-1. The configuration parameters are defined in `<config>` element as a list of `<option>` elements.

Example 3-1 *The component model descriptors from [24].*

```
<component id="parkListing"
  xmlns:cm=http://www.openxup.org/2006/xpil/component
  adapter="org.openxup.adapter.SackAdapter"
  address="http://www.nps.gov/findapark/index.htm">
  <event name="ParkSelectionChanged" address="selectPark">
    <param element="nps:parkName"/>
  </event>
</component>
<component id="imageDisplayer"
  xmlns:cm=http://openxup.org/2006/08/xpil/component
  adapter="org.openxup.adapter.dotNETCompAdapter"
  address="http://.../FlickrNet.dll">
  <operation name="search" address="PhotosSearch">
    <input element="nps:tags"/>
  </operation>
</component>
```

The composition model consists of a list of event listeners, which are defined in `<listener>` elements, and layout information, which is defined in `<layout>` element.

Publisher, event, subscriber and operation are defined as arguments in the `<listener>` elements as illustrated in Example 3-2.

***Example 3-2** The composition model description from [24].*

```
<listener id="parkChangedImgListener"
  xmlns=http://www.openxup.org/2006/08/xpil/integration
  publisher="parkListing"
  event="ParkSelectionChanged"
  subscriber="imageDisplayer"
  operation="search"
/>
<layout manager="CSS2" xmlns="http:.../xpil/integration">
  ...
</layout>
```

3.2 Communication patterns

Pietschmann et al. [26] argue that although mashup development has moved towards presentation layer enabling lightweight combination of distributed backend and frontend resources, the communication patterns are still too basic and do not support the coordination needs implied by the UI. The authors propose a new approach of modeling rich communication patterns as part of the mashup composition model. In addition to traditional one-way communication, the patterns include synchronization between widgets and asynchronous data requests to backend services. The authors introduce definitions for unidirectional, bidirectional and synchronization connections as links, backlinks and propertylinks.

The basic type of unidirectional communication is represented by links. Each link is defined with parameters it carries and can only be connected to an operation that has semantically the same parameters. No response is expected, thus, a one-way communication is established.

Bidirectional request-response connections are represented by backlinks. These are typically used for receiving asynchronous updates from backend services. Similarly to links, a target operation is invoked when a respective event is published. However, upon completion, the target component issues a callback event with a return message. The callback event in turn invokes the callback operation in the source component.

The synchronization between stateful components is accomplished by using propertylinks for connecting their properties. They are used to align the state of two or more components

that are able to understand the meaning of the event and its parameters. As an example, the propertylinks can be used for synchronizing the filters of different components that share the same data. The propertylinks are mapped to links between change events and setter operations of the corresponding properties.

3.3 Shortcomings

The biggest shortcoming of the mashup framework described in Section 3.1, in the context of enterprise mashups, is that it does not consider the concept of roles. This is needed to support multi-user interactions and to decompose such complex mashups into simpler role-specific views, which will be perceived as separate mashups by their users. Another limitation of the framework is that it does not specify means for initializing, preserving and restoring multiple instances of the same mashup. This functionality is essential for situational mashups where mashup users perform everyday business tasks. Mashup users need to be able to create a session for each task, such that task execution can be effectively stopped and resumed whenever needed such as supported by workflow execution engines.

Additionally, loose coupling in the proposed framework is achieved by defining event listeners in the composition model. However, this approach exposes too much component's inner logic in the composition model. Instead, by using OpenAjax topic-based publish-subscribe system, the topics could be specified in the component model, and components themselves could handle when the messages should be published or how to respond to the messages they have subscribed to. This way the mashup composer would not need to know exactly which operations certain components support and what parameters they consume.

Other limitations arise from the fact that event, operation and configuration parameters are all represented as attribute name-value pairs. This is a simple but somewhat limited approach which becomes a problem in case of more complex data structures such as lists.

The communication patterns described in Section 3.2 also exhibit some limitations. With situational enterprise mashups, additional patterns are needed to facilitate interaction and collaboration between users potentially using different views. It becomes necessary to specify internal communication that will not be shared with other views and external communication that will be shared with other views.

4 Solution

This chapter describes solutions that this thesis proposes to overcome the shortcomings of the mashup framework and communication patterns described in Section 3.3. Firstly, extensions and alterations to the mashup meta-model proposed by Yu et al. [24] are described including the component and the composition models. Secondly, the formalization proposed by Wilson et al. [27] is extended to capture the meta-model. Thirdly, extensions are proposed to the XML-based mashup markup language by Yu et al. [24] followed by description of extensions to the communication patterns by Pietschmann et al. [26]. Finally, an overview is given of widget types supported by the mashup framework proposed in this thesis. These widget types are categorized with respect to categorizations proposed by Yu et al. [28] and Imran et al. [23].

4.1 Mashup framework

4.1.1 Component and composition models

The framework described in Section 3.1 is extended with the concept of roles to support multi-user interaction and decomposition of complex enterprise mashups into simpler role-specific views. In the context of this thesis a view defines a subset of widgets a mashup is made of. Each view has an identifier (`id`) and a name (`name`) attribute, the first defining the unique identifier for the view within a composition, the latter giving a short formal description of the view. Views define a list of components, with references to the components defined in the component model, which would be included in a particular mashup view. The list of components includes both the presentation components and also the non-visual application components.

Secondly, the framework is extended to support multiple instances of the same mashup. Each mashup instance is assigned a unique identifier that is used to identify which messages belong to which mashup instance. Users can either select an existing mashup instance or initialize a new one. Whenever a mashup state is restored then the messages that need to be replayed can be located by the mashup instance identifier.

In this thesis, OpenAjax Hub fills the role of the event broker proposed in the framework by Yu et al. [24] described in Section 3.1. The component model already defines what topics the components publish or subscribe to, hence, when using OpenAjax Hub, there is no need to additionally define event listeners. Components are only allowed to communicate through Hub's messaging bus, thus, ensuring safe inter-component communication.

Additionally, to overcome the limitation of simple attribute name-value pair event and operation parameter formats, this thesis uses message topics to define the inputs and outputs of the components. Events trigger messages with certain topics to be published. Which events trigger which topics is defined in the underlying component model. Similarly to events, there is no need for specifying the operation parameters as attribute name-value pairs as the message topic already defines the outgoing messages structure. To simplify the component model, the logic about what functionality is triggered upon receiving each topic is defined in the components internal logic. Thus, in this thesis, the component model does not specify each input operation with its parameters but instead only the input topic names. Similarly to the proposed framework described in Section 3.1, in this thesis the UI components use configuration parameters to add additional elements to the component, such as buttons to trigger certain events, or data input fields. Configuration parameters are also used in this thesis to define necessary setup for querying SOAP web services.

Finally, as a minor alteration to the mashup framework described in Section 3.1, the composition model in this thesis itself does not contain any layout information. Instead, this is handled by the automated rule-based selection of layout templates functionality [29]. This layout selection functionality is triggered by the mashup generator during runtime.

4.1.2 Formalization

Wilson et al. [27] define mashup as a tuple $m = \langle L, W, VA \rangle$ with:

- $L = \langle l, V \rangle$ being the layout of the mashup, where l is the layout template and $V = \{v_i\}$ is a set of viewports inside l where widgets can be rendered;
- $W = \langle w_i \rangle$ being the set of widgets in the mashup; and
- $VA = \langle va_k | va_k \in W \times V \rangle$ being the set of widget-viewport associations needed for placing and rendering the widgets inside the mashup.

In this thesis, the layout templates with viewports are constructed by the automatic rule-based template selection application [29], described in more detail in Section 5.1.10, which automatically assigns a set of widgets to appropriate viewports.

Wilson et al. [27] define inter-widget communication through a set of events that the widgets generate, and a set of operations supported by the widgets which are then connected to a message flow. In this thesis, the communication is handled by OpenAjax Hub using input and output message topics. Hence, there is no need to specify direct inter-widget communications as defined by Wilson et al. [27]. This thesis proposes a widget definition for OpenAjax Hub widgets as $w = \langle id, name, I, O, C \rangle$ with:

- $I = \{i_j\}$ being the set of input message topics;
- $O = \{o_j\}$ being the set of output message topics; and
- $C = \{c_j\}$ being the set of configuration options.

Wilson et al. [27] define configuration options as attribute name-value pairs and state that the semantics of these parameters is entirely component-specific. In this thesis, these parameters are defined as a list of attribute name-value pairs that is simply passed to the mashup component. An exception is the `publishtopic` attribute that is expected to be an output message topic identifier, and is replaced by the topic name it references.

Furthermore, to support role-based multi-instance mashups, this thesis proposes a definition for mashup application model a as $a = \langle V, S \rangle$. Application consists of a set of views *V* and a set of sessions *S*.

- A view $V = \langle R, M \rangle$, where $R = \{r_i\}$ being a set of roles and M being a mashup associated with the particular view.
- A session $S = \{s_i\}$ being a set of sessions.

Hence, there may be different views of a mashup for each role consisting of different widgets. However, the mashup views still share the same mashup sessions, meaning that information can be shared between the individual role-based mashup views. Additionally, there may be multiple mashup instances initialized forming a set of mashup sessions for a particular set of mashup views. Finally, the various mashup views and sessions form a mashup application.

4.2 Language representation

Based on the proposed extensions and alterations to the mashup framework, this thesis also proposes respective changes to the XML-based mashup markup language XPIL proposed by Yu et al. [24] that was described in more detail in Section 3.1.3.

This thesis proposes a new set of elements to the XPIL language proposed by Yu et al. [24] to capture the concept of mashups roles. For each role a view is defined represented by `<view>` element. Views have an `id` attribute as the unique identifier within the composition model for middleware to locate the components that need to be loaded for each view. The `name` attribute is simply a short formal descriptor of the component. Each view must have at least one `<component>` element defined as shown in Example 4-1. The component `id` attribute refers to a component defined in the component model.

Example 4-1 *Composition view description*

```
<view id="customer" name="Customer">
  <components>
    <component id="companysummary" />
  </components>
</view>
```

Similarly to XPIL language proposed by Yu et al. [24], this thesis uses `<component>` elements to describe the components in the component model. However, component inputs and outputs in this thesis are defined by input and output message topics. Hence, instead of using `<event>` and `<operation>` elements as proposed by Yu et al. [24], this thesis uses `<input>` and `<output>` elements to define the topics that the components publish and subscribe to. Each component should have at least one `<input>` or `<output>` topic defined unless the topics are instead specified in the component's internal logic. Example 4-2 shows an example of a component description.

Example 4-2 *Component description*

```
<component id="timetrack" name="Time Tracking" source="Table/Table">
  <input topic="CreditManager.Data.TimeTrack" />
</component>
```

In the XPIL language proposed by Yu et al. [24], components can also have attributes that are attribute name-value pairs. In this thesis, components have three attributes defined: `id`, `name`, and `source`. The `id` attribute defines the unique identifier within the mashup by which the component could be referred to from the composition model description.

Component `name` attribute is not used by the middleware but simply passed to the component's inner logic. In this thesis, it is used for presentation components to describe the component by displaying their name in the HTML `<legend>` tag that surrounds the component. The component `source` attribute refers to the widget location in the file system. In Example 4-2, it refers to the Table Widget implementation location path under Table directory.

This thesis borrows from the XML language proposed by Yu et al. [24] when describing optional configuration elements in the component model. In this thesis, these are used as additional pre-defined HTML elements that would be added to the visualization components, or as configuration parameters for querying SOAP web services. The only attribute handled by the mashup framework is the `publishtopic` attribute. This refers to the corresponding output topic and is translated into the defined output topic name. The other attributes are simply passed to the respective components.

Example 4-3 shows an example set of configuration items that are translated into HTML elements by the components. There are two text area input fields defined, first for entering call answered by information, and latter for call notes. The third configuration item is a button that publishes a topic with a specified `id`. Before passing the configuration options to the `callclient` component, the mashup framework translates the `publishtopic` attribute of the third configuration option into the respective output topic which is `CreditManager.Data.CallComplete`.

Example 4-3 *Component configuration options*

```
<component id="callclient" name="Call client" source="Page/Page">
  <input topic="CreditManager.Data.CallClient" />
  <input topic="CreditManager.Data.CallComplete" />
  <output id="callcomplete"
          publishtopic="CreditManager.Data.CallComplete" />

  <config>
    <option type="textarea" text="Call asnwered" name="callansweredby" />
    <option type="textarea" text="Call notes" name="callnotes" />
    <option type="button" value="Done" publishtopic="callcomplete" />
  </config>
</component>
```

Another example of configuration option definition is shown in Example 4-4 on the next page. The configuration option defines three attributes that are needed for a SOAP widget:

`wsdl`, `operation`, and `proxy`. Again, all three are simply passed to the SOAP component and handled by the component internally.

Example 4-4 *SOAP widget configuration options*

```
<config>
  <option wsdl=http://sample.org/WSDLs/InterfaceService.wsdl
          operation="getOrganizationDetails"
          proxy="http://sample.org/proxywidget"/>
</config>
```

4.3 Communication patterns

In this thesis, there are four types of message topics used for inter-widget communication. Data topic is the basic topic type for exchanging messages inside individual user's mashup view. These messages are also picked up by the State Widget and are saved to the server-side database so that mashup state could later be restored. State Widget is a mashup framework widget that handles mashup state saving, restoring and sharing between different views. The functionality of the State Widget is described in more detail in Section 5.1.6. The messages with topic Data correspond to the links in the classification proposed by Pietschmann et al. [26] described in Section 3.2. Another message topic type is Property topic which is used for synchronization of visual components' filters. These correspond to the propertylinks described in Section 3.2. To support interaction and collaboration between users, two additional communication types are proposed in this thesis: OutData and InternalData. The need for the latter comes from the fact that not all communication needs to be shared with other users, for example, data transformation messages. The OutData topic needs to be separated from the regular Data topic to avoid distribution of messages to the originator mashup view.

4.4 Widget categorization

This thesis proposes two widgets for mashup state and session handling. First is State Widget that is responsible for mashup state sharing between different views and also state preservation and restoration. The latter is Session Widget that handles mashup instance selection amongst a list of existing mashup instances and new mashup instance initialization. To better outline the differences of these two proposed widgets with respect to other widgets used in the mashup framework, this section categorizes the widgets using categories proposed by Yu et al. [28] and Imran et al. [23].

Based on their type, the components can be divided into three groups as suggested by Yu et al. [28]: data components that act as pure data sources, application logic components that provide access to application logic, and user interface components that provide GUI to the users. In this thesis, the widgets querying SOAP web services fall into the first category. Additionally, data widgets needed to output mashup initialization data (e.g. access key for SOAP web services) would be categorized as data components. All application-specific transformations and aggregations are handled in one widget – the Controller Widget. With respect to categorization by Yu et al. [28], the Controller Widget falls under the application logic component category. Finally, various visualization components that fall under UI component category in this thesis include tables, forms, bar charts, line charts, and pie charts.

This thesis proposes a fourth category to group framework components which are non-visual components that do not contain any application-specific logic. Instead, they provide sessioning functionality that is needed for situational enterprise mashups, and general data transformations. In this thesis, the State Widget is considered a framework component as it is responsible for state sharing between different views and also preservation and restoration of the mashup state. Additionally, the Session Widget that enables session selection and new session initialization would fall under the framework component category. Finally, the Transformer Widget proposed by Villido [30] is categorized as framework component as it performs data transformations needed for SOAP widgets without itself containing any application-specific logic.

Imran et al. [23], on the other hand, categorize components from the data flow perspective. They define that each component must have at least an input or an output port. Components with no input ports are defined as information sources. Components with no output ports are called information sinks. Components with both input and output ports are called information processors. In this thesis, most of the components are information processors as they act as subscribers and publishers. Even SOAP widgets are considered information processors by this definition as they receive certain parameters as input based on which they perform SOAP operations and return results. The exceptions are some visualization widgets that only receive data to display it, and data widgets that only publish mashup initialization information (e.g. access key for SOAP web services).

Table 4-1 visualizes the components used in this thesis with respect to both categorizations described on the preceding page.

Data components	Data widgets		SOAP proxy widgets
Application logic components			Controller Widget
User interface components		Table widgets; Chart widgets (pie, line, bar)	Form widgets
Framework components			State Widget; Session Widget; Transformer Widget
	Information sources	**Information sinks**	**Information processors**

Table 4-1 *The component classification with respect to [28] and [23].*

5 Implementation

The proposed solution is a role-based multi-instance mashup framework. It is described using the 4+1 View Model of Architecture Framework [31]. The 4+1 View Model describes software architecture using five concurrent views, each covering a specific set of concerns. The four views are logical, development, process and physical view. The fifth view is an addition to the four views describing the architecture by using a small set of use cases or scenarios. Tailoring the model to better suit the size of the mashup framework solution implemented in this thesis, only logical and process views are presented while other views are omitted. The following sections first introduce the components that are used in the implementation of the framework and the way they should be set up, and then present the logical and process views of the architecture.

5.1 Components

5.1.1 OpenAjax Hub

OpenAjax Hub [6] is a set of standard JavaScript functionality that addresses key interoperability and security issues that arise when multiple Ajax libraries and/or components are used within the same web page. The specification was developed by OpenAjax Alliance [32], an organization of vendors, open source projects and companies that are dedicated to adopt interoperable Ajax-based web technologies. The key aspect of OpenAjax Hub is the publish-subscribe engine that includes a "Managed Hub" mechanism allowing a host application to isolate untrusted components into secure sandboxes. In this thesis, OpenAjax Hub is used for establishing inter-widget communication. Furthermore, it is used to store the mashup state by recording messages that are exchanged between widgets so that later these messages could be republished to restore the mashup state.

5.1.2 OpenAjax Metadata 1.0

OpenAjax Metadata [33] represents a set of industry-standard metadata defined by OpenAjax Alliance that enhances interoperability across Ajax toolkits and Ajax products. The OpenAjax Metadata files are XML-format files containing widget definition and the resources it uses. The metadata file name must have a suffix “oam.xml”. All widgets in this thesis are accompanied by a corresponding OpenAjax metadata files. This is needed for

supplying necessary information about the widgets to the external automatic rule-based layout template selection application, which is described in more detail in Section 5.1.10.

5.1.3 NodeJS

NodeJS [34] is a server-side software system built on Chrome's JavaScript Runtime. It enables developing an application using one language – JavaScript – on both the server and the client side. Latest version of NodeJS can be downloaded from NodeJS web page and installed as shown in Example 5-1. As of NodeJS version 0.6.3, the node package manager (npm) [35] is deployed and automatically installed with the environment [36]. It runs through command line and manages dependencies for an application. In this thesis, NodeJS is used to run Socket.io for sharing messages between multiple mashup users.

Example 5-1 *NodeJS installation*

```
./configure
make
make install
```

5.1.4 Socket.io

When multiple users are operating in the same mashup instance at the same time then it becomes necessary to synchronize the mashup state between the views associated with these user roles. In this thesis, Socket.io [37] is used to provide this functionality. Socket.io is a JavaScript library for real time web applications that consists of two parts: a client-side library running in a browser, and a server-side library that runs using NodeJS. Socket.io is event-driven and also supports broadcasting messages to multiple sockets. It can be installed using the node package manager tool as shown in Example 5-2.

Example 5-2 *Socket.io installation*

```
npm install socket.io
```

The Socket.io client-side, implemented in the State Widget, connects to the server-side Socket.io and upon receiving a topic it has been set up to listen to, publishes the message to components in user's own mashup view using OpenAjax Hub. In addition to standard `'connection'` and `'disconnect'` events, Socket.io supports emitting custom events. Example 5-3 illustrates a sample of client-side Socket.io functionality. First, a new connection is established to a URI specified as the first parameter. The second parameter is a list of additional options. The option specified in Example 5-3 is `resource` that defines the Socket.io root directory. This is useful if direct port cannot be used for Socket.io and a

proxy needs to be set up. In that case, the `resource` option is added to the connection URL. If a direct port can be used, then instead of `resource` option a `port` option can be defined. Finally, when the new connection has been established a callback function is defined for `CustomEvent` event.

Example 5-3 *Socket.io client-side*

```
socket = new io.connect('http://sample.org/', {resource:"socket.io"});
socket.on('CustomEvent', function(data) {
  ..
});
```

The server-side script is used to distribute messages to other connected users. A `broadcast` flag is added to `emit` method call to forward messages to everyone except to the socket that started it. Example 5-4 shows a sample of a server-side Socket.io script. Upon receiving a message `'CustomEvent'` from port 80, it is forwarded to all other connected sockets except the socket where the message originated from. Transport setting `transports` used in Example 5-4 is needed in case Socket.io is used through a proxy.

Example 5-4 *Socket.io server-side*

```
var io = require('socket.io').listen(80);
io.set('transports', ['xhr-polling']);
io.sockets.on('connection', function (socket) {
  socket.on('CustomEvent', function (data) {
    socket.broadcast.emit('CustomEvent', data);
  });
  socket.on('disconnect', function () { });
});
```

5.1.5 Forever

NodeJS also provides a simple tool Forever [38] for running the server-side script continuously. The tool can be installed using the node package manager as shown in Example 5-5. In this thesis, it is used to run the server-side Socket.io script continuously.

Example 5-5 *Forever installation and running*

```
npm install forever -g
forever start server.js
```

5.1.6 State Widget

State Widget is a non-visual OpenAjax Hub widget implemented in this thesis. The purpose of this widget is to preserve the state of a particular mashup instance and later allow the state to be restored. Additionally, it allows actions of one user in one mashup view to be shared with another user's mashup view.

State Widget listens to certain messages sent inside the mashup view through the OpenAjax Hub by subscribing to topic `'Data.**'` (double asterisk) as illustrated in Example 5-6. The topic sets limitations to which messages are stored to avoid preserving unnecessary messages which in this thesis are messages with `'OutData'`, `'InternalData'` and `'Property'` topics. The `'OutData'` topics are assigned to messages transferred between different users and are skipped because messages from other users would already be stored in the database by the State Widget in the originating mashup view. Additionally, the `'InternalData'` topics are assigned to messages that carry temporary transformation information and are skipped to avoid duplicate messages because these transformation messages would be re-generated when original messages are republished. Finally, the `'Property'` topics are assigned to synchronization messages between particular widget properties and thus by essence do not need to be saved and restored either. The messages with prefix `'Data'` in the topic are continuously stored in a server-side database with their full topic and the mashup session identifier in the order they are received. The content of the message is serialized to JSON format.

Example 5-6 *Handling message saving in the State Widget.*

```
widget.OpenAjax.hub.subscribe("ExampleApplication.Data.**",
  function(topic, message) {
    return widget.saveMessage(topic, JSON.stringify(message), widget);
  }
);
```

When a user resumes mashup usage then State Widget restores the mashup to the state the user stopped the usage. This is done by playing back all messages of the corresponding mashup session when the mashup is re-loaded. The messages are queried from the server-side database and then published to the OpenAjax Hub in the order they were received.

All messages that are saved to the database are also communicated to other users of the mashup. This is another task of the State Widget and is accomplished by using the Socket.io as illustrated in Example 5-7.

Example 5-7 *Handling message distribution in State Widget*

```
widget.socket = new io.connect('http://sample.org/',
  {resource: "socket.io"});
widget.socket.on(ExampleApplication.OutData',
  function(data) {
    widget.OpenAjax.hub.publish(data['topic'], data['message']);
  }
);
```

State Widget connects to the server-side Socket.io and upon receiving a message with `'OutData'` topic from another user the message is published to the other components inside that mashup view using the original message topic.

5.1.7 Proxy Widget

Proxy Widget [39] is a non-visual OpenAjax Hub widget that proxies requests to SOAP web services. It was developed to overcome several obstacles that SOAP request creation on the client-side introduces – the Same Origin Policy that restricts a web page from making requests only to the same domain it originates from, the lack of support for creating SOAP requests in JavaScript, and difficulties in parsing the request result as it requires knowledge of the SOAP message structure.

Proxy Widget consists of two parts - the client-side components to enable communication with SOAP web services via non-visual widgets, and the server-side components to provide metadata for the client-side and to proxy the service requests from client-side to the actual service endpoints. In this thesis, the Proxy Widget is used to enable data widgets to query SOAP web services.

5.1.8 Transformer Widget

Transformer Widget [30] is another non-visual OpenAjax Hub widget that uses semantic integration to transform the exchanged data to be interpretable to other widgets. It listens to all messages exchanged by other widgets that are connected to the hub, and uses preconfigured mappings to identify and combine data elements it receives from these messages. Mappings define the structure and the semantics of the messages. By using the mappings, the Transformer Widget is able to collect data elements from the received messages and generate new messages that can be sent to other widgets that can interpret them. In this thesis, the Proxy Widget relies on Transformation Widget to handle the semantic integration for the SOAP widgets.

5.1.9 Controller Widget

All application-specific data transformations are handled by a single non-visual Controller Widget implemented in this thesis. In the proof of concept application, the Controller Widget is used to transform certain messages to the correct format to be consumed by the chart visualization widgets and the SOAP proxy widgets. For the visualization widgets, the Controller Widget subscribes to certain topics, filters out data that needs to be visualized,

performs needed transformations, and publishes messages that are in the correct format for chart widgets to consume. For the SOAP proxy widgets, the Controller Widget handles data transformations and aggregations needed to query SOAP web services and later combines the result data with existing data in the mashup.

5.1.10 Auto Microsite application

The role-based multi-instance mashup framework itself does not specify any layout templates for the presentation components, but instead uses an external application for automatic rule-based layout template selection [29]. The Auto Microsite application provides a set of layout templates with placeholders for various types of widgets, and a set of rules to direct the widget-template matching process. The automatic layout template selection process is initialized by submitting a URL to the Auto Microsite application. The URL consists of widget metadata file URLs sent with `widget` parameter, and widget parameters sent with `property` parameter as illustrated in Example 5-8.

Example 5-8 *Decoded minimal Auto Microsite input URL*

```
http://sample.org/automicrosite/?
widget[0]=http://sample.org/Application/widgets/Table/Table.oam.xml&
property[0][role]=consultant&
property[0][mashupId]=06271989f416789b3846fc31f7cd3670
```

Based on a set of rules, the Auto Microsite application selects a template that matches the submitted set of widgets and then loads the mashup. Only presentation components need to be assigned to correct placeholders, whereas the non-visual components are attached to the end of the `<body>` element of the HTML document. The application also creates an instance of the OpenAjax Hub and attaches all widgets to it. The end result is a web page with all necessary components loaded and ready for exchanging messages through the OpenAjax Hub.

5.2 Setup

The mashup framework is designed to be part of an existing web application that handles user identification and role assignment. Additionally, the selection of an existing mashup instance or initialization of a new instance is handled by the web application framework. Implementation of the Session Widget is considered as future work of this thesis. This section describes how the framework components should be structured and how the web application should be modified to include the mashup framework.

The mashup framework consists of various components as described in Section 5.1. The way the components should be structured in the file system is illustrated in Table 5-1.

Path from server-side root	Description
/MashupModel.xml	Mashup model description file containing descriptions of components and views
/MashupArea.html	Main file for loading the mashup area of an application
/utils/	Main directory for mashup framework files
/utils/XmlParser.js	Utility functions to ease parsing of MashupModel.xml file
/utils/Generator.js	Utility functions to collect component descriptions for a particular mashup view
/utils/Mashup.js	Main file for generating a mashup URL
/utils/conf.ini	Configuration file for the mashup framework
/widgets/	Main directory for widgets
/widgets/System/	Main directory for mashup framework widgets
/widgets/System/State.oam.xml	State Widget metadata file
/widgets/System/State.js	State Widget implementation
/widgets/System/conf.ini	State Widget configuration file
/socket.io/socket.io.js	Socket.io main library
/socket.io/node_modules/	Directory containing additional Socket.io libraries
/socket.io/server.js	Server-side Socket.io script for distributing messages

Table 5-1 *Structure of the framework components*

There are certain rules that the web application framework needs to follow in order to include the mashup framework. The mashup area needs to be a separate web page. Thus, in order to include it inside an existing web application framework it needs to be wrapped inside an HTML `<iframe>` element as illustrated in Example 5-9.

Example 5-9 *Mashup HTML* `<iframe>` *element*

```
<iframe id="mashupFrame" src="MashupArea.html?
  role=consultant&
  mashupid=bd5382fc182a4f56095457538d3ae6ad"
  user=user@sample.ee&
  style="width:100%;height:100%"
  frameborder="0">
</iframe>
```

The `role` parameter is needed to identify components for the corresponding mashup view, and the `mashupid` parameter is needed to populate the mashup view with respective mashup session data. Third parameter, `user`, is optional. It is simply passed to all mashup view components. In this thesis, the proof of concept application uses the `user` parameter for logging user actions. The last two parameters, `style` and `frameborder`, are standard HTML `<iframe>` element attributes.

The `MashupArea.html` file includes the JQuery library and all required mashup framework utility scripts as illustrated in Example 5-10.

Example 5-10 *Mashup area main HTML file*

```
<script src="utils/jquery-1.9.1.min.js"></script>
<script src="utils/XmlParser.js"></script>
<script src="utils/Generator.js"></script>
<script src="utils/Mashup.js"></script>
```

Additionally, the mashup framework needs to specify the URL for the Auto Microsite application as illustrated in Example 5-11. The Auto Microsite application is responsible for generating the mashup layout.

Example 5-11 *Mashup framework configuration file*

```
[layout_generator]
url = 'http://sample.org/automicrosite'
```

As the State Widget interacts with a server-side database and Socket.io, then there are additional requirements that need to be followed. The connection details for Socket.io and for the database where sessions are stored need to be specified in a configuration file - `conf.ini` – as illustrated in Example 5-12.

Example 5-12 *State Widget configuration file*

```
[session]
hostname = 'localhost'           ; Database hostname for saving sessions
database = 'creditmanager'       ; Database name for saving sessions
username = 'user'                ; Username for connecting to database
password = 'userpassword'        ; Password for connecting to database

[socketio]
hostname = 'http://sample.org/'  ; Hostname where Socket.io is installed.
;port = '8080'                   ; Port that Socket.io binds to. Needs
                                 ; to be same as on server side
                                 ; server.js script.
proxy = 'socketioproxy/socket.io'; If direct port cannot be used, proxy
                                 ; can be specified instead to point to
                                 ; server-side Socket.io.
```

For establishing database connection, the following settings need to be specified: `hostname`, `database`, `username` and `password`. For Socket.io, the `hostname` needs to be specified to point to the host where Socket.io resides. Additionally, either `port` or `proxy` setting needs to be specified. In case a direct port can be used for Socket.io to bind to, then `port` setting should be set. Otherwise, a proxy should be set up and specified using the `proxy` setting.

Finally, the `sessions` table either needs to be manually created or the user, specified in the configuration file, needs to have `create` privilege to enable creation of new tables. The required columns for the `sessions` table are shown in Figure 5-1.

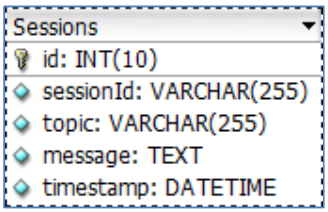

Figure 5-1 *Session table model*

The `sessions` table includes two identifiers: the unique identifier `id` for each message that is saved, and the `sessionid` to identify which messages belong to which mashup view. Additionally, for each message their topic is saved in the `topic` column, and message itself in JSON format in the `message` column. Finally, the timestamp when each message was saved is stored in the `timestamp` column.

5.3 Architectural overview

5.3.1 Logical view

The proposed solution for role-based multi-instance mashup framework consists of client-side and server-side components. The client-side consists of OpenAjax Hub and any number of widgets connected to it. Connected widgets can be non-visual application widgets for handling application-specific transformations and aggregations, or data widgets for querying SOAP web services. Additionally, they can be presentation components for accepting user input or simply visualizing some data to the user. In order to preserve and restore the state of the mashup and to share the state with other users, there is a special framework widget – State Widget – that operates with the server-side. Figure 5-2 on the next page illustrates the conceptual class diagram of the logical view of the architecture.

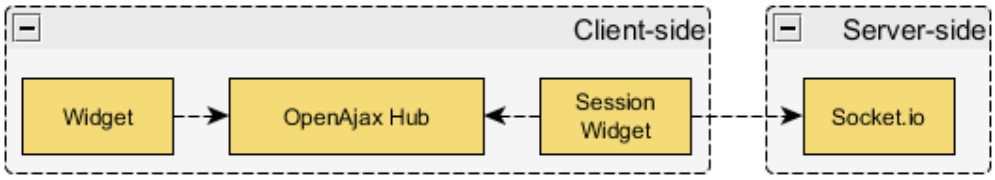

Figure 5-2 *Conceptual class diagram of the solution architecture*

5.3.2 Process view

The mashup construction process involves three parties, the web application framework, the mashup framework, and the Auto Microsite application as illustrated in Figure 5-3. The web application handles user identification, role assignment, session selection amongst existing sessions and also new session initialization.

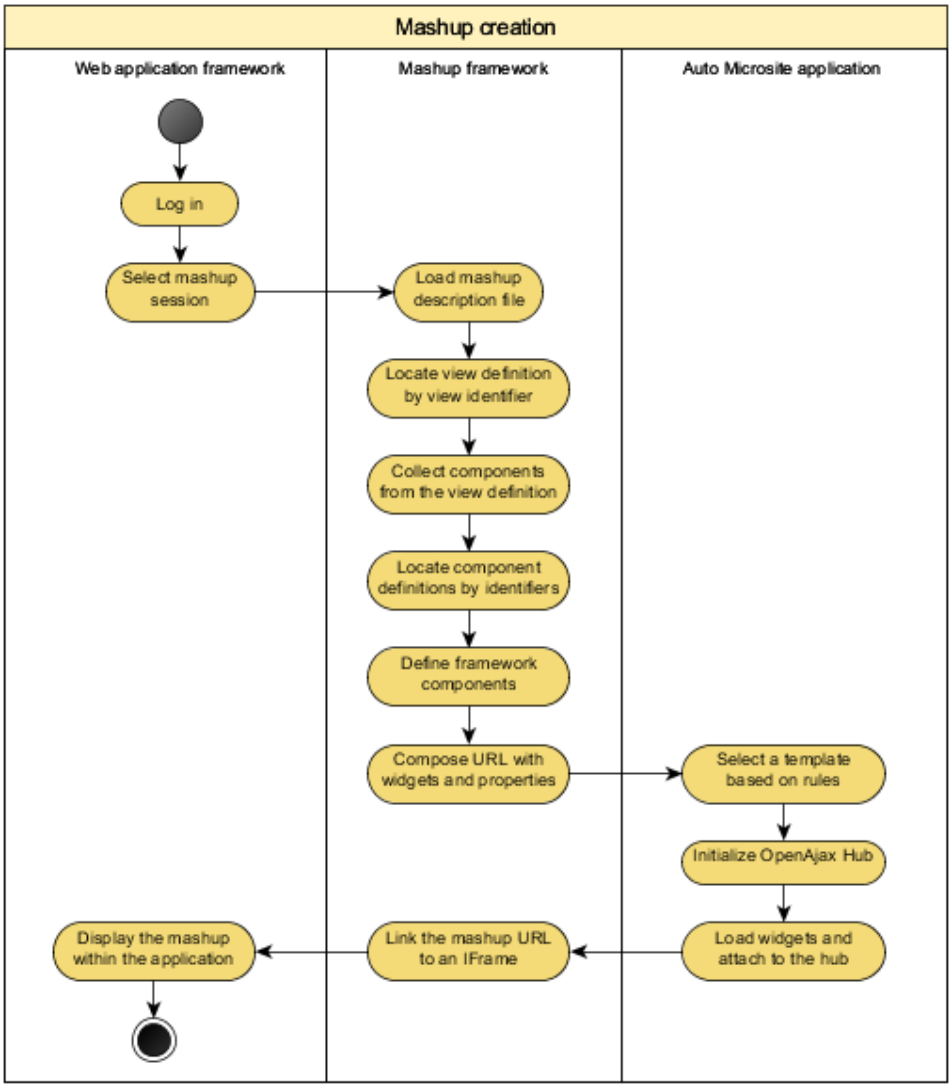

Figure 5-3 *Mashup creation process view*

Each user in the mashup process needs to be associated with a specific role to support multi-user interaction and decomposition of complex mashups into smaller role-specific views. A view defines a subset of components a mashup is made of. All components and views are defined in a mashup description file – `MashupModel.xml` – in XML format. Supporting multiple instance of the same mashup is accomplished by allowing users to either select an existing mashup instance to continue their work or initialize a new one. Each mashup instance is assigned a unique identifier.

Upon loading the `MashupArea.html`, the included `Mashup.js` JavaScript file decodes the web page URL to locate the `user`, `role` and `mashupId` parameters. The `role` parameter is then passed to `Generator.js` to identify the components needed for the particular view. The `Generator.js` uses `XmlParser.js` library to load the `MashupModel.xml` file content into an object to ease parsing of the description file. The view descriptions are located by the `role` parameter. These descriptions define the list of components belonging to that view. The `Generator.js` then collects the component identifiers from that list and by those identifiers locates the component descriptions. For each component the input, output and configuration parameters are gathered and after all components have been loaded the information is passed back to `Mashup.js`. In addition to data, application and presentation components defined in the mashup description file, framework components are added.

Once the necessary components and their parameters have been collected, the `Mashup.js` composes a URL that is then submitted to the Auto Microsite application. The URL consists of widget metadata file URLs sent with `widget` parameter, and widget parameters sent with `property` parameter. The `user` property, that defines the mashup user identifier, is always passed in the URL for each widget. The `mashupId` property, that defines the mashup session identifier, is only added to the State Widget properties.

Additional properties that are passed are: `widgetName`, `subscribeTopics`, `publishTopics` and `configOptions`. The presence of these properties depends on whether the respective information was defined in the mashup description file. Example 5-13 illustrates a sample URL that contains references to one visualization component, which is a table, and one framework component, which is the State Widget.

Example 5-13 *Decoded Auto Microsite input URL*

```
http://sample.org/automicrosite/?
 widget[0]=http://sample.org/Application/widgets/Table/Table.oam.xml&
  property[0][user]=user@sample.ee&
  property[0][widgetName]=Sample Widget&
  property[0][subscribeTopics]=[SubscribeTopicOne,SubscribeTopicTwo]&
  property[0][publishTopics]=[PublishTopicOne,PublishTopicTwo]&
  property[0][configOptions][0][type]=select&
  property[0][configOptions][0][text]=Sample input field&
  property[0][configOptions][0][name]=InputFieldName&
  property[0][configOptions][0][value]=High|Normal|Low&
  property[0][configOptions][0][publishtopic]=publishTopicOne&
 widget[1]=http://sample.org/Application/widgets/System/State.oam.xml&
  property[1][mashupId]=06271989f416789b3846fc31f7cd3670&
```

The end result of the Auto Microsite application is a web page with all necessary components loaded and ready for exchanging messages through the OpenAjax Hub. The resulting web page is rendered in the mashup area `<iframe>` HTML element within the web application.

6 Proof of concept application

6.1 Motivating scenario

The motivating example is based on a credit management workflow. The process involves a customer of a credit company system forwarding its list of invoices, which then would be analyzed by the credit specialist working in the credit company. The analysis involves assignment of a credit risk to each invoice and applying certain actions based on the assigned risk. The customer can check the progress of the analysis at any given time. Additionally, the scenario involves the credit company manager, who can monitor the work progress of the analysis.

6.1.1 Roles

The credit management process involves collaboration of four roles: the credit consultant, the credit specialist, the manager from credit company side, and the customer as separate outside role. Figure 6-1 shows the four roles and the mashup views of each role. The mashup views in the figure only show the visual mashup components.

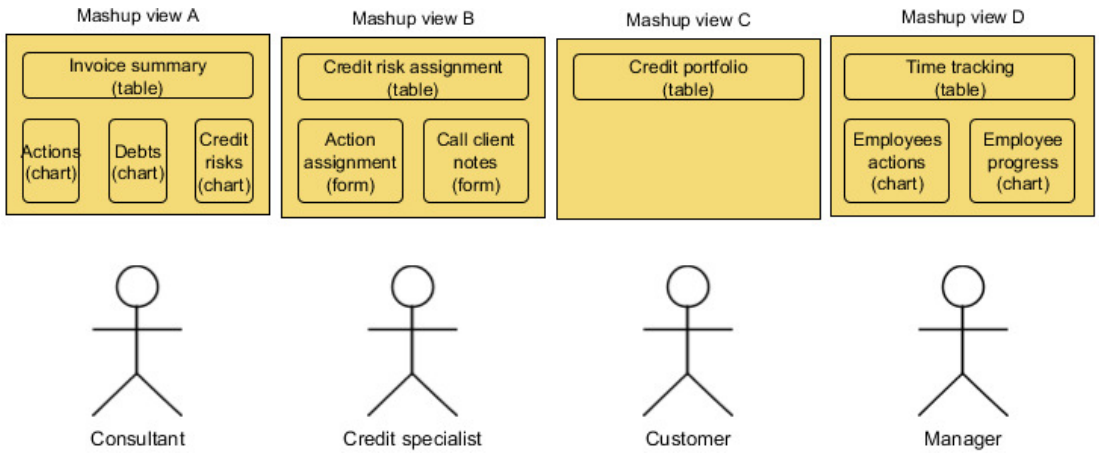

***Figure 6-1** Mashup views of roles*

The credit consultant is a credit company side role, who receives an invoice list from the customer and uploads it to the system. Later on, the consultant will be able to monitor the analysis process. The credit specialist is another credit company side role, who analyses the invoice information and assigns credit risks to each invoice. The credit specialist also carries out specific actions based on the assigned credit risks. The customer will be able to

monitor how many invoices have been analyzed, which credit risks have been assigned, and what actions have been carried out. The fourth role, the credit company manager, will be able to track the work progress of the credit company employees involved in the process.

6.1.2 Process

The detailed credit management workflow is the following:

1. The credit consultant receives a list of invoices from the customer as a CSV formatted file.
2. The credit consultant uploads the file which is then displayed in a table format each row displaying the information of one invoice.
3. Each invoice row is then enhanced with additional information from a SOAP web service which takes the company registration code as an input and outputs further details about the company.
4. Based on the enhanced invoice information, the credit specialist manually assigns a credit risk for each invoice record in the table. The value is in the scale of low, normal, and high.
5. By inspecting the assigned credit risk, a specific action can be assigned to each invoice record. The actions include: adding the invoice record to a monitor list, making a phone call to the invoice receiver, or initiating debt collection.
    a. Monitoring initiates a frame containing visualizations of events of the subject at inforegister.ee monitoring solution.
    b. Phone call action invokes a form for making notes about the call (e.g. who answered the phone and what information was gathered from the call).
    c. Debt collection initiating publishes the invoice data at volaregister.ee via another SOAP web service.

All actions performed by the credit specialist are added to the overall process summary and displayed in the views of the credit consultant and the customer, where respective reports can be downloaded. Additionally, all actions done by the credit specialist (credit risk assignment, assigned action, and call notes) will be logged and displayed as an employee work report in the view of the credit manager. These reports are also available for downloading. Figure 6-2 on the next page shows one use case involving a phone call being issued.

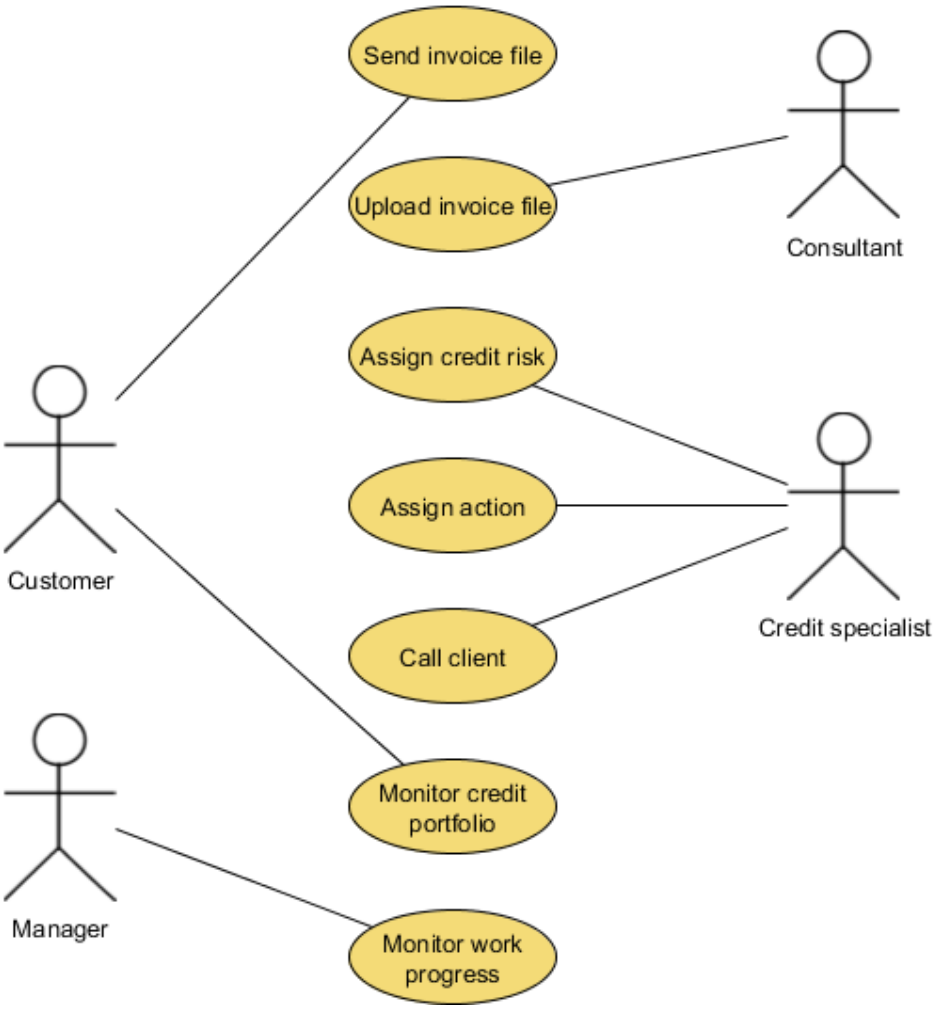

Figure 6-2 *Use case scenario*

6.1.3 Collaboration

As illustrated in Figure 6-3 on the next page, different roles might be working on the same set of invoices at the same time. Hence, they need to be able to share the information within the mashup. For example, as credit specialist is assigning credit risks to the invoices the credit consultant and the customer need to be able to see the information as it becomes available. This is even more important if multiple credit specialists are working on the same invoice list. As one credit specialist assigns credit risks to the invoices, the other will immediately need to see those values and will then either also add credit risk values to other invoice records or, for example, start assigning actions to the invoices that have credit risks assigned, or perform calls to clients.

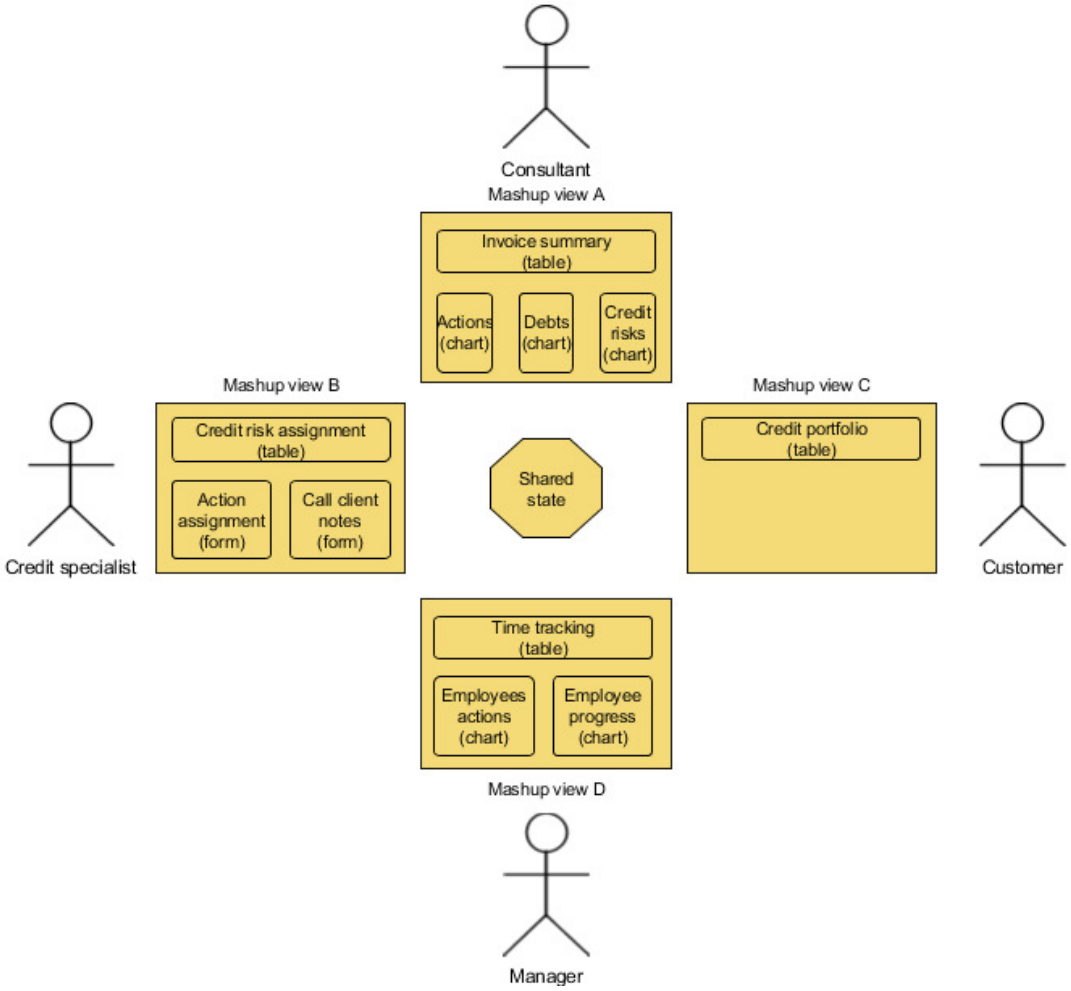

***Figure 6-3** Collaboration*

6.1.4 State

All parties involved in the process need be able to stop their work at any given time, and resume to the same state at a later time. For example, the credit specialist might assign credit risk values to certain set of invoices on one day and continue with the assignments the next day, or assign actions (debt collection, monitor, or call client) on one day and issue those calls the next day. In both cases, the credit specialist needs to be able to see all actions done previously to continue with their work. Also the customer might log in to the credit management system once a day to review the progress and actions performed so far, or the credit company manager reviews employee work progress on weekly bases. Again, in both cases information about all previously done actions needs to be available.

6.1.5 Sessions

The credit management process is divided into sessions by input invoice files to keep actions done to different invoice lists separately. When each role enters the process, as the first step they need to select which session would be loaded. The session selection list in essence is the list of uploaded invoice files and when the user selects a session only the source invoice list and the actions performed on those invoices would be shown. In the proof of concept scenario, the credit consultant is the only role that can upload new invoice files to the system and by doing so initialize a new session.

Additionally, the system only displays sessions that are relevant to the particular user role. For example, each customer role would only be able to see invoice files that they have provided, and each credit consultant would only see invoice files that they have uploaded to the system. On the other hand, the credit manager and the credit specialists would be able to see all invoice files in the sessions list.

6.2 Solution description

Data, application and presentation components, and also views of each role involved in this proof of concept application are all described in the mashup description file – `MashupModel.xml`. Additionally, the message topics that each component publishes or subscribes to are described there. Figure 6-4 on the next page illustrates the mashup composition for the proof of concept scenario including the data, application logic and presentation components, and inter-component communication.

6.2.1 Components

In the proof of concept scenario, there are four types of components: data, application logic, presentation, and framework components.

Presentation components are visual UI components supporting certain user actions. In the proof of concept scenario, there are several types of presentation components. Company summary, credit risk assessment, credit portfolio reporting, and work log reporting components are implemented as table type components as they contain information about a list of incoming records (e.g. invoice records or time tracking information). Action assignment and phone call components are implemented as form widgets with pagination.

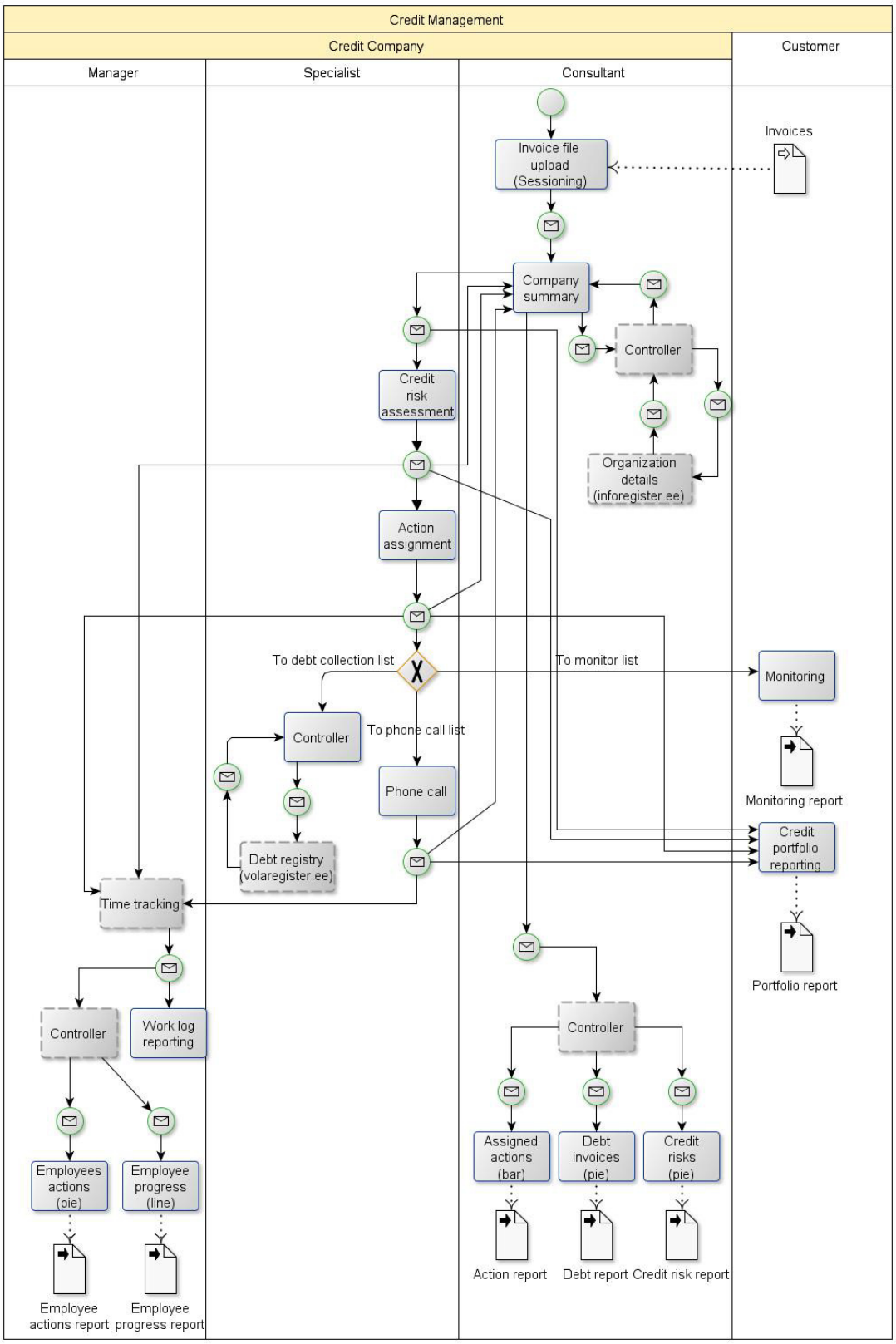

***Figure 6-4** Mashup composition*

These widgets contain information about multiple invoices but only display one at a time for additional user input. They include pagination functionality so that the user can switch pages to review information about the other invoices. Employee actions and employee progress components in the manager role view are both chart type widgets, first is a pie chart and the other is a line chart. Additionally, assigned actions, debt invoices and credit risk components in the credit consultant role view are chart widgets, first is a bar chart and latter two are pie charts.

In the proof of concept scenario, there are two data components that query SOAP web services, namely inforegister.ee and volaregister.ee. The first component is used to gather additional information about a particular company by using company's registration code to query inforegister.ee SOAP web service `getOrganizationDetails` operation. This request is done for each company in the invoice list. The second SOAP component is a debt collection widget that publishes invoice data to a debt registry.

Amongst application components, the proof of concept scenario uses the Controller Widget for handling all application-specific data filtering, transformations, and aggregations. Mainly, the Controller Widget is responsible for modifying certain messages to correct format to be consumed by the chart widgets. Additionally, the Controller Widget handles data aggregations that needed for the SOAP widgets. More precisely, the Controller Widget keeps track of the invoice records to which SOAP queries have been sent. When the SOAP results are asynchronously received back, the Controller Widget combines them with correct records in the invoice record list.

The framework components include the State Widget and the Transformation Widget. First is used for mashup state preservation, restoration, and sharing. The latter is used by the Proxy widget to transform SOAP query result messages. Additionally, a Session Widget was proposed for handling the selection of a mashup instance amongst a list of existing mashup instances, and initialization of a new mashup instance. However, the Session Widget has not been implemented within this thesis and is listed under future work in Chapter 7.

6.2.2 Roles

The proof of concept scenario involves four roles which are defined as mashup views in the composition model. Example 6-1 on the next page shows the mashup view description

of the credit consultant role. The view description includes the `id` and the `name` parameters, and a list of components that would be included in the mashup view. The component `id` attribute values refer to components described in the component model. Example shows that six components would be included in the credit consultant role view, namely controller, company summary, organization deails, action bar, debt pie and credit risk pie widgets. The other three views are described analogously.

Example 6-1 *Credit consultant mashup view description*

```
<view id="consultant" name="Credit consultant">
  <components>
    <component id="controller" />
    <component id="companysummary" />
    <component id="organizationdetails" />
    <component id="actionbar" />
    <component id="debtpie" />
    <component id="creditriskpie" />
  </components>
</view>
```

6.2.3 Process

The process of each role starts with the selection of a mashup session. The credit manager is the only role that can initialize a new session by uploading a new invoice file to the system as illustrated in Figure 6-5. The session selection and new session initializing functionality is implemented using CodeIgniter [40] that is a PHP-based open source web application framework.

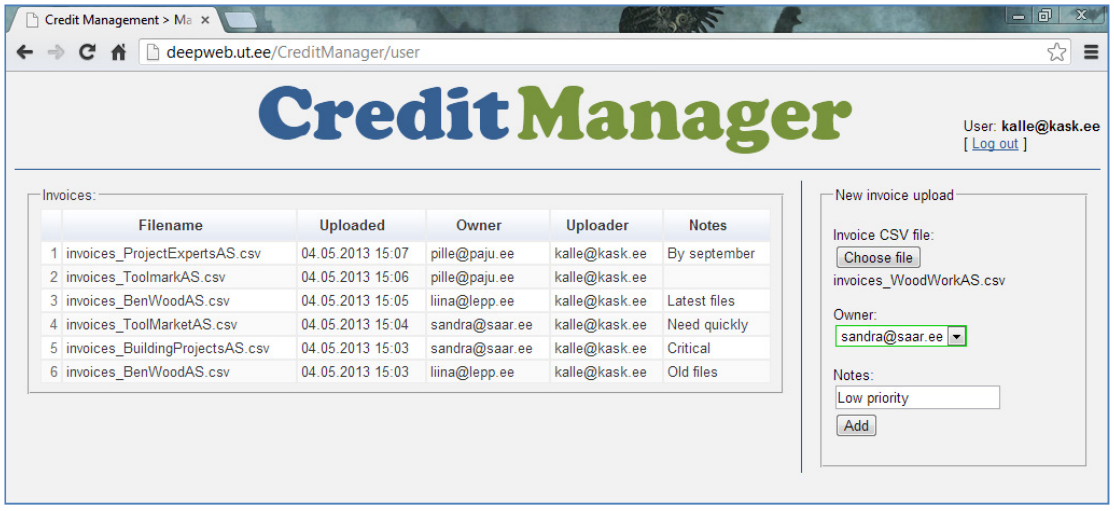

Figure 6-5 *Session selection*

When the invoice file has been uploaded, the credit manager role can start assigning credit risk values to each invoice as illustrated in Figure 6-6 on the next page. Based on the credit risk, the credit specialist can assign one of the three actions to each invoice: add the

invoice record to a monitor list, make a phone call to the invoice receiver, or initiate debt collection. Phone call action invokes an additional form for making notes about the call.

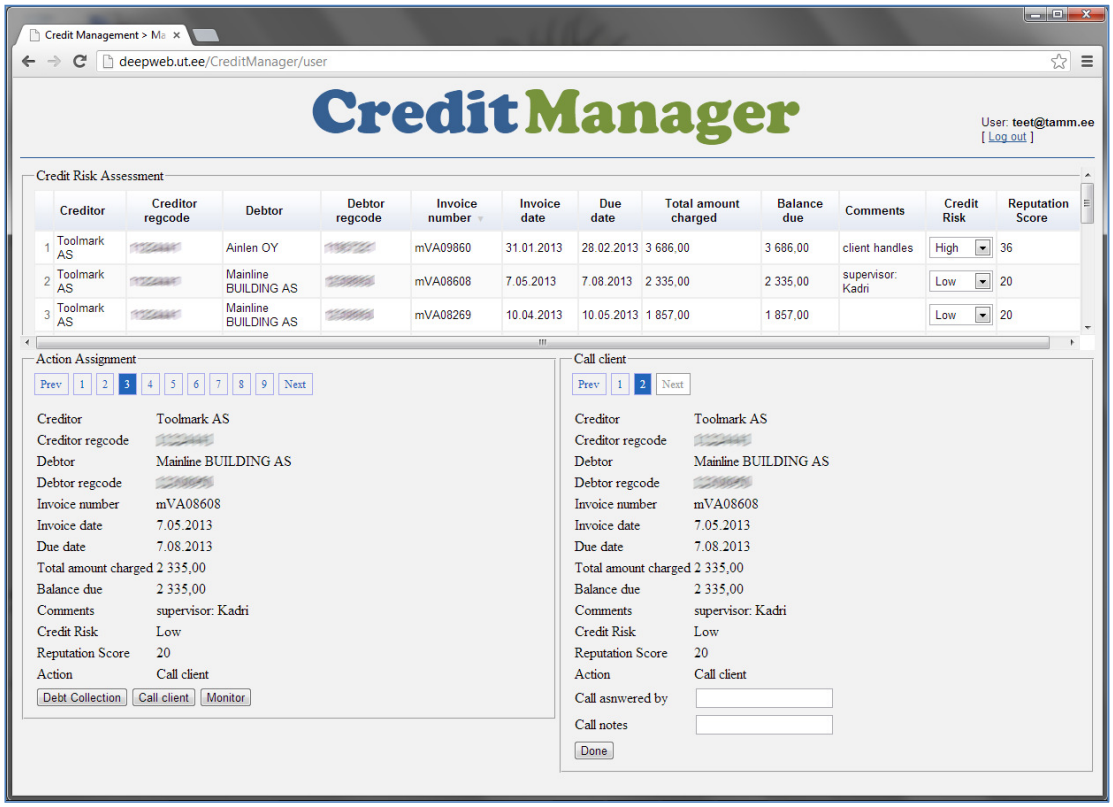

Figure 6-6 *Credit specialist role view*

While the credit specialists are assigning credit risk values and actions to each invoice, the credit consultant will be able to monitor the invoice analysis process. The view of the credit consultant includes a summary table with credit risk and action columns. Additionally, there are three chart components for different visualizations of balance due amounts. The first is a bar chart that combines the balance due amounts of each debtor and then for each debtor visualizes the total balance due amounts per each action. The second is a pie chart that visualizes the total balance due amounts per invoice age, meaning whether the invoice due date has passed and if so then how many months. The third is another pie chart that visualizes the total balance due amounts per each credit risk (e.g. high, normal, low, or not applied). Additionally, all chart widgets have an option to download the visualized data as a CSV-formatted report file. An example view of the credit specialist role is illustrated in Figure 6-7 on the next page.

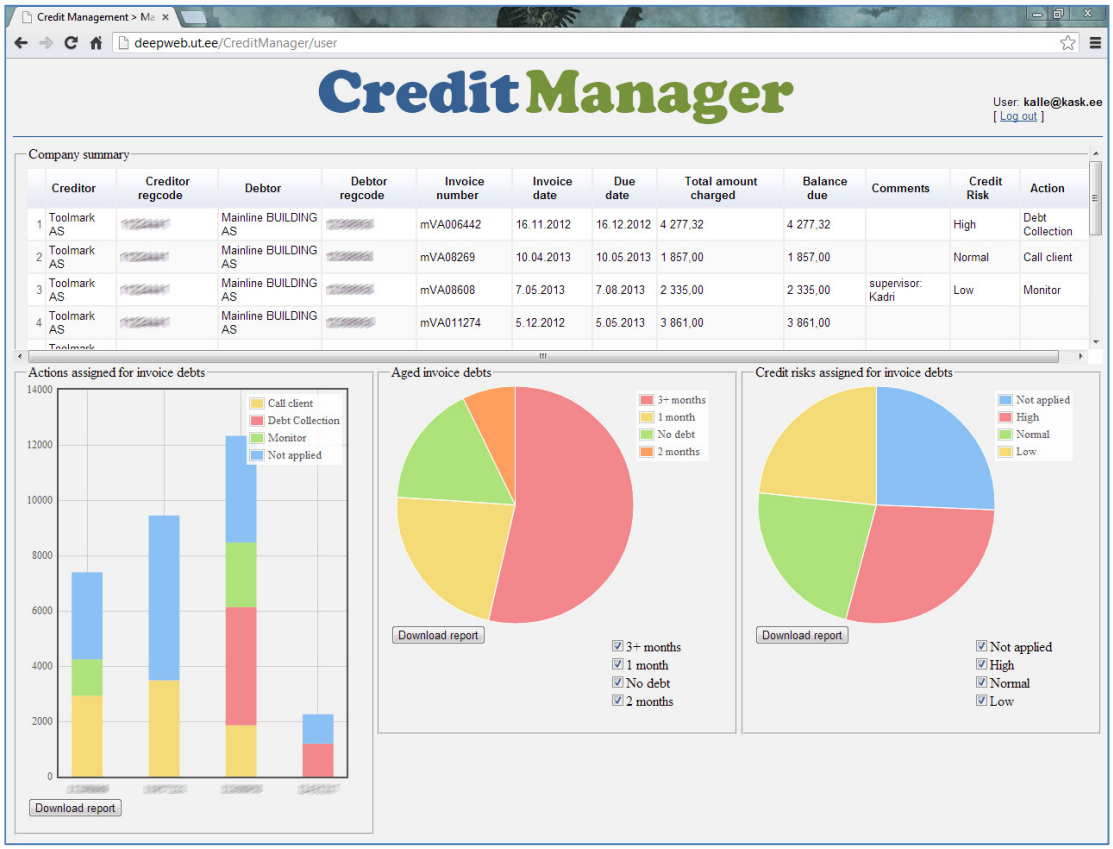

***Figure 6-7** Credit consultant view*

Finally, the credit company manager can monitor the work progress of the credit specialists as illustrated in Figure 6-8 on the next page. The view of the company manager includes a table that lists actions taken by the credit specialists with the action timestamp. The record id column represents the unique identifier of each invoice record. Additionally, the pie chart visualizes the total count of actions taken by each user, and the line chart visualizes the action counts by each user on a date scale.

6.2.4 Mashup state and sessions

In the proof of concept application, when users are working on the same set of invoices (e.g. in the same session), then it becomes necessary to share the actions done by one user in the views of all other users. This is accomplished by the State Widget. This non-visual component listens to all messages exchanged in each users mashup view and through a server side component publishes them to all other users. In order to allow users to stop their work at any given time and resume to the same state at a later time, it is necessary to save the state of the mashup. Mashup state preservation and restoration is also implemented in the State Widget.

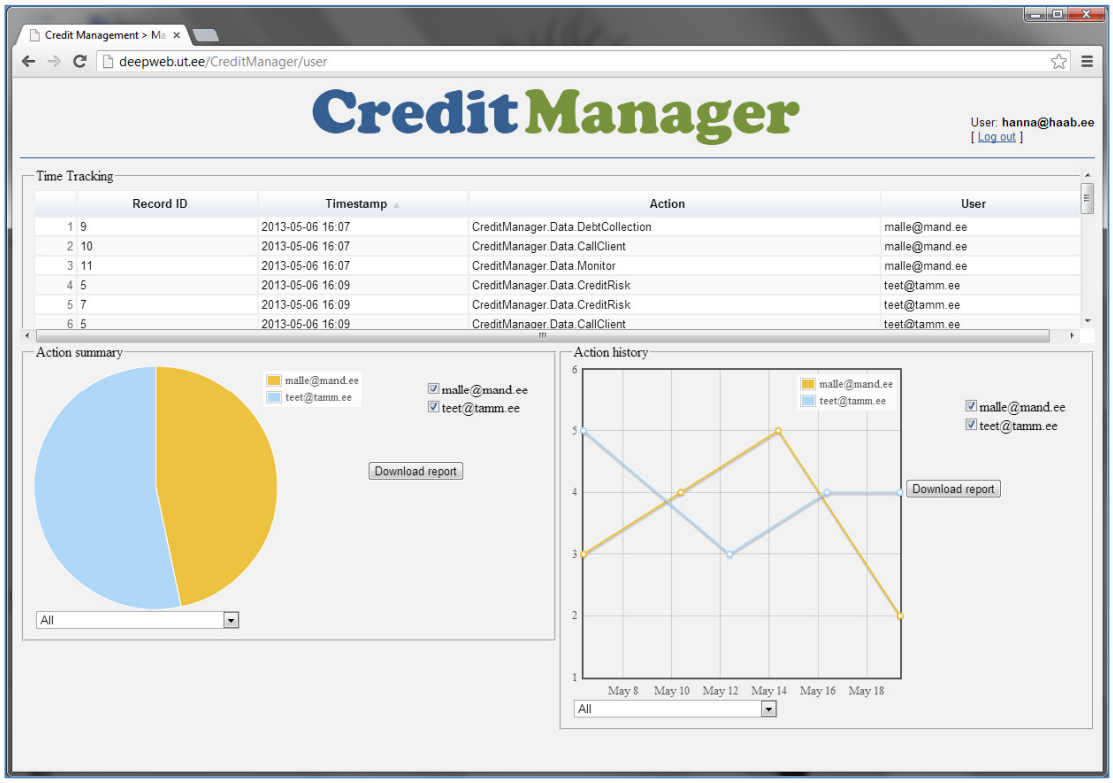

Figure 6-8 *Credit manager view*

7 Conclusion and future work

This thesis extended an existing general-purpose mashup framework with the concept of roles to support multi-user interaction and decomposition of complex enterprise mashups into simpler role-based views. Associating roles with particular mashup views provided an effective mechanism for decomposing enterprise mashups with rich interactions to mashups as simple as general-purpose mashups. Furthermore, this thesis proposed a generic solution for sharing the mashup state between different mashup views, and also preserving the state of a particular mashup view so it could later be restored when the user resumes mashup usage.

Additionally, a solution was proposed for multi-instance mashups. This is essential for enterprise mashups that for solving specific business user's day-to-day problems need to support multiple instances of the same mashup. Finally, based on the proposed extensions necessary components were implemented for the role-based multi-instance mashup framework. The provided solution was validated on a proof of concept application that represents a credit management work flow.

There are several aspects that could be improved in the proposed solution. In this thesis, a State Widget was proposed as a mashup framework widget that handles mashup state preservation, restoration and sharing between different users. Additionally, Session Widget was proposed, but not implemented. The purpose of the Session Widget is handling new mashup instance initialization, instance selection amongst a list of existing mashup instances, and instance deletion in case a mashup instance is no longer needed. In this thesis, the mashup instance initialization and selection were implemented as part of the web application framework.

Additionally, a manual mashup session saving functionality could be implemented. This would provide an effective mechanism for optimizing mashup session restoration. Manually saving a mashup state would remove all existing session data and give a new starting point for the mashup playback by requesting each widget to publish their full state. The State Widget could then catch those messages and save them in the session storage.

Furthermore, mashup state restoration could be optimized by specifying which topics the State Widget would need to publish. In this way, only those topics to which widgets in the particular mashup view have subscribed to could be published to restore the state of the particular mashup view.

Finally, the Transformation Widget used in the enterprise mashup framework could be extended with more sophisticated data aggregation functionality. This improvement is also mentioned as future work for the Transformation Widget proposed by Villido [30]. The enterprise mashup framework would benefit from this extension as situational mashups aim at solving specific tasks. This means that task-specific information can be shared and modified by different mashup user roles. To correctly combine the data modified by various users, additional data aggregation rules would need to be defined.

Rollipõhine dekompositsioon vidinail baseeruvate oleku jagamise, salvestamise ja taastamise toega ärirakenduste mitme eksemplariga käivitamiseks

Magistritöö (30 EAP)

Liisi Haav

Resümee

Veebimaastikul suurt populaarsust kogunud tavatarbijatele suunatud vidinapõhised veebirakendused on loonud soodsa pinnase üldotstarbeliste *mashup*'ite loomise raamistike ning tööriistade tekkeks. Need tööriistad on eelkõige suunatud tava-Interneti kasutajatele, et luua lihtsaid *mashup*-tüüpi rakendusi. Samal ajal oleks vidinapõhistest veebirakendustest kasu ka ärirakendustena. Peamiseks takistuseks ärirakenduste loomisel veebipõhiste rakendustena on keerulisest äriloogikast tulenevad keerukad nõuded ning protsessid. Antud magistritöö uurib, kuidas teostada veebividinatel põhinevate *mashup*-tüüpi ärirakenduste arendamist nii, et säiluks *mashup*'ite loomisega seotud peamised eelised, lihtsus ja kiirus.

Käesolev magistritöö pakub välja laienduse olemasolevale *mashup*-tüüpi raamistikule, et toetada *mashup*'i dekompositsiooni rollipõhisteks vaadeteks. Selleks jagatakse *mashup* väiksemateks vidinate komplektideks, tagades igale kasutajarollile komplekt just temale vajaminevatest vidinatest. Kuigi igal kasutajarollil võib olla erinev vaade kogu ärirakendusest, tagab käesolevas magistritöös pakutud lahendus suhtluse nende erinevate vaadete vahel. See on vajalik tagamaks *mashup*'i eksemplari ühtsust kõikide *mashup*'i vaadete vahel, olenemata sellest, millistest vidinatest antud kasutaja vaade koosneb.

Lisaks pakub käesolev magistritöö välja lahenduse toetamaks mitut eksemplari samast vidinapõhisest ärirakendusest ning toetamaks ärirakenduse oleku salvestamist ning taastamist. Kuna ärirakendused on suunatud lahendamaks kasutajate igapäevaseid ülesandeid, on vajalik, et kasutaja saaks valida olemasolevate *mashup'i* eksemplaride hulgast või alustada uut eksemplari. Lisaks on vajalik, et kasutaja saaks igal ajahetkel rakenduse kasutamise lõpetada selliselt, et hiljem rakenduse kasutamist jätkates oleks tagatud sama rakenduse olek, millest kasutamine katkestati. Väljapakutud lahenduse toimimist testitakse näidisrakendusega, mis realiseerib krediidihalduse protsessi.

Bibliography

[1] T. O'Reilly. What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. In *International Journal of Digital Economics*, Number 65, pages 17-37, 2007.

[2] A. Taivalsaari and T. Mikkonen. Mashups and Modularity: Towards Secure and Reusable Web Applications. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference*, pages 25-33, 2008.

[3] V. Hoyer and M. Fischer. Market Overview of Enterprise Mashup Tools. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 708-721, 2008.

[4] R. Gurram, B. Mo, and R. Gueldemeister. A Web Based Mashup Platform for Enterprise 2.0. In *Proceedings of WISE 2008 International Workshops on Web Information Systems Engineering*, pages 144-151, 2008.

[5] F. Daniel and M. Matera. Turning Web Applications into Mashup Components: Issues, Models, and Solutions. In *Proceedings of 9th International Conference on Web Engineerin*g, pages 45-60, 2009.

[6] OpenAjax Hub 2.0 Specification. `http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification`. Cited: May 15, 2013.

[7] V. Dheap, C.M. Gladd, A.C. Lindsay, and D.P. Sink. Distributed Multi-User Mashup Session. Patent 20110161833, June 30, 2011.

[8] R. Tuchinda, P. Szekely, and C.A. Knoblock. Building Mashups by Example. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, pages 139-148, 2008.

[9] J. Lin, J. Wong, J. Nichols, A. Cypher, and T.A. Lau. End-User Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*, pages 97-106, 2009.

[10] G. Wang, S. Yang, and Y. Han. Mashroom: End-User Mashup Programming Using Nested Tables. In *Proceedings of the 18th International Conference on World Wide Web*, pages 861-870, 2009.

[11] J. Wong and J.I. Hong. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1435-1444, 2007.

[12] R. Ennals and M. Garofalakis. MashMaker: Mashups for the Masses. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1116-1118, 2007.

[13] I. Yamada, W. Yamaki, H. Nakajima, and Y. Takefuji. Ousia Weaver: A Tool for Creating and Publishing Mashups as Impressive Web Pages. In *MEM 2010: 3rd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web, in WWW 2010: Proceedings of the 18th International World Wide Web Conference*, 2010.

[14] M. Albinola, L. Baresi, M. Carcano, and S. Guinea. Mashlight: a Lightweight Mashup Framework for Everyone. In *Proceedings of WWW 2009*, 2009.

[15] D.E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: Data Mashups for Intranet Applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1171-1182, 2008.

[16] A. Tammik. Interactive Data Sharing Mechanism for Widget-Based Microsites. MSc thesis, Institute of Computer Science, University of Tartu, Tartu, Estonia, 2011.

[17] M. Heinrich, F.J. Grüneberger, T. Springer, and M. Gaedke. Reusable Awareness Widgets for Collaborative Web Applications – A Non-Invasive Approach. In *Proceedings of 12th International Conference on Web Engineering*, pages 1-15, 2012.

[18] OpenAjax Alliance Enterprise Mashup Markup Language Documentation. `http://www.openmashup.org/omadocs/v1.0`, Cited: May 16, 2013.

[19] E.M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A Domain-Specific Language for Web APIs and Services Mashups. In *Proceedings of the 5th International Conference on Service-Oriented Computing*, pages 13-26, 2007.

[20] K. Prutsachainimmit, P. Chaisatien, and T. Tokuda. A Mashup Construction Approach for Cooperation of Mobile Devices. In *ICWE 2012 International Workshops: MDWE, ComposableWeb, WeRE, QWE, and Doctoral Consortium*, pages 97-108, 2012.

[21] S. Pietschmann. A Model-Driven Development Process and Runtime Platform for Adaptive Composite Web Applications. In *International Journal on Advances in Internet Technology*, Volume 2, Number 4, pages 277-288, 2009.

[22] S. Pietschmann, V. Tietz, J. Reimann, C. Liebing, M. Pohle, and K. Meißner. A Metamodel for Context-Aware Component-Based Mashup Applications. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, pages 413-420, 2010.

[23] M. Imran, S. Soi, F. Kling, F. Daniel, F. Casati, and M. Marchese. On the Systematic Development of Domain-Specific Mashup Tools for End Users. In *Proceedings of the 12th International Conference on Web Engineering*, pages 291-298, 2012.

[24] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A Framework for Rapid Integration of Presentation Components. In *Proceedings of the 16th International Conference on World Wide Web*, pages 923-932, 2007.

[25] F. Daniel, F. Casati, B. Benatallah, and M-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *Proceedings of the 28th International Conference on Conceptual Modeling*, pages 428-443, 2009.

[26] S. Pietschmann, M. Voigt, and K. Meißner. Rich Communication Patterns for Mashups. In *Proceedings of the 12th International Conference on Web Engineering*, pages 315-322, 2012.

[27] S. Wilson, F. Daniel, U. Jugel, and S. Soi. Orchestrated User Interface Mashups Using W3C Widgets. In *Proceedings of the 11th International Conference on Current Trends in Web Engineering*, pages 49–61, 2011.

[28] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, Volume 12, Number 5, pages 44-52, 2008.

[29] H. Mäesalu. Automated Rule-Based Selection and Instantiation of Layout Templates for Widget-Based Microsites. MSc thesis, Institute of Computer Science, University of Tartu, Tartu, Estonia, 2013.

[30] R. Villido. Semantic Integration Platform for Web Widgets' Communication. MSc thesis, Institute of Computer Science, University of Tartu, Tartu, Estonia, 2010.

[31] P.B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, Volume 12, Issue 6, pages 42-50, 1995.

[32] OpenAjax Alliance. `http://www.openajax.org`. Cited: May 15, 2013.

[33] OpenAjax Metadata 1.0 Specification. `http://www.openajax.org/member/wiki/OpenAjax_Metadata_Specification`. Cited: May 15, 2013.

[34] NodeJS. `http://www.nodejs.org`. Cited: May 15, 2013.

[35] NPM – Node Packaged Modules. `https://www.npmjs.org`. Cited: May 15, 2013.

[36] NPM. `http://en.wikipedia.org/wiki/Npm_(software)`. Cited: May 15, 2013.

[37] Socket.io. `http://socket.io`. Cited: May 15, 2013.

[38] Forever. `https://npmjs.org/package/forever`. Cited: May 15, 2013.

[39] K. Kirsimäe. Automated OpenAjax Hub Widget Generation for Deep Web Surfacing. MSc thesis, Institute of Computer Science, University of Tartu, Tartu, Estonia, 2011.

[40] CodeIgniter. http://ellislab.com/codeigniter. Cited: May 15, 2013.

Appendix

Source code

The source code of the provided solution and the test application, which were discussed in this thesis, can be downloaded from Github:

`https://github.com/lhaav/enterprise-mashup-framework`

Non-exclusive licence to reproduce thesis and make thesis public

I, Liisi Haav
(*author's name*)
(date of birth: 22.04.1986),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Role-Based Enterprise Mashups with State Sharing, Preservation and Restoration Support for Multi-Instance Executions,
(title of thesis)

supervised by Peep Küngas,
(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **20.05.2013**