Multi-device UI Development for Task-Continuous Cross-Channel Web Applications

Enes Yigitbas[1(✉)], Thomas Kern[2], Patrick Urban[2], and Stefan Sauer[1]

[1] S-lab - Software Quality Lab, Paderborn University, Zukunftsmeile 1, 33102 Paderborn, Germany
{eyigitbas,sauer}@s-lab.upb.de
[2] Wincor Nixdorf International GmbH, Heinz-Nixdorf-Ring 1, 33106 Paderborn, Germany
{thomas.kern,patrick.urban}@wincor-nixdorf.com

Abstract. The growing number of various types of web-enabled smart devices presents a special challenge for retail banks. In the world of Omni-Channel-Banking, customers demand a flexible and easy usage for carrying out their banking activities. Establishing such an Omni-Channel-Banking experience is a challenging task that requires support for the development of heterogeneous user interfaces (UIs) allowing flexible access to different channels (e.g. PC, Smartphone, ATM) and a seamless hand-over between these channels to allow task-continuity for the customer. Therefore, we present a model-based solution architecture for the development of multi-device UIs. Our solution architecture minimizes recurrent UI development efforts for different channels and enables data synchronization between them. To show the feasibility of our approach, we present an industrial case study, where we implement a cross-channel banking web-application that enables a modern customer experience.

Keywords: Model-based development · Multi-device UI development · Liquid software development · Cross-channel web applications · Self-service systems

1 Introduction

The growing number of various types of web-enabled smart devices (e.g. smartphones, smartwatches, tablets, etc.) presents a special challenge for retail banks. Customers demand a flexible and easy usage for carrying out their banking activities. While customers accessed banking services solely via isolated channels

This work is based on "KoMoS", a project of the "it's OWL" Leading-Edge Cluster, partially funded by the German Federal Ministry of Education and Research (BMBF).

S. Casteleyn et al. (Eds.): ICWE 2016 Workshops, LNCS 9881, pp. 114–127, 2016.
DOI: 10.1007/978-3-319-46963-8_10

(through banking personnel or ATM) in the past, using different channels during a transaction is nowadays increasingly gaining popularity. Depending on the situation, customers are able to access their banking services where, when and how it suits them best. In the world of Omni-Channel-Banking, customers are in control of the channels they wish to use, experiencing a self-determined "Omni-Channel-Journey". For example, if the customers pursue an "Omni-Channel-Journey" for a payment cashout process, they can begin an interaction using one channel (prepare cashout at desktop at home), modify the transaction on their way on a mobile channel, and finalize it at the automatic teller machine (ATM) (see Fig. 1). Thus, Omni-Channel-Banking brings the industry closer to the promise of true contextual banking in which financial services become seamlessly embedded into the lives of individual and business customers.

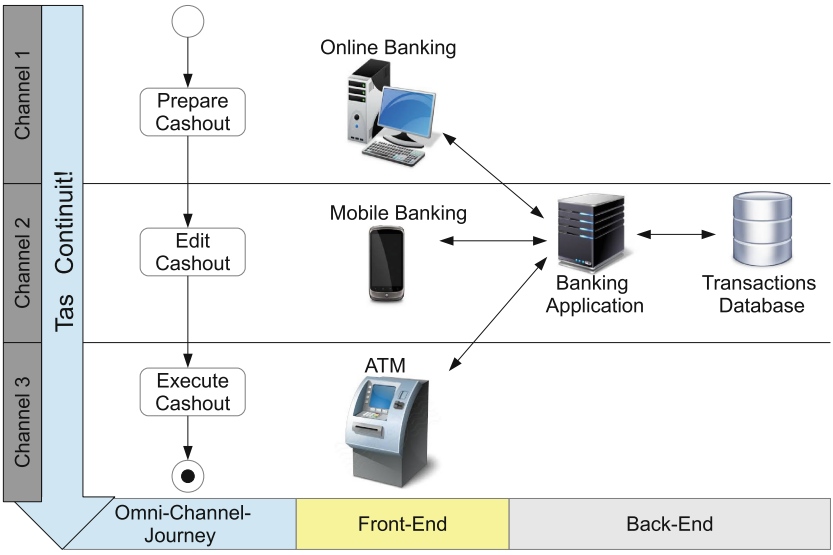

Fig. 1. Example scenario: omni-channel-journey with task-continuity

Table 1. Multi-channel-banking vs. omni-channel-banking

Multi-channel-banking	Omni-channel-banking
Fixed channel usage	Flexible channel usage
Separation of channels	Integration of channels
Data redundancy in channels	Data synchronization between channels
Little or no channel switch	Continuous channel switch

The advancement from Multi-Channel- to Omni-Channel-Banking (compare Table 1) is a difficult task for developers of such systems. Developers are facing the following challenges:

- **C1:** Support for heterogeneous user interfaces (UIs) allowing access for different channels (e.g. PC, Smartphone, ATM).
- **C2:** Support for a flexible channel usage depending on the context.
- **C3:** Support for a seamless handover between channels allowing task-continuity. When the user moves from one device to another, the user is able to seamlessly continue her task.

Tackling these issues by combining and integrating heterogeneous channels on a cross-platform software infrastructure imposes huge efforts for development and maintenance. Therefore, we present a model-based solution architecture that minimizes recurrent UI development efforts for different channels and enables data synchronization between these channels according to the paradigm of Liquid Software Development. To show the feasibility of our approach, we present an industrial case study, where we implement a cross-platform software stack for Omni-Channel cash transactions employing the latest HTML5-based technologies and open source frameworks that enable a modern customer experience. The implementation of our web-based cross-platform software infrastructure demonstrates how liquid software development can be applied to the banking domain.

The paper is structured as follows: First, we describe some background information and related work in the area of multi-device and cross-channel UI development. Then, we present our model-based solution architecture for the development of multi-device UIs supporting task-continuity. After that, we present the implementation of our approach based on a case study from the banking domain. Finally, we conclude with a summary and an outlook for future research work.

2 Background and Related Work

In recent years, a number of approaches have addressed the problem of interacting with liquid software applications distributed across various types of smart devices. Our work is inspired by and based on existing approaches from the area of distributed user interfaces (DUIs). In this section, we especially review prior work that explores the development of multi-device and cross-channel user interfaces (UIs) supporting task-continuity.

2.1 Multi-device UI Development

The development of multi-device UIs has been subject of extensive research [1] where different approaches were proposed to support efficient development of UIs for different target platforms. On the one hand, model-based UI development approaches were proposed which aim to create multi-device UIs based on the transformation of abstract user interface models to final user interfaces.

Two widely studied approaches are MARIA [2] and IFML[1] that support the abstract modeling of user interfaces and their transformation to multi-device UIs including web interfaces. In [3] we present a specialized approach for model-based development of heterogeneous UIs for different target platforms including self-service systems. On the other hand there are also existing approaches like Damask [4] and Gummy [5] following the WYSIWYG paradigm. While Damask is a prototyping tool for creating sketches of multi-device web interfaces, Gummy is a design environment for graphical UIs that allows designers to create interfaces for multiple devices using visual tools to automatically generate and maintain a platform-independent description of the UI. While above mentioned approaches support the development of multi-device UIs regarding specification and generation of UIs for different target platforms, they do not cover mechanisms to support channel switches and data synchronization between different target platforms at runtime.

2.2 Cross-Channel UI Development

Previous work by the research community has covered concepts and techniques to dynamically support the distribution of UIs by supporting task-continuity for the end-users. One of the concepts is called *UI migration*, which follows the idea of transferring a UI or parts of it from a source to a target device while enabling task-continuity through carrying the UI's state across devices. In [8], we present a model-based framework for the migration and adaptation of user interfaces across different devices. In [6], the authors present an agent-based solution to support migration of interactive applications among various devices, including digital TVs and mobile devices, allowing users to freely move around at home and outdoor. The aim is to provide users with a seamless and supportive environment for ubiquitous access in multi-device contexts of use. In the case of web applications, most solutions rely on HTML proxy-based techniques to dynamically push and pull UIs [7]. An extension of this concept is presented in [9], where the authors propose XDStudio to support interactive development of cross-device UIs. In addition, there is also existing work on the specification support for cross-device applications. In [10] for example, the authors present their framework Panelrama which is a web-based framework for the construction of applications using DUIs. In a similar work [11], the authors present Conductor, which is a prototype framework serving as an example for the construction of cross-device applications.

Leaning on the existing concepts of cross-channel UI development, we present a model-based solution architecture for multi-device UIs that supports task-continuity. By extending existing architectural cross-device application frameworks to the banking domain, we aim to support an omni-channel banking experience for the customers.

[1] http://www.ifml.org.

3 Solution Architecture

In this section, we present a model-based solution architecture for multi-device UI development in order to tackle the motivated challenges C1, C2 and C3. Figure 2 shows our solution architecture which is divided into three main steps: *Modeling*, *Transformation*, and *Execution*.

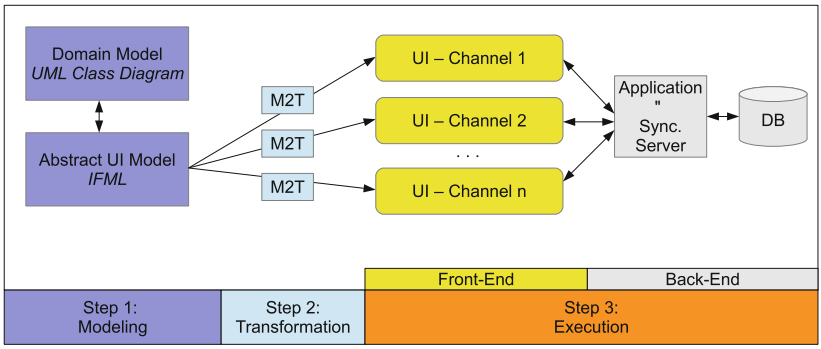

Fig. 2. Model-based solution architecture for multi-device UIs supporting task-continuity

For supporting the development of various UIs allowing access to the different channels and minimizing recurrent development efforts in establishing the needed *Front-Ends* (C1), we have a modeling step in our solution architecture. In the modeling step, a *Domain Model* described as UML class diagram and an *Abstract UI Model* based on the *Interaction Flow Modeling language (IFML)*, serve as specification of the data entities as well as structure, content and navigation needed to characterize the UI in an abstract manner. Based on the IFML *Abstract UI model*, which is referencing the *Domain Model*, a transformation is defined to generate heterogeneous UIs for the different *UI-Channels (1..n)*. Therefore, in the transformation step, several *model-to-text transformation (M2T)* templates are defined that transfer the *Abstract UI models* into the final UIs, running on different target platforms in order to support access to the different channels (Front-End). By generating different UI views and supporting different *UI-Channels*, the users are able to flexibly select the channel of their choice depending on the context (C2). In order to support a seamless handover between channels and allowing task-continuity for the user (C3), our solution architecture includes an *Application and Synchronization Server* in the *Back-End*, which is responsible for storing and sharing of data (e.g. UI state or user preferences). The UI state, including entered input data by the users, is stored and restored, allowing the user to move across channels while seamlessly continuing her task.

4 Instantiation of Development Process

In this section, the instantiation of the development process according to the previously described solution architecture is presented in more detail. To show the feasibility of our approach, we first present the setting of an industrial case-study dealing with the implementation of a cross-platform software infrastructure for Omni-Channel cash transactions employing web-based technologies. After that, we present the realization of the solution architecture by describing the implementation of the different steps.

4.1 Setting of the Case Study

Our "Omni-Channel-Banking" case-study supports a variety of different channels to access banking services. Figure 3 shows its overall architecture.

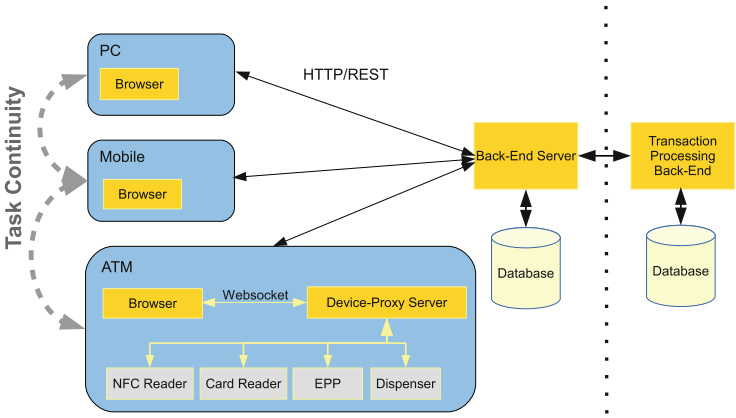

Fig. 3. Case-study application architecture

On each device - *PC*, *Mobile*, *ATM* - the client application is running as a single-page web application inside a browser. The application communicates with a *Back-End Server*, which is responsible for

- serving an application to the browser, adapted to a particular target device,
- serving application specific data to the client via HTTP/REST,
- managing application state and user preferences,
- requesting information from a *Transaction Processing Back-End* and serving it to the client,
- sending financial transactions to the *Transaction Processing Back-End* for execution.

The data format for all data exchanged through HTTP/REST requests is *JavaScript Object Notation* (JSON).

The *Transaction Processing Back-End* is not part of our application, but represents an existing infrastructure for processing financial transactions. The *Back-End Server* communicates with this transaction processing system. The communication protocol between the *Transaction Processing Back-End* and our sample application's *Back-End Server* depends on an existing infrastructure. Thus, the *Back-End Server* needs to provide a custom adapter for interfacing with this system.

In our case study, PC and mobile applications are identical concerning their functionality. The main difference comes from adaptation to different screen sizes and operation through a touch screen. This also includes spreading of functionality on the mobile device over multiple dialogs, compared to the PC application. In contrast to PC and mobile clients, the application architecture of the ATM client is significantly different. This is due to the need for supporting a whole variety of ATM specific hardware devices, like *NFC Reader*, *Card Reader*, *Encrypting Pin Pad (EPP)*, *Cash Dispenser*, etc. For interoperability reasons, ATM vendors are using a common software stack called XFS, which is layered on top of device specific drivers. XFS stands for *Extensions for Financial Services* and is standardized by CEN, the *European Commitee for Standardization*. Since a browser itself can not directly access the XFS-API, we delegate device control to a *Device-Proxy Server* running directly on the ATM.

4.2 Modeling and Transformation

For realizing the modeling and transformation step of the solution architecture, we have implemented a model-based UI development (MBUID) process which is depicted in Fig. 4. This MBUID process supports the modeling and transformation of the UIs, which are the view parts of the single-page application rendered as HTML5 by the browser. Using the open source IFML Editor Eclipse plugin[2], developers are able to specify the domain and abstract UI model. For transforming these models into final web UI views, we implemented an Xtend[3] plugin that maps the IFML model elements to specific HTML5 elements. The Xtend plugin includes different Xtend templates to transfer the IFML source model into web UIs supporting manifold devices.

During the transformation process, the application's view is built upon basic components with a custom look &feel, like buttons, text input fields, dropdown lists, tables, etc. As a basis for these components, we did not use AngularJS directives, but implemented components based on the *HTML5 Web Components*[4] specification promoted by Google as W3C standard.

[2] http://ifml.github.io.
[3] http://www.eclipse.org/xtend.
[4] https://www.w3.org/TR/components-intro.

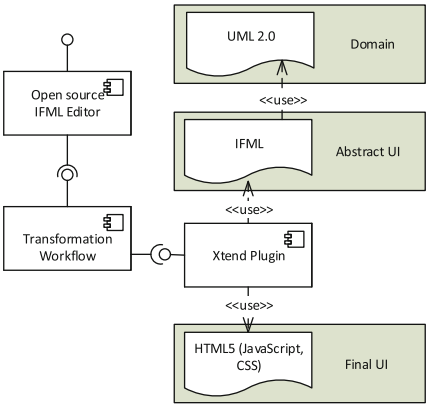

Fig. 4. Implemented model-based UI development process

Our custom components are sensitive to the application environment they are being used in (desktop, mobile, ATM) and adapt themselves accordingly. On mobile devices, for example, buttons are larger and more suitable for touch operation than on desktop devices.

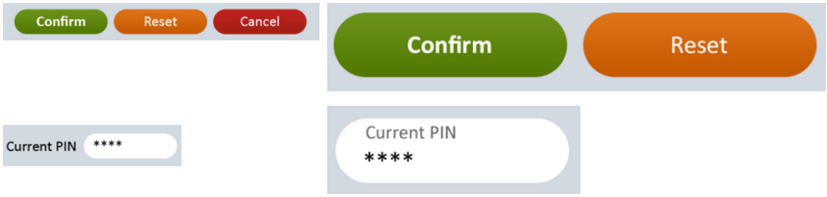

Fig. 5. Buttons and text fields for desktop and mobile

Figure 5 shows buttons and text input fields. Their desktop representation is depicted on the left side, their mobile appearance on the right side of the picture.

During the transformation process for all device classes, a button is created the same way:

```
<komos-button colorscheme="cs1" ng-click="confirm()">
  Confirm
<komos-button>
```

The following example shows how to create a text input field with a label by mapping an IFML simple field element to the following code snippet:

```
<komos-textfield label="Current PIN" ng-model="model.currentPin">
  </komos-textfield>
```

In order to provide a unified layout management for our application, our model-to-text (M2T) transformation process implements a custom layout manager. It provides an easy to use grid layout system, based on row and column

elements realized as AngularJS directives. Under the hood, it uses HTML5 Flexbox. The following listing shows the generated code snippet to create the dialog shown in Fig. 6.

```
<komos-container>
  <komos-row>
    <komos-column span-3>
      <komos-label>Username</komos-label>
    </komos-column>
    <komos-column span-8>
      <komos-textfield name="username" ng-model="model.username">
        </komos-textfield>
    </komos-column>
  </komos-row>

  <komos-row>
    <komos-column span-3>
      <komos-label>Password</komos-label>
    </komos-column>
    <komos-column span-8>
      <komos-textfield name="password" ng-model="model.password">
        </komos-textfield>
    </komos-column>
  </komos-row>

  <komos-row>
    <komos-column offset-3 span-8>
      <komos-button colorscheme="cs1" ng-click="login(form)">
        Login
      </komos-button>
      <komos-button colorscheme="cs5" ui-sref="public.signup">
        Register
      </komos-button>
      <komos-button colorscheme="cs3" ng-click="reset(form)">
        Cancel
      </komos-button>
    </komos-column>
  </komos-row>
</komos-container>
```

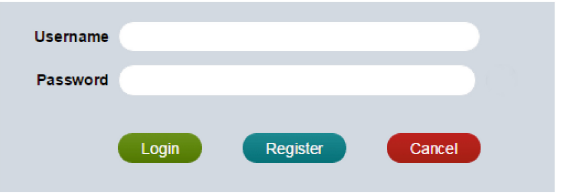

Fig. 6. Login dialog

4.3 Execution (Front-End)

While the previous subsection presented our MBUID process to support the modeling and generation of view aspects of the *Front-End*, this section deals with the execution step. In this context, we especially present the controller part of the *Front-End*, which is responsible for application logic and communication with the *Back-End Server*. In conjunction with this topic, we also present the aspect of channel handover and task-continuity.

As shown in Fig. 7, the *Front-End* consists of a HTML5/JavaScript single-page application running in a web browser. It exchanges JSON messages with the *Back-End Server* through HTTP/REST.

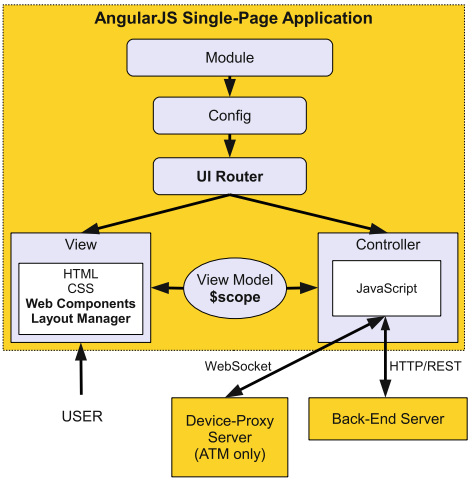

Fig. 7. Front-end architecture

The browser application's main building blocks are:

- AngularJS[5]: Google's open-source web application framework for developing single-page applications in JavaScript
- UI Router[6]: flexible client-side routing with nested views in AngularJS
- Web Components: UI components with custom look & feel
- Layout Manager: custom layout manager

AngularJS supports the model-view-controller (MVC) design pattern by decoupling the application's presentation layer, which is defined through HTML5 (see previous subsection), from the model and application logic by two-way data-binding through a `$scope` object. In addition, AngularJS provides a variety of other services, including modularization and definition of custom directives.

UI Router is the client-side routing component of AngularJS and the central key component to implement task continuity. The developer assigns a particular application state, identified by a name (`protected.main`), with a view (`main.html`) and a controller (`MainCtrl`):

[5] https://angularjs.org.
[6] https://github.com/angular-ui/ui-router/wiki.

```
angular.module('komosApp').config(function ($stateProvider) {
  $stateProvider
    .state('protected.main', {
      url: '/',
      templateUrl: 'protected/main/main.html',
      controller: 'MainCtrl',
      authenticate: true
  });
});
```

In order to support task continuity and transfer application state between devices, the current state name and its associated context are saved to the *Back-End Server*.

Inside a view controller and prior to saving a state, all context information necessary for recovery is added to a state-context object. This includes the UI's view-model, as well as any other necessary information associated with the current state.

```
var context = {
  // the view model:
  viewModel: $scope.model,
  // state specific arbitrary properties:
  param1: someValue,
  data: someData
};

PersistStateService.save('protected.main', context, function (err, data) {
  if (err) model.errors.message = err.data.message;
});
```

We implemented an AngularJS service named `PersistStateService`, which converts the object `context` to JSON and sends it to the *Back-End Server*, where it is stored under the name of the state, e.g. `protected.main`. To invoke a previously saved state, the application just needs to retrieve the current state name and invoke it:

```
$rootScope.$state.go('protected.main');
```

On instantiation of the AngularJS controller associated with this state, the controller calls the service's restore method to retrieve the previously stored information:

```
PersistStateService.restore('protected.main', function (err, context) {
  if (err) {
    model.errors.message = err.data.message;
  } else {
    // context now contains the previously saved information
    $scope.model = context.viewModel; // this updates the UI!!
    someValue = context.param1;
    someData = context.data;
  }
});
```

Both, saving and retrieving context data for a state happens within the same controller. Each controller knows exactly which data needs to be saved in order to be able to restore itself. This information is hidden from other parts of the application. The only knowledge necessary from the outside is the name of the state, `protected.main` in our example.

Because of AngularJS' two-way data-binding, assigning the view-model to `$scope.model` immediately updates the view.

4.4 Execution (Back-End)

The application's *Back-End* is implemented in JavaScript (see Fig. 8) and uses Node.js[7] as its runtime environment. It is built upon Google's V8 JavaScript engine also used by Google Chrome and provides a high-performance runtime environment for non-blocking and event-driven programming.

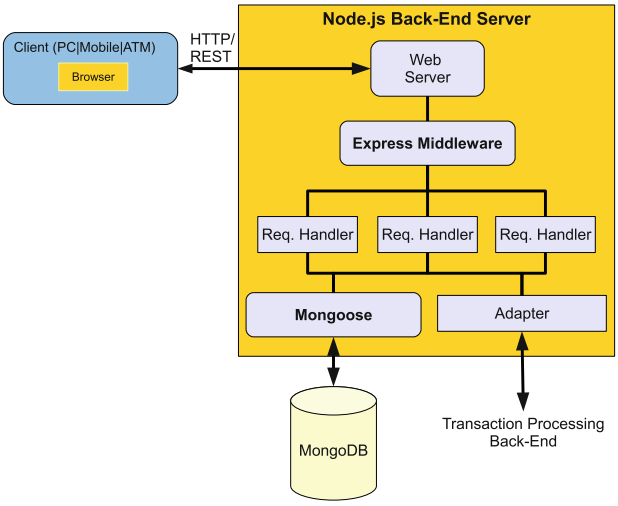

Fig. 8. Back-end architecture

ExpressJS[8], which is a middleware for Node.js, provides components for processing of requests and routing. An application sets up request handlers, which are automatically invoked when a client request arrives. Within a request handler, the request is processed, a response is prepared and returned. Request handlers communicate with the database or *Transaction Processing Back-End*.

The document database *MongoDB*[9] belongs into the category of NoSQL ("Not Only SQL") databases. In this context, a "document" consists of a user-defined data structure of key-value pairs, which is associated with a key. Documents can also contain other documents. The schema of a database is dynamic and can be modified at runtime. To access the database in an object-oriented fashion, we use an *Object Document Mapper* called *Mongoose* on top of MongoDB's Node.js driver.

The instantiation of our solution architecture and interaction of all described technologies resulted in the demonstrator which is shown in Fig. 9. Our demonstrator shows the implemented cross-channel web-application that supports different channels (Desktop, Tablet, and ATM) for a cash payout process enabling task-continuity for the customers.

[7] https://nodejs.org.
[8] https://expressjs.com.
[9] https://www.mongodb.org.

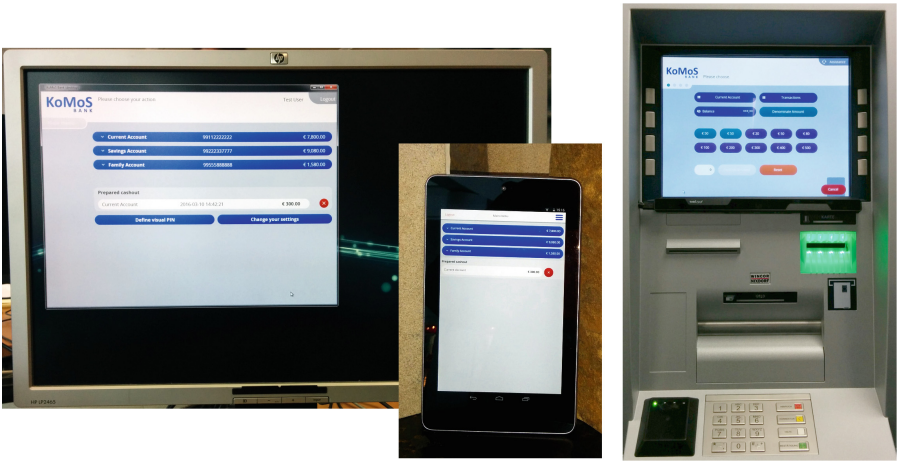

Fig. 9. Cross-channel banking web application supporting task-continuity

5 Conclusion and Outlook

This paper presents a model-based solution architecture that supports the efficient development of UIs for different channels (e.g. PC, Smartphone, ATM) and enables data synchronization between them. This solution offers end-users a flexible and easy usage for accessing their services through variable channels and a seamless hand-over between channels allowing task-continuity. We showed the feasibility of our approach based on a cross-channel banking web-application that was implemented according to our solution architecture. The implementation of our case study includes a cross-platform software infrastructure for Omni-Channel cash transactions employing the latest web technologies and open source frameworks. In ongoing work we are developing and extending the model-to-text transformation process in order to support the generation of dynamic UI aspects (e.g. input validation, controller artifacts, etc.). Our future work will focus on studies with web designers and developers to further evaluate the efficiency and effectiveness of our approach.

References

1. Paternò, F., Santoro, C.: A logical framework for multi-device user interfaces. In: Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2012), pp. 45–50. ACM, New York (2012)
2. Paternò, F., Santoro, C., Spano, L.D.: MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput.-Hum. Interact (2009)
3. Yigitbas, E., Fischer, H., Kern, T., Paelke, V.: Model-based development of adaptive UIs for multi-channel self-service systems. In: Sauer, S., Bogdan, C., Forbrig, P., Bernhaupt, R., Winckler, M. (eds.) HCSE 2014. LNCS, vol. 8742, pp. 267–274. Springer, Heidelberg (2014)

4. Lin, J., Landay, J.A.: Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2008), pp. 1313–1322. ACM, New York (2008)
5. Meskens, J., Vermeulen, J., Luyten, K., Coninx, K.: Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In: Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2008), pp. 233–240. ACM, New York (2008)
6. Paternò, F., Santoro, C., Scorcia, A.: Ambient intelligence for supporting task continuity across multiple devices and implementation languages. Comput. J. **53**(8), 1210–1228 (2010)
7. Ghiani, G., Paternò, F., Santoro, C.: Push and pull of web user interfaces in multi-device environments. In: Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI 2012), pp. 10–17. ACM, New York (2012)
8. Yigitbas, E., Sauer, S., Engels, G.: A model-based framework for multi-adaptive migratory user interfaces. In: Kurosu, M. (ed.) Human-Computer Interaction. LNCS, vol. 9170, pp. 563–572. Springer, Heidelberg (2015)
9. Nebeling, M., Mintsi, T., Husmann, M., Norrie, M.: Interactive development of cross-device user interfaces. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2014) (2014)
10. Yang, J., Wigdor, D.: Panelrama: enabling easy specification of cross-device web applications. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2014), pp. 2783–2792. ACM, New York (2014)
11. Hamilton, P., Wigdor, D.J.: Conductor: enabling and understanding cross-device interaction. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2014), pp. 2773–2782. ACM, New York (2014)