MMIR framework

Multimodal Mobile Interaction and Rendering Framework

Ehsan Gholamsaghaee
30.03.2012
Aaron Ruß
11.03.2013

Table of Contents

Introduction

Mobile access to information is an integral part of our digital life. Smartphones will replace conventional mobile phones within the next few years. As a result, people will increasingly use the mobile internet. As a drawback, portable devices usually bring limitations for user interactions, for example due to small display size and missing/small keyboards. In this regard, with the advance of speech recognition and speech synthesis technologies, their complementary use becomes attractive for mobile devices in order to implement real multimodal user interaction.

This document describes the Multimodal Mobile Interaction and Rendering framework (MMIR framework) that is developed in context of the MultiModal Interaction Group (MMIG) at the German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI GmbH).

The MMIR framework aims to provide the basic components of multimodal (mobile) interaction systems such as multimodal input manager, interaction manager, and data manager so that the developers can focus on unconventional aspects of theirs project rather than repetitive tasks which are common to all projects. The adjustment and extension of such a framework should be as easy as possible. That is one of our main goals for the design and development of MMIR. Although the focus for the current version of MMIR is targeted at mobile devices such as smartphones and tablet PCs, you can use it for regular PCs, too.

Mobile applications or apps are compact software programs that perform specific tasks for the mobile user. There are basically two types of mobile apps:

Native Apps:

Native apps are the products of apps developed using the device's own development kit (SDK). The native app must be installed on the device; they are either delivered pre-installed on the phone, or they can be downloaded from Web sites. Native apps are written specifically for a type of handset so they can take more advantage of a phone's functions.

Pros:

- With the ability to be completely offline and the availability of local databases, native apps can easily and securely perform their task.
- A native app has the ability to interact with all the available device functions through the SDK. Whether the app needs to transmit instructions to an external device, or simply capturing an image, the native app has many options from which to choose.

Cons:

- Native apps are more difficult to develop and require specific platform knowledge and programming language.
- Lack of cross platform compatibility, which requires the software vendors to choose a preferred platform or have multiple teams developing for multiple platforms.

Web Apps:

The Web app resides on one or multiple servers and is accessed via the Internet. It performs specified tasks – potentially the same ones as a native application – for the mobile user, usually by downloading part of the application on the device for local processing each time it is used. The software is written as Web pages in HTML and CSS, with interactive parts in JavaScript. This means that the same application code can be used by almost all devices that can surf the Web (regardless of the brand of phone).

Pros:

- A web based app relies on server side processing for rendering content. It is a web page that has some additional scripting done to it to ensure that the look and feel is somewhat similar to the look of native apps.
- Server side processing means better performance in many cases, since the end user does not have to wait for the information to transferred and manipulated on the device itself.
- From a development standpoint, web solutions tend to be easier to develop than native mobile device apps.
- Web based apps are also cross platform and can run on (almost) any browser regardless of the device you are using.

Cons:

- Loss of hardware integration: In a web based model you will not have access to most of the hardware capabilities such as the device's cam or capture data from an external device, such as geo coordination from GPS sensor.
- Loss of some features, such as multi touch .There are web sites that have created controls similar to iOS controls, but they still lack the typical experience that accompanies the native ones.
- Lack of strong support for offline functionality when the user do not have access to WiFi or internet.

Building mobile web apps is a good way for creating portable solutions. It is an approach for mobile app development that when done right, will have you rewriting a lot less code to target the variety of devices that exist in the marketplace. A mobile web app is an application that is built with the core client web technologies if HTML, CSS, and JavaScript, and is specifically designed for mobile devices.

Hybrid Approach

An alternative approach is developing cross-platform mobile applications using web technologies as well as native code. Such applications are able to interact with mobile device hardware, such as Accelerometer or GPS (native part), but they can implement all other functionalities web-based. For more details see chapter Why Cordova (p. 4).

Why Cordova

Mobile development is a mess. Building applications for the major platforms – iPhone, Android, Windows Mobile, etc. – requires different frameworks and languages. An alternative approach is developing cross-platform mobile applications with HTML, CSS, and JavaScript using Apache Cordova[1]. Cordova is a HTML5 app platform that allows developers to create native applications with Web technologies and get access to APIs and app stores. Applications built with Cordova are not just like normal mobile web sites. Cordova applications are able to interact with mobile device hardware, such as Accelerometer or GPS, in a way that is (currently) not available for normal web applications. The resulting applications are hybrid, which means that they are neither truly native (all layout rendering is done via the web view instead of the platform's native UI framework) nor purely web based (they are not just web apps and have access to part of the device APIs).

---

[1] The Apache Cordova project was started off with the donation of code from PhoneGap. Now PhoneGap is a distribution using the Open Source code base of Cordova; for more see e.g. http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/.

Figure 1: Comparison of supported features by Cordova on different platforms[2]

	iOS iPhone / iPhone 3G	iOS iPhone 3GS and newer	Android	OS 4.6-4.7	OS 5.x	OS 6.0+	WebOS	WP7	Symbian	Bada
ACCELEROMETER	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
CAMERA	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
COMPASS	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓
CONTACTS	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
FILE	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗
GEOLOCATION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MEDIA	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗
NETWORK	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (ALERT)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (SOUND)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NOTIFICATION (VIBRATION)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
STORAGE	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗

jQuery Mobile

jQuery Mobile (jQM) is an open source, touch-optimized web framework developed by the jQuery team. The development focuses on creating a framework compatible with a wide variety of smartphones and tablet computers, made necessary by the growing but heterogeneous tablet and smartphone market. jQM divides the world up into pages, which are essentially just screens. Each page is divided into three areas; the header, the main content, and the footer.

With jQM developers can create applications that will run on a number of different platforms, not only the major ones (as iOS and Android). jQM is built around the principle of progressive enhancement, meaning any jQM application will work in many browsers, even those that do not support JavaScript. Another important aspect is the small file size of the jQM library, specifically designed for mobile usage.

For more information on jQM visite http://jquerymobile.com/.

What is MMIR?

MMIR is a web-based development framework, especially designed to make the process of developing multimodal (mobile) interaction systems easier. This section explains the background of the MMIR framework in detail.

Model-View-Controller (MVC) Architecture

It is common to think of an interactive application as having three main layers: presentation (UI), application logic, and resource management. It is reasonable to propose that any given applications is likely to change its interface as time goes by, or indeed have several interfaces at any one point in time. Yet the underlying application might stay fairly constant. For example, a banking application that used to work behind character-based menu systems or command-line interfaces is likely to be the same application that today is working behind a graphical user interface (GUI). As the example illustrates, that it makes sense, to

---

[2] Source: http://phonegap.com/about/feature/

keep the essence of an application separate from any and all of its interfaces. Thus at the core of MMIR is the MVC architecture.

Long ago, in the 70's, Smalltalk defined architecture to cope with this, called the Model-View-Controller architecture, usually just called MVC. MVC paradigm is a way of splitting up your application to that it is easier to change parts of it without affecting other parts. In MVC, the presentation layer is split into controller and view. The most important separation is between presentation and application logic. The View/Controller split is less so.

Models
A model represents the information (data) of the application and manages the behavior of data in the application domain. In the case of MMIR, each model is a JavaScript file intended to store specific data and rules to manipulate that data (for example a model for user which in the simplest case contains the username and password of the application's user).

Views
Views represent the user interface of your application. In MMIR, views are often HTML files with embedded MMIR and JavaScript code that perform tasks related solely to the presentation of the data.

Controllers
Controllers provide the "glue" between models and views. In MMIR, controllers are responsible for processing the incoming requests from input devices, interrogating the models for data, and passing that data on to the presentation manager for presentation.

Basic Components
Figure 2 illustrates the basic components of MMIR framework. The framework is built upon current Web technologies such as JavaScript, HTML5, CSS3 and a range of existing W3C markup languages.
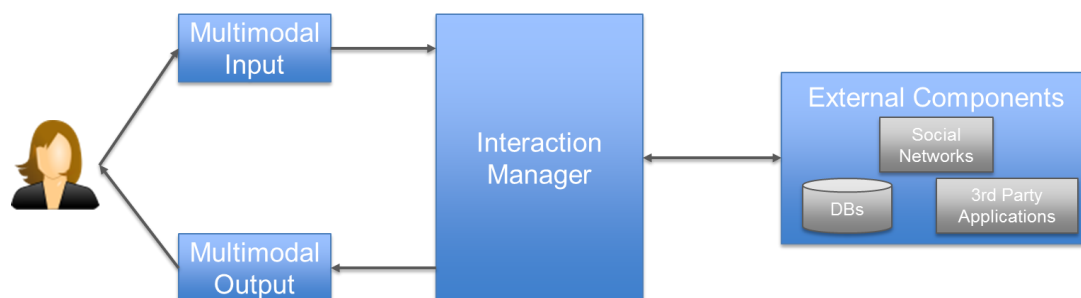

Figure 2: Basic Components of MMIR Framework

Human user
A user who enters input into the system and in turn observes and hears information presented by the system. In the following, we will use the term "user" to refer to a human user.

Input
An interactive multimodal framework should provide multiple input modes such as touch, audio/speech, and gesture.

Output
An interactive multimodal framework should use one or more modes of output, such as speech, text, and graphics.

Interaction manager (IM)

The interaction manager is the central logical component of the MMIR framework. It coordinates the data and manages the execution flow from various input and output modality component interface objects. IM maintains the interaction state and context of the application and responds to inputs from component interface objects and changes in the system and environment.

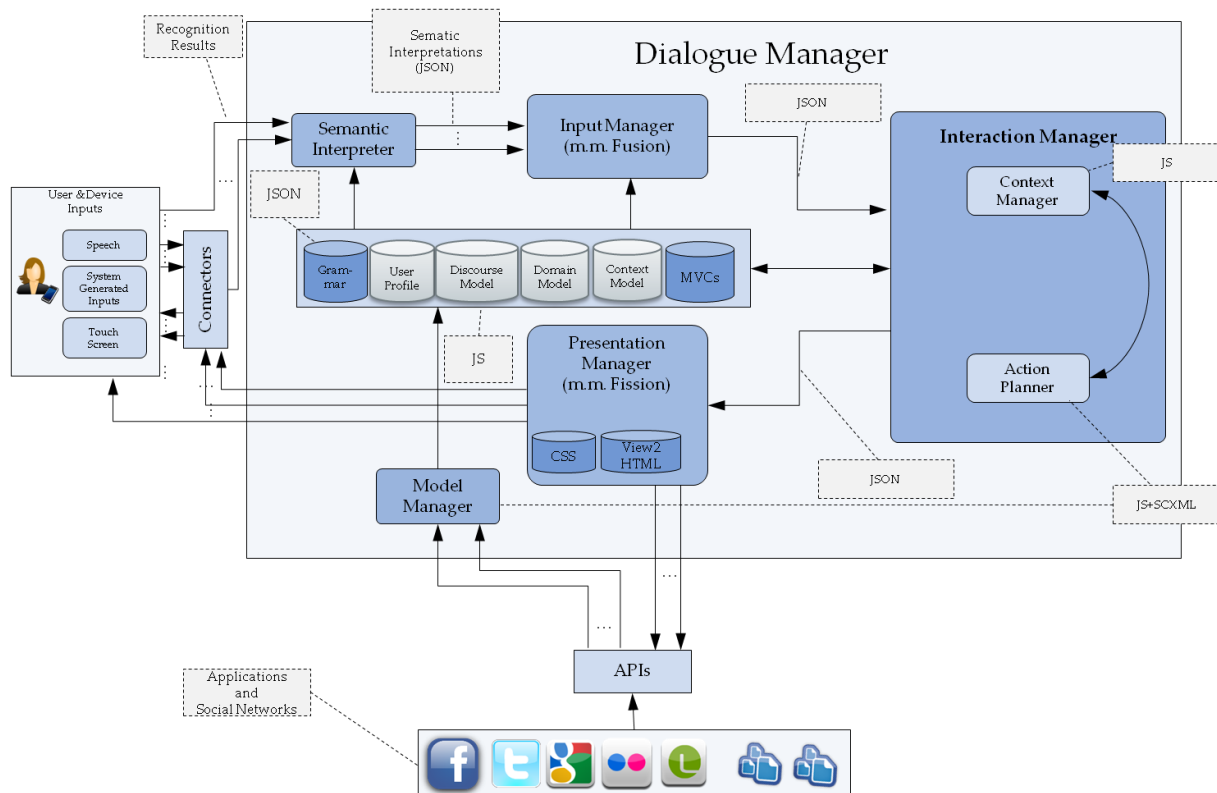## External components (services)

TBD

MMIR Architecture

Figure 3: MMIR architecture

Input Manager (Multimodal Fusion)

In contrast to speech-only-based dialog systems, multimodal dialog systems encompass a number of input modalities that can be employed by the user in an isolated or combined way in order to interact with the system. A classic example of a multimodal interaction is a spoken command like "show me information about this movie" that is accompanied by a pointing gesture for selecting a specific movie that is currently displayed on the screen. In order to trigger an appropriate reaction to such an utterance, a multimodal dialog system needs to integrate the 2 unimodal actions of the user into a coherent multimodal interpretation of their intention. This task is usually carried out by a component called *input manager* or *multimodal fusion.*

In general, the task of a modality fusion component is to combine and integrate all incoming unimodal events into a single representation of the intention most likely expressed by the user. A fusion component has to ensure that every unimodal event that could potentially contribute to the integrated meaning of a multimodal utterance is considered. Thus, a fusion component needs to synchronize the recognition and analysis components so that all unimodal components of an utterance will be taken into account.

Presentation Manager (Multimodal Fission)

The opposite process of multimodal fusion is modality fission where an abstract representation of the content that is about to be communicated to the user has to be distributed over the available modalities for optimal presentation. The presentation manager should generate an appropriate system response by planning the actual content, distributing it over the available modalities, and by finally coordinating and synchronizing the output.

Language Manager

The language manager handles language specific resources. Often this is referred to as localization or internationalization. In general, the language manager handles texts for the user interface. For instance, translations for different languages to allow localization of the application during run time.

TBD: Handling grammars for different languages. Currently MMIR only supports one single grammar definition. In future, support for multiple grammars is planned, e.g. for processing language specific speech input.

Interaction Manager (IM)

The interaction manager is a logical component that controls the flow of the dialog. In MMIR, the interaction manager is in direct communication with the input manager, presentation manager and controllers of the MVC-architecture.

The Interaction Manager (IM) holds the dialog description. The processing of MMIR IM is based on this dialog description in form of State Chart XML (SCXML). SC XML is a standard for describing the flow of an application and for managing interactions among components, and is a natural choice for the IM in a multimodal architecture implementation. SCXML is an XML syntax for describing state machines with the semantics of Harel State Charts. In SCXML, the flow of an application is represented as a set of states and transitions. By sending life cycle events, an SCXML interpreter can start and stop modalities and receive back user input. The SCXML interpreter can transition to other states through application-events based on user or device inputs. After an event is triggered by user or device or 3rd Party applications the action planner selects the possible transition and executes the appropriate actions or/and renders the appropriate view.
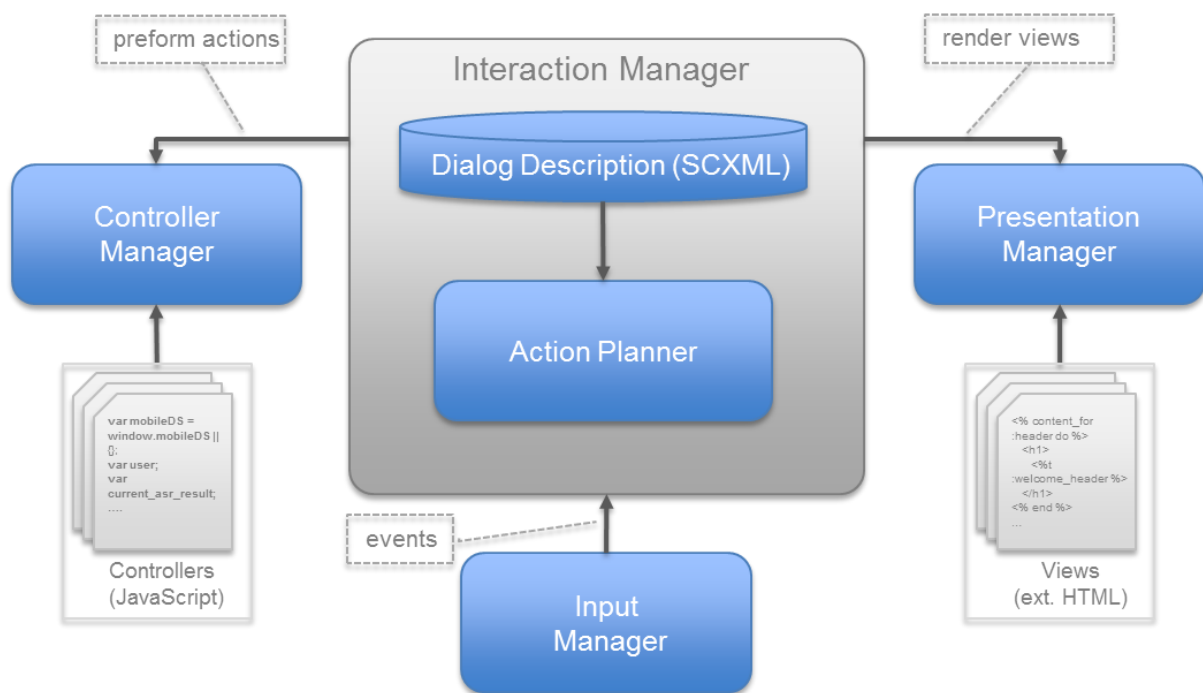

Figure 4: Interaction manager

MMIR Project Structure

This chapter describes the structure of MMIG Starter KIT (MSK). Figure 11 depicts the structure of MSK. If you are familiar with Cordova (previously PhoneGap), you may have noticed that MSK has the same structure as Cordova projects (v2.x).
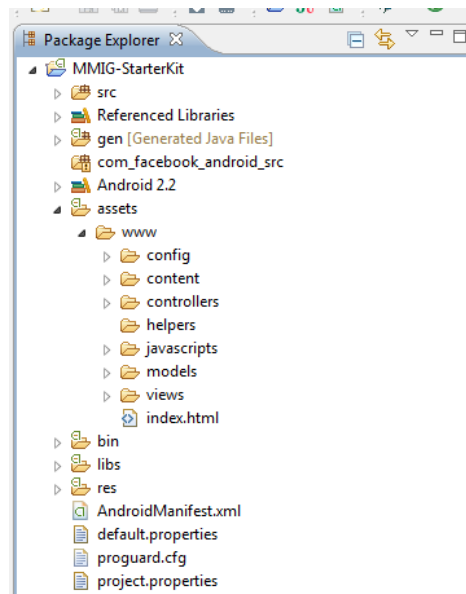

Figure 5: Basic directory structure for a MMIR application

WWW

This folder is the main part of MSK and is the location where the web-based part of your project should be stored. There are some folders and an HTML file in `www` which we will describes in this section. Understanding the contents and structure of this folder is important for extending and developing multimodal applications based on MSK. In the remainder of this chapter we describe the content of this folder in more detail.

Application Configurations (`www/config`)

As the name implies this folder contains the configurations of your application. In the current version of MMIR `config` consists of the following contents:

- `configuration.json`: This is a well-formatted JSON file, which contains the main configuration of MMIR application. For example to specify the default language of your application as German add the following line to `configuration.json` :

  `"language": "de"`

  By convention, languages should be referred to by their 2-letter language code[3], e.g. use `en` for English.

- `grammar`: The `grammar` folder contains the speech grammar of your application (if you intent to use speech recognition in your application). **TBD:** Currently, only 1 grammar definition is supported; future releases will support multiple grammar analogous to the language dictionary files.

  In the current version of MMIR we use the JS/CC parser. You can find a detailed instruction about how to define you JS/CC here. You can also define and test your grammar online at http://jscc.jmksf.com/jscc/jscc.html. There is an HTML file for testing the grammar

---

[3] see ISO 639-1

in/assets/www/testSemanticInterpreter.html. The Ant build file /build.xml provides tasks for compiling an executable grammar from the JSON grammar file.

- config/dictionaries: MMIR is designed to provide an easy-to-use and extensible framework for localizing your application for languages other than English/German or for providing multi-language support in your application. In the following, we refer to this process also as "internationalization". In MMIR, internationalization means to abstract all string-messages, static text in your views, etc. – i.e. separating it from your application code. In the process of internationalizing your application, you have to define dictionaries for the targeted languages. For example if you want to support French in your application, you should define a folder fr in config/dictionaries where you create your French dictionary. Dictionaries are well-formatted JSON-Files. MSK provides two dictionaries for English and German. For details on creating a new dictionary and using it in your application see chapter Add a new Dictionary (p. 17).

Application Contents (www/content)

The content folder contains all images, fonts, and CSS files that you would like to import into your application. The content folder consists of the following folders:

- fonts: Contains the fonts of your application.
- images: Contains the images of your application
- stylesheets: Contains the CSS description of your application.

See chapter Asset Tag Helpers (p. 15) for details on how use and reference the content files.

Application Controllers (`www/controllers`)

This folder contains the controllers of your application. Controllers are one of the most important components of each MMIR application. In MSK you will find the file `application.js` which contains the following code that specifies the controller `Application`:

```javascript
var mobileDS = window.mobileDS || {};

var Application = function(){
    this.name = "Application";
};

Application.prototype.on_page_load = function(ctrl, data){

};

Application.prototype.login = function(){
    var email = $('#emailField #email').val();
    var password = $('#passwordField #password').val();
    mobileDS.User.create(email);
    setTimeout(function(){
      mobileDS.DialogEngine.getInstance().raiseEvent("user_logged_in");
    },0);
};

Application.prototype.register = function(){
    var email = $('#registration-form #email').val();
    var password = $('#registration-form #password').val();
    mobileDS.User.create(email);
    setTimeout(function(){
      mobileDS.DialogEngine.getInstance().raiseEvent("user_logged_in");
    },0);
};
```

<p align="center">Figure 6: Initial code of application controller</p>

The `on_page_load` function will be called after rendering each view of the controller `Application`; this method must be implemented by each controller. This function definition also shows the default arguments for functions in controller implementation: if a function is called via the `DialogManager`, these 2 arguments are always supplied. The first argument is `ctrl`, which is a reference to the Controller class instance, and the second argument is `data`, which is passed on from the call to `DialogManager` (see the example below and the API documentation and for `ControllerManager` and `DialogManager`).

Optionally, you can provide a function `on_page_load_[view name]` for each view (e.g. `on_page_load_welcome` for the view `welcome.ehtml`); this function is called directly after the `on_page_load` function when the corresponding view is rendered, that is if the function exists.

As you can see, there are 2 more functions `login` and `register`, which you might not require in your application. These demonstrate the process of registering a new user and logging in an existing user. If you like, you can remove these methods from application controller or adjust them to your application logic. Note, as these functions do not utilize the arguments `ctrl` or `data`, these are simply omitted.

You also can add new methods to the Application controller. For example, you can add a function for removing an existing user from your database as follows:

```
Application.prototype.remove_user = function(ctrl, data){
    var user_name = data.user_name;
    //write your code here
}
```

`data` is a well-formatted JSON object which contains the information (parameters) which you would like to send to the method (in this case the `user_name`).To call this method from other controllers or scripts in your application you need to first define the data object and call the method through `ControllerManager` as follows:

```
var data = $.parseJSON('{"user_name":"MAX"}');
mobileDS.ControllerManager.getInstance().performAction(
  'application',                          'remove_user',                          data
);
```

Currently, the argument for the name of controller should be in lowercase letters. Please note, that this may change in future version of MMIR, i.e. this argument may become case sensitive.

Application Helpers (`www/helpers`)

Helpers are something of a bridge between controllers and views. They can be used to generate dynamic view-content. In order to respect the MVC pattern, view-specifications should be preferred over helpers wherever possible. If this is not possible, helpers should be used with caution, since they can circumvent the strict separation between controllers and views.

Each controller can optionally have a helper definition. This helper is specified in the folder `www/helpers`. On startup, this folder is scanned and all helper definitions are automatically loaded.

The file containing the helper definition must have the controller's name as prefix followed by `Helper.js`, e.g. for controller `Application`, the helper would be specified in the file `www/helpers/applicationHelper.js`. In this case, the class for the helper would be `ApplicationHelper`:

```
var ApplicationHelper = function(){
    this.name = "ApplicationHelper";
};

ApplicationHelper.prototype.someFunction = function(ctrl, data){
    return "Test User";
};
```

The functions specified in the helper can be invoked from views of the controller, and the String of the return value will be written into the location from where the helper function was called, for example the following statement in a view of the `Application` controller

```
<p>Hello, @helper("someFunction")</p>
```

would mean that on rendering the view, that function `someFunction` in `ApplicationHelper` is invoked with the arguments of controller instance and the `data` argument that was used when issuing the rendering (see API for `DialogEngine.render(viewName, data)`). Then the result of the function call would be inserted into the view, which would result in this example to:

```
<p>Hello, Test User</p>
```

TBD: in future releases, the `@helper` statement will – in addition to the passed-on `data` argument – support an additional argument that can be either a primitive (e.g. a String or Integer) or a JSON object, e.g. `@helper('someFunction', "value")` or `@helper('someFunction', {someStringValue : "the value", "someNumberValue" : 23.4})`.

Application Views (`www/views`)

This folder contains the views of MMIR applications. Each controller can have a set of views which will be rendered by the `PresentationManager`. The view definitions will be loaded automatically when the application is started. Additionally to the specific views, there is a folder containing the layout definitions. The MSK has one layout (`application.ehtml`) which belongs to the `Application` controller. This layout must exists, as it is used as default fallback: In General, each controller can have its own layout (but only one layout per controller) which is used when rendering the views of this controller. If there is no specific layout defined for a controller, the `Application` layout is used by default.

Structuring Views

In MVC based architecture, the view component is responsible for presenting the data to the user in a way that is "palatable" to the user, abstracting from the unnecessary details. In other words, any framework implementing and supporting the MVC pattern should provide means for to ease abstraction and presentation of the required data. This is also true for the MMIR framework, which follows the MVC pattern. The view definition should provide all necessary services for building a dynamic View component. The following are the main services provided by the views:

- Layout (will be described below)
- Templates (the views and partial views; will be described below)
- View Helpers (see chapter Application Helpers (`www/helpers`), p. 13)

Structuring Layouts

When MMIR renders a view, it does so by combining the view with the layout of the view's controller. If the controller has no specific layout defined, the `Application` layout will be used instead. For defining layouts, you have access to 2 tools for combining different bits of output to form the overall response:

- Asset tags
- Yield tags

TBD: future releases will support additional template-view syntax. Support for the following syntax is planned: @render, @helper, @if, @for, @args, @data, @{JS-SCRIPT-CODE}, @(JS-VARIABLE-REF).

Asset Tag Helpers

Asset tag helpers provide methods for generating HTML that link views to JavaScript files and stylesheets. You can use these tags in layouts and even in views, although these tags are most commonly used in the `<header>` section of a layout. Note that the asset tag helpers do not verify the existence of the assets at the specified locations; they simply assume that you know what you are doing and generate the link.

Linking to JavaScript Files with `javascript_include_tag`

The `javascript_include_tag` helper returns an HTML `script` tag for each source provided. This helper generates a link to `/assets/www/javascripts/`. For example, to link to a JavaScript file that is inside a directory called `3rdParty` inside of the `/assets/www/javascripts/` folder, you would do this:

```
@script("3rdParty/jpath")
```

MMIR will then output a script tag such as:

```
<script src='/javascripts/3rdParty/jpath.js' type="text/javascript"></script>
```

Linking to CSS Files with the `stylesheet_link_tag`

The `stylesheet_link_tag` helper returns an HTML `link` tag for each source provided. This helper generates a ling to `/assets/www/stylesheets/`. For example, to link to a stylesheet file that is inside a directory called `ui_style` inside of the `/assets/www/stylesheets/` folder, you would do this:

```
@style("main")
```

MMIR will then generate the following link tag:

```
<link rel="stylesheet" hred="content/stylesheets/main.css"/>
```

Understanding yield

Within the context of a layout, `yield` identifies a section where content from the view should be inserted. The simplest way to use this is to have a single `yield`, into which the entire contents of the view currently being rendered will be inserted:

```
<html>
    <head>
    </head>
    <body>
        @yield("yield-name")
    </body>
</html>
```

You can also create a layout with multiple yielding regions:

```
<html>
    <head>
      @yield("head")
    </head>
    <body>
      @yield("content")
    </body>
</html>
```

The main body of the view will always render into the unnamed yield. To render content into a named `yield`, you should use the `contentFor` method in your view definition.

Using contentFor

The `contentFor` method allows you to insert content into a named yield block in your layout. For example, this view would work with the layout that we just saw:

```
@contentFor("head"){
    <title>A simple page</title>
}@

@contentFor("content"){
    <p>Hello, MMIG! </p>
}@
```

The result of rendering this page into the supplied layout would be this HTML:

```
<html>
    <head>
      <title>A simple page</title>
    </head>
    <body>
      <p>Hello, MMIG</p>
    </body>
</html>
```

The `contentFor` method is very helpful when your layout contains distinct regions such as headers and footers that should get their own blocks of content inserted.

> The order in which the `contentFor` blocks are specified does not matter: in the example above you could also specify the `contentFor` "content" before the "head" – the result would be the same.

Setup the MMIR Applications for Internationalization

Only few steps are necessary to add a new dictionary to your application and set up MMIR to use a new language. These steps are described in the following.

Configuration

To configure your application using a specific language open the `config/configuration.json` file and edit the value for `language`. For example if you want to use English as the language of your application use `en`, for German use `de`.

Add a new Dictionary

The first step of supporting a new language in your application is to add a new dictionary to it. MMIR searches for dictionaries in `config/dictionaries/…` and automatically loads all found dictionaries. MSK provides two dictionaries for the English and the German language (`config/dictionaries/en/dictionary.dic`, `config/dictionaries/de/dictionary.dic`). Dictionaries are well-formatted JSON-Files. The example below illustrates the English dictionary of MSK:

```json
{
        "mmig": "MMIG",
        "login_header": "Please login",
        "password_place_holder": "password",
        "user_name_place_holder": "user name",
        "login_label": "Login",
        "registration_text": "or register yourself",
        "registration_label": "Sign Up",
        "mainPanelAudioConfirmation": "Audio Confirmation",
        "buttonOk": "Ok",
        "buttonCancel": "Cancel",
        "buttonBack": "back",
        "ratingStar": "star",
        "ratingStars": "stars",
        "dialogCapture": "Capture",
        "dialogPlay": "Play",
        "welcome_header": "MMIG",
        "welcome_date": "some Date",
        "welcome_text":"Welcome to MMIG"
}
```

If you want to add support for French in your application: create a new dictionary file in the appropriate folder, i.e. `config/dictionaries/fr/dictionary.dic`. The new dictionary will be available by its folder name, in this case `fr`. You can use this e.g. in the `config/configuration.json` file, or the `LanguageManager`.

For this example, first copy the contents of the English dictionary into the newly created file (or just copy the English dictionary file into the new sub-folder) and replace the English values with French translations:

```json
{
        "mmig": "MMIG",
        "login_header": "S'il vous plapla&icirc;t connectez-vous",
        "password_place_holder": "mot de passe",
…
```

Adding Translations

Translations are looked up by keys defined in dictionaries.

```
<label for="Login">
  @localize("login_label")
</label>
```

On rendering the value for the currently set language will be used to replace the statement. For the setting en as language and the example dictionary from above, the result would be:

```
<label for="Login">
  Login
</label>
```

Getting Started

This chapter describes the system and software requirements for developing multimodal mobile interaction applications using MMIR-framework for Android operating system.

Supported Operating Systems

-   Windows XP, Vista, or Windows 7
-   Mac OS

Requirements

For development based on the MMIR Starter Kit application

-   Download and install [Eclipse](#) (3.4+) (e.g. the *Classic*, or *EE Developer* edition)
-   Download and install [Android SDK](#) (2.2+)
-   Download and install [ADT Plugin](#)
-   Download the MMIG-StarterKit.zip

You should also ensure that you have created at least one Android virtual device (AVD). On how to create a new AVD in Eclipse, we refer to the [online instructions](#). You will need an AVD to run your project in the Android emulator. You should have the Android **tools** and android **platform-tool** folders listed in your system path. Both these folders can be found inside your Android installation directory.

Setting up Eclipse

While using Eclipse as development environment is not strictly required, it is recommended. The following gives some hints for setting up Eclipse in order to ease development of MMIR based applications.

First, you should download and install the Android SDK, and then install the ADT Plugins (e.g. using the update site); see chapter Requirements.

Hint: If your Eclipse environment brings an HTML editor (e.g. as the EE edition does) you can setup Eclipse to use this editor for eHTML files. eHTML is the template format used by the framework, similar to JSP (Java Server Pages) or ASP (Active Server Pages) templates.

For setting the HTML editor as default editor: **Preferences ► General ► Editors ► File Associations**, then in **File types**, add an entry for ***.ehtml** and set the HTML editor as its default.
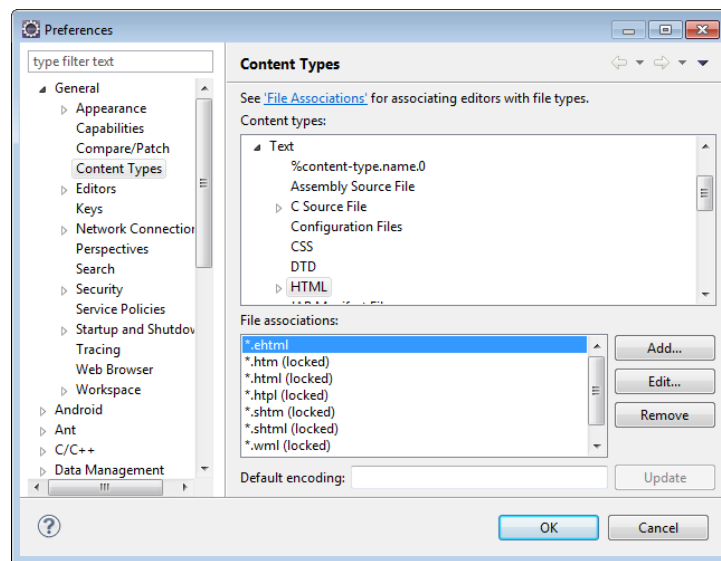
Figure 7: Eclipse Preferences dialog for associating file extensions with content types.

For using the syntax highlight of the HTML editor: in **Preferences ► General ► Content Types** select **Text ► HTML**, then in **Text ► File Associations** add an entry ***.ehtml**.

Figure 8: *.ehtml file without (left) and with (right) syntax highlighting for content type HTML in Eclipse.

Importing MMIG-StarterKit into Eclipse

To import the MMIG-StarterKit (MSK) to Eclipse follow these steps:

1. Start Eclipse and go to **File ► Import.**
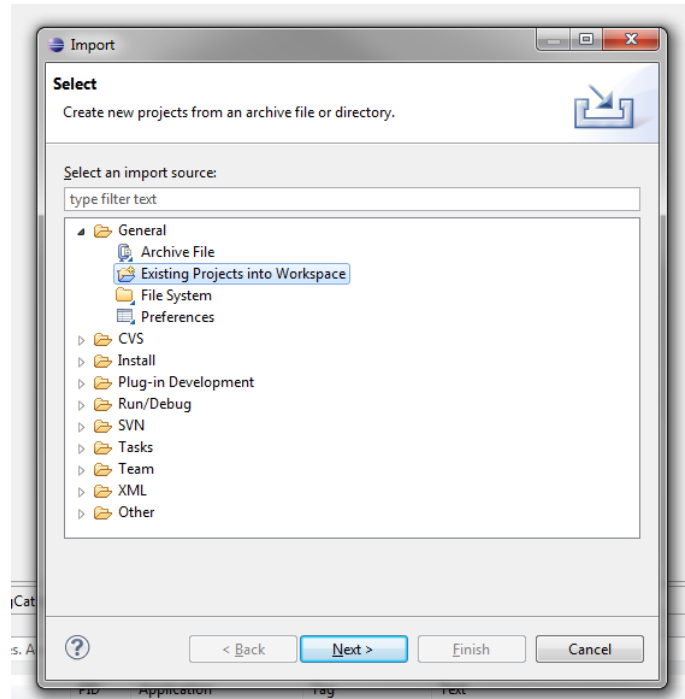2. Select **Existing Project into Workspace** and click the **Next** button.

Figure 9: Import an Existing Project into Eclipse

3. Click the radio button next to **Select archive file** and click the **Browse** button on the following dialog.
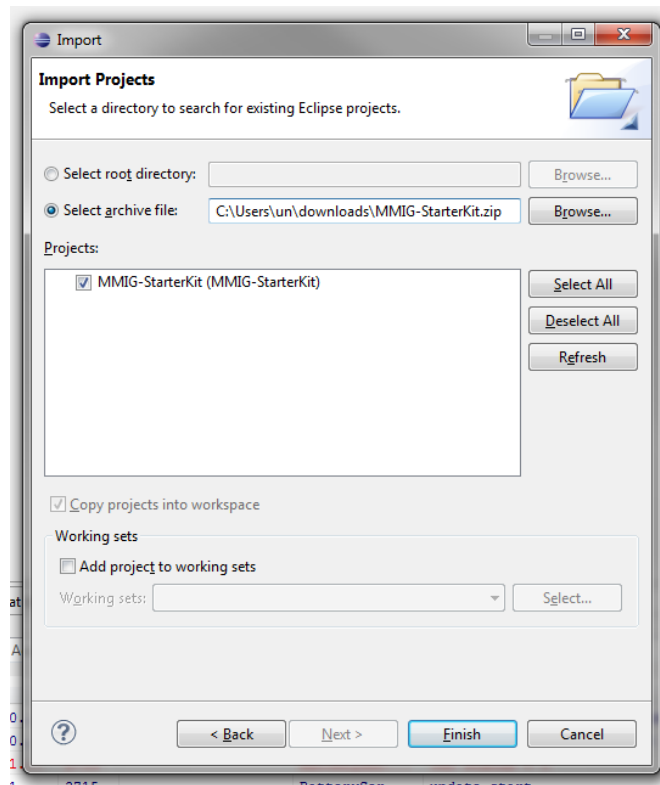
Figure 10: Select MMIG-StarterKit.zip

4. Navigate to MMIG-StarterKit.zip on your disk. Click **open** to select it.

MSK is based on the Apache Cordova platform (previously PhoneGap) and has almost the same structure as a Cordova project. Figure 11 illustrates the structure of MSK in Eclipse.

Figure 11: Structure of MSK

Deploy to Simulator

- Right click the project and go to **Run As** and click **Android Application**.
- Eclipse will ask you to select an appropriate AVD. If there does not exist one, then you need to create it.

Deploy to Device
- Make sure USB debugging is enabled on your device and plug it into your system (**Settings ► Applications ► Development**).
- Right click the project and go to **Run As** and click **Android Application.**

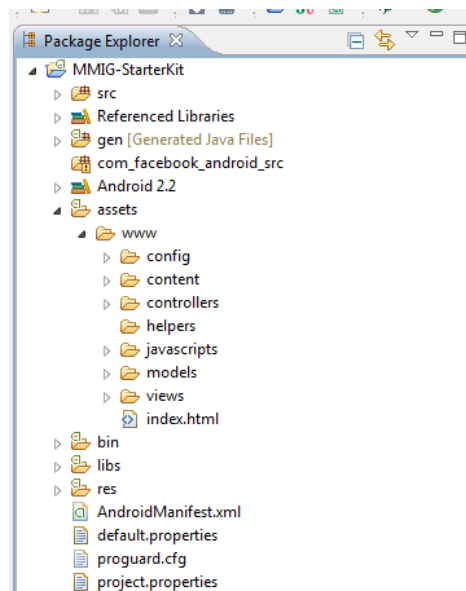**Deployment for Web Site**
- Open `assets/www/index.html` in a web browser

Notes on support for speech interactions:

- Currently, only Google Chrome (version 25.x.x) supports access to the microphone
    - o To enable Chrome for accessing the microphone activate the **Web Audio Input** in `chrome://flags/`
    - o On opening the `index.html` choose `Accept` for granting access to the microphone
- MSK is setup to use the Google web service for speech recognition. This requires converting the audio stream from the microphone in WAV format to the FLAC format. **TBD:** a lightweight web service component will be released (e.g. to be run on a Tomcat server), that allows conversion from WAV to FLAC

Run MSK

After deploying MSK to the simulator or your device and depending on the specified language of your application (to configure the language of MSK see chapter Configuration, p. 17), you should see one of the screens depicted in Figure 12.

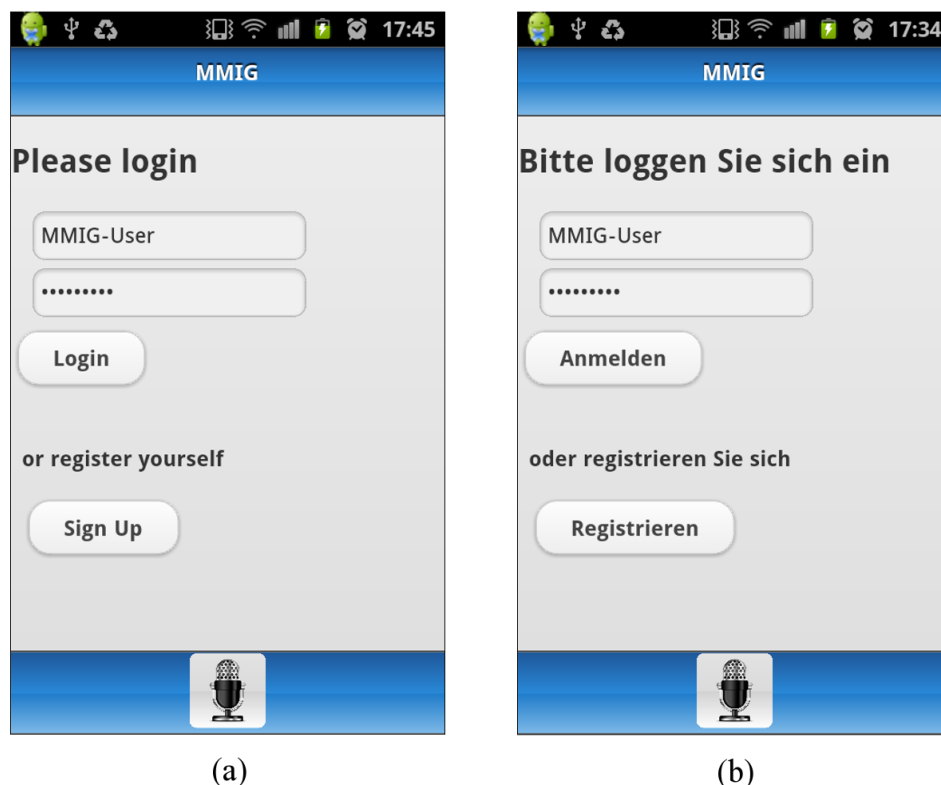(a)   (b)

Figure 12: Start screen of MSK. (a) for English (b) for German

MSK provides a minimal set of touch interactions. It simulates a simple login/sign up process which can be triggered by pressing Login (dt.: *Anmeldung*) or Sign Up (dt.: *Registrieren*) buttons respectively. Figure 13 depicts the welcome and sign up screen of MSK.
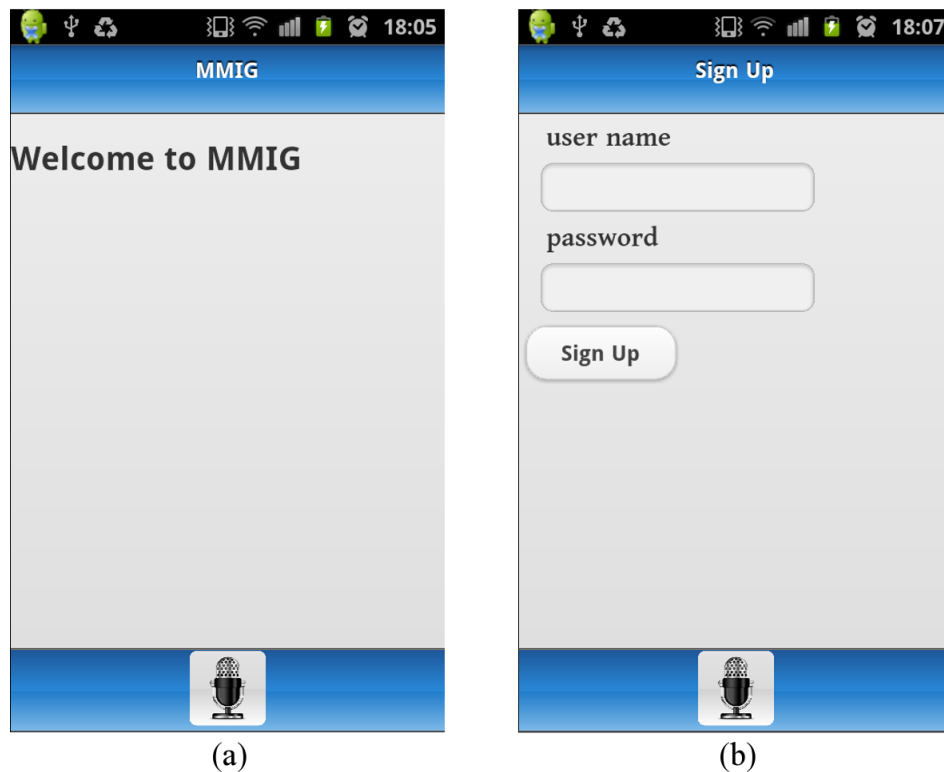
Figure 13: *Welcome* (a) and *sing up* (b) screens of MSK

Expanding MSK

Controllers

Expand an existing controller

Expanding an existing controller means to add some new methods (actions) to its JS code. For example you can add a `remove_user` method to the application controller to remove an existing user from your database as follows:

```
Application.prototype.remove_user = function(data){
    var user_name = data.user_name;
    //write your code here
}
```

`data` is a well-formatted JSON object which contains the information (parameters) that you would like to send to this method (in this case the `user_name`). To call this method from other controllers or scripts in your application you first need to define the data object and then call the method through `ControllerManager` as below:

```
var data = $.parseJSON('{"user_name":"MAX"}');
mobileDS.ControllerManager.getInstance().performAction(
    'Application','remove_user',                                    data
);
```

Add new controllers

Let us make a Calendar controller with an action of create appointment, which will create a new appointment for the user. To do this:

1- In Eclipse, navigate to www/controllers.
2- In controllers folder create a new file and name it calendar.js.
3- Copy the following code in calendar.js

```javascript
//application namespace
var mobileDS = window.mobileDS || {};

//controller's constructor
var Calendar = function(){
    this.name = "Calendar";
}

//This method will be called before rendering the views of this controller.
Calendar.prototype.on_page_load = function(){

}
```

This is the basic skeleton of each controller. You can use this code each time you would add a new controller to your application. You just have to replace Calendar with the name of your controller.

4- Add the following method to the calendar controller:

```javascript
Calendar.prototype.create_appointment = function (ctrl, data){
  var container_id = data.container_id;
  var container = $("#" + container_id);
  var subject = $("#" + container_id + " #subject").val();
  var   year   =    $("#"   +   container_id   +   "   #app-date").data('datebox')
              .theDate.getFullYear();
  var month = $("#" + container_id + " #app-date").data('datebox').theDate.getMonth()
            + 1;
  var day = $("#" + container_id + " #app-date").data('datebox').theDate.getDate();
  var   start_h   =   $("#"   +   container_id   +   "   #start-time").data('datebox')
              .theDate.getHours();
  var   start_m   =   $("#"   +   container_id   +   "   #start-time").data('datebox').theDate
              .getMinutes();
  var end_h = $("#" + container_id + " #end-time").data('datebox').theDate.getHours();
  var   end_m   =   $("#"   +   container_id   +   "   #end-time").data('datebox')
              .theDate.getMinutes();

  var eventData = '{"subject":"' + subject + '","year":"' + year +
              '","month":"' + month+'","day":"' + day +
              '","start_hours":"' + start_h+'","start_minutes":"' + start_m +
              '","end_hours":"' + end_h+'","end_minutes":"' + end_m+ '"}';

  var jData = jQuery.parseJSON(eventData);
  var cb_func = function(){
    alert("appointment for " + jData.subject + " successfully created!");
  };
  mobileDS.CalendarModel.getInstance().save_appointment(jData, cb_func);
}
```

Views

Add a new view for an existing controller

To add a new view for an existing controller, you have just to add a new `.ehtml` file to its view folder (`www/views/controller_name`). For example, if you need a new view for application controller, which should be displayed if the registration of a new user failed, you simply could add a new file named `reg_failed.ehtml` to `www/views/application` where you then put the definition of this view.

Add views of a new controller

Adding the views of a new controller is as simple as the process of adding a new view to an existing controller. The only difference is that you first need to create the corresponding folder in `www/views.` In order to add the only view of calendar controller, create a new file in `www/views/calendar`, name it `create_appointment.ehtml`, and copy the following code into it.

```
@contentFor("header"){
    <h1>
        @localize("create_appointment_header")
    </h1>
}@

@contentFor("content"){
  <div id="create_appointment">
     <label for="subject">
       @localize("subject_label")
     </label>

     <input type="text" name="subject" id="subject" value="">
     <label for="app-date">
       @localize("date_label")
     </label>

     <input name="app-date" id="app-date" type="date" data-role="datebox"
           data-options='{"mode": "flipbox"}' >
     <label for="start-time">
       @localize("start_time_label")
     </label>

     <input name="start-time" id="start-time" type="date" data-role="datebox"
           data-options='{"mode": "timeflipbox"}' >
     <label for="end-time">
       @localize("end_time_label")
     </label>

     <input name="end-time" id="end-time" type="date" data-role="datebox"
           data-options='{"mode": "timeflipbox"}'>
     <button id="save_appointment" name="save_appointment_btn" data-
           inline="true">
       @localize("save")
     </button>

     <button id="discard_appointment" name="discard_appointment_btn" data-
           inline="true">
       @localize("discard")
     </button>

  </div>
}@
```

As the last step in adding a new view, the MMIR framework needs to be told, that there is an additional view, that is needs to process (i.e. load on startup of the application). The default Ant task

`generateFileListJSONFile` in `/build.xml` creates the appropriate information by updating the file `parsed_directory.json` in `assets/www/config/`.

The MMIR framework knows about all components that it needs to load by looking into the JSON file `/assets/www/config/parsed_directory.json`.

You can automatically create an updated version of this file by using the Ant task `generateFileListJSONFile` in `/build.xml` (this is the default target in the Ant build file). To run the task, e.g. open the context menu on `/build.xml` and select **Run As ► Ant Build**.

You should update the information in `parsed_directory.json` always after creating a new file for

- A controller in   `www/controllers/`
- A helper in   `www/helpers/`
- A model in   `www/models/`
- A view or layout in   `www/views/`
- A plugin in   `www/javascripts/plugins/`
- A dictionary in   `www/config/languages/`

Models

Add a new Model

Now that we have the calendar controller and the required view, it is time to add the calendar model. To add a new model into your application, you have to navigate to www/models and add a new JavaScript file which has the same name as your model (in lowercase letters). We name the new model calendarmodel.js which should contain the following code:

```javascript
var mobileDS = window.mobileDS ||
{};

mobileDS.CalendarModel = (function(){
    var instance;

    function constructor(){
        var calendar_server_url = 'http//...';
        return {

            save_appointment: function(data, cb_func){
                //We suppose the appointment should be created at a server
                //running on 'calendar_server_url' and send the information
                //via POST request to that server.

                var user_name = mobileDS.User.getInstance().getName();

                $.ajax({
                    type: 'POST',
                    url: 'calendar_server_url?user_name=' + user_name,
                    data: data,
                    success: cb_func()
                });

                //if you only want to test that you have successfully pass the
                //process of adding new model-contrller-view comment out the
                //whole
                //ajax request and add just call the call back function as
                //follow:

                //cb_func();
            }

        }
    }

    return {
        getInstance: function(){
            if (!instance) {
                instance = constructor();
            }
            return instance;
        }
    }
})();
```

After creating the new model, we – again – need to update the file parsed_directory.json in assets/www/config/ by invoking the default Ant task generateFileListJSONFile in /build.xml (see chapter Add views of a new controller, p. 26).

Expand Applications Dictionaries

By default, MSK supports two languages: German and English. To expand the English dictionary you have to edit `www/config/languages/en/dictionary.dic`. Open it in Eclipse and add the following key value pairs to the JSON definition of your dictionary.

```
"create_appointment": "Make an appointment",
"create_appointment_header":"Appointment",
"date_label" : "Date",
"start_time_label":"Start",
"end_time_label":"End",
"subject_label":"Subject",
"save":"Save",
"discard":"Discard"
```

If your new application should support German language, you have to expand the German dictionary also. Open `www/config/languages/de/dictionary.dic` and copy the following key value pairs into the definition of your German dictionary.

```
"create_appointment": "Termin erstellen",
"create_appointment_header":"Termin",
"date_label" : "Datum",
"start_time_label":"Start",
"end_time_label":"Ende",
"subject_label":"Betreff",
"save":"Speichern",
"discard":"Verwerfen"
```

Expand Dialog Description (SCXML)

As described in described in section Interaction Manager (IM), MMIR defines the dialog flow of an application by specifying transitions between application states using SCXML. The description file is located at `www/config/statedef/dialogDescription.xml`. This description is automatically loaded and used via the `DialogEngine`: The dialog engine recognizes events and dispatches them to controllers' actions and/or renders controllers' views based on transition specified in dialog description.

Figure 14 shows the dialog description of MSK. This description is depicted in Figure 15 as a finite state automaton (FAS). In this description `start` is the initial state of our dialog. The `init` transition (the transition from `start` state to the `main_state`) means that after raising the `init` event which will be raised automatically by MSK after loading all required component, the dialog engine have to take a transition from start `state` to the `main_state` followed rendering the login view of application controller, as specified in `onentry` actions of `main_state`. For more information on SCXML, we recommend reading the W3C SCXML standard.

```xml
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
    profile="ecmascript" id="scxmlRoot" initial="start">

    <state id="start" name="start">
        <transition event="init" name="init" target="main_state" />
    </state>

    <state id="main_state" name="main_state">
        <onentry>
            <script>
                dialogManager.render('Application', 'login');
            </script>
        </onentry>

        <transition event="touch_end_on_login_btn" target="login_user" />
        <transition event="touch_end_on_sign_up_btn" target="registration_form" />
    </state>

    <state id="registration_form" name="registration_form">
        <onentry>
            <script>
                dialogManager.render('Application', 'registration');
            </script>
        </onentry>
        <transition event="touch_end_on_register_btn" target="try_to_register_new_user" />
        <transition event="back" name="back" target="main_state" />
    </state>

    <state id="try_to_register_new_user" name="try_to_register_new_user">
        <onentry>
            <script>
                dialogManager.perform('Application', 'register');
            </script>
        </onentry>
        <transition cond="mobileDS.User.getInstance() == null" target="main_state" />
        <transition cond="mobileDS.User.getInstance() != null" target="logged_in" />
    </state>

    <state id="login_user" name="login_user">
        <onentry>
            <script>
                dialogManager.perform('Application','login');
            </script>
        </onentry>
        <transition event="login_failed" target="main_state" />
        <transition  event="user_logged_in"  cond="mobileDS.User.getInstance()  !=  null"
                target="logged_in" />
        <transition  event="user_logged_in"  cond="mobileDS.User.getInstance()  ==  null"
                target="main_state" />
    </state>

    <state id="logged_in" name="logged_in">
        <onentry>
            <script>
                dialogManager.render('Application', 'welcome');
            </script>
        </onentry>
        <transition event="touch_end_on_appointment_btn" target="create_appointment" />
        <transition event="back" name="back" target="main_state" />
    </state>

</scxml>
```
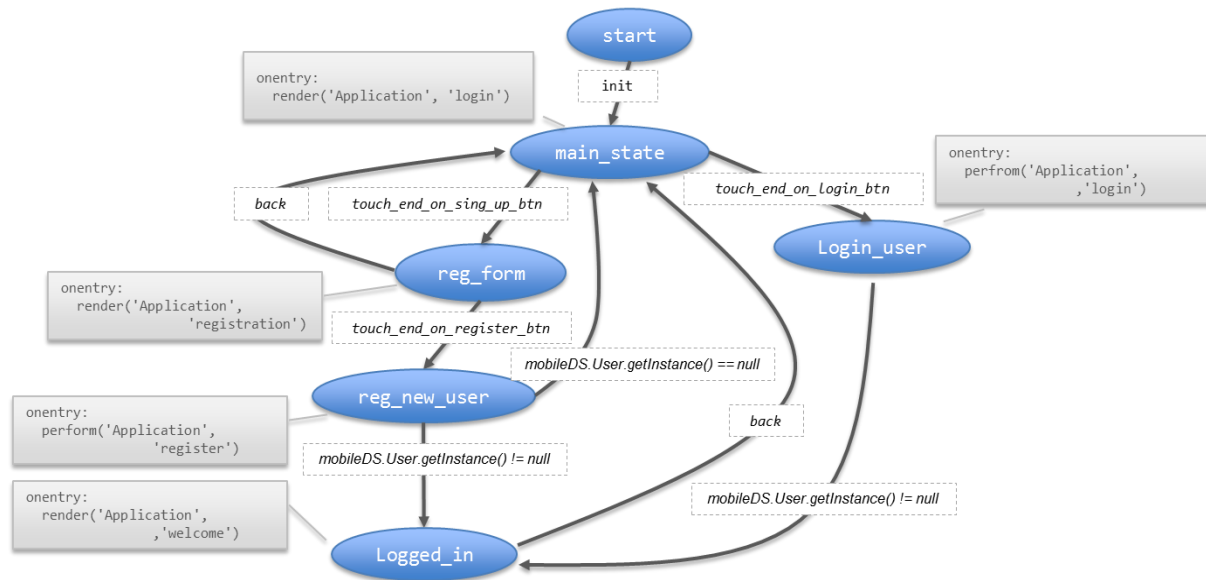
Figure 14: Dialog Description of MSK

Figure 15: Dialog Description of MSK as FSA

Expanding MSK with MVC does not extend the behavior of your application. For this, you have to extend the dialog description of MSK. There are also some modifications required in the `welcome` view of application controller. We start with the `welcome` view. As you have seen in Figure 13, after logging in, there is no more interaction possible for the user. Now we want to allow the user to create a new appointment. To do this, we add a new button to the `welcome` view. The new `welcome` view of your application should look as follows:

```
@contentFor("header"){
    <h1>
        @localize("welcome_header")
    </h1>
}@

@contentFor("content"){
    <h2>
        @localize("welcome_text")
    </h2>
    <button id="appointmentButton" name="appointment_btn" data-inline="true">
        @localize("create_appointment")
    </button>
}@
```

Now we have to add a transition in our dialog description which should be taken after raising the `touch_end_on_appointment_btn` event. To do this navigation to `www/javascripts/scxml` and open the `dialogDescription.xml` file. Find the `logged_in` state and add the following transition to it:

```xml
<transition event="touch_end_on_appointment_btn"
            traget="create_appointment" />
```

We now need to add the new `create_appointment` state. The following code snippet contains the definition of our new state.

```xml
<state id="create_appointment" name="create_appointment">
  <initial>
    <onentry>
      <script>
        dialogManager.render('Calendar', 'create_appointment');
      </script>
    </onentry>
  </initial>

  <transition event="touch_end_on_save_appointment_btn" target="save_appointment" />
  <transition event="touch_end_on_discard_appointment_btn" target="logged_in" />
  <transition event="back" name="back" target="logged_in" />

  <state id="save_appointment" name="save_appointment">
    <onentry>
      <script>
        var data = jQuery.parseJSON('{"container_id":"create_appointment"}');
        var result = dialogManager.perform('Calendar','create_appointment',data);
      </script>
    </onentry>
    <transition target="logged_in" />
  </state>
</state>
```

In MSK the event flow does not go directly to `DialogEngine` (i.e the dialog dialog description), but is handled through the `InputManager`. Generally, first an event is raised on the `InputManger` signaling the modality: For touch events, the `touch_input_event` is raised. Then, the actual "meaningful" event is raised on the `InputManager`, e.g. `touch_end_on_login_btn`.

Analogously for a speech input, a `speech_input_event` would be raised first, followed by an event submitting the ASR (Automatic Speech Recognition) result.

See also `config/statedef/input_manager_scxml.xml` for the state chart specification of the MSK.

After adding the new state to our dialog description, we are can start the application and inspect our newly added modifications. `Fig 11` shows the new screens of calendar application.
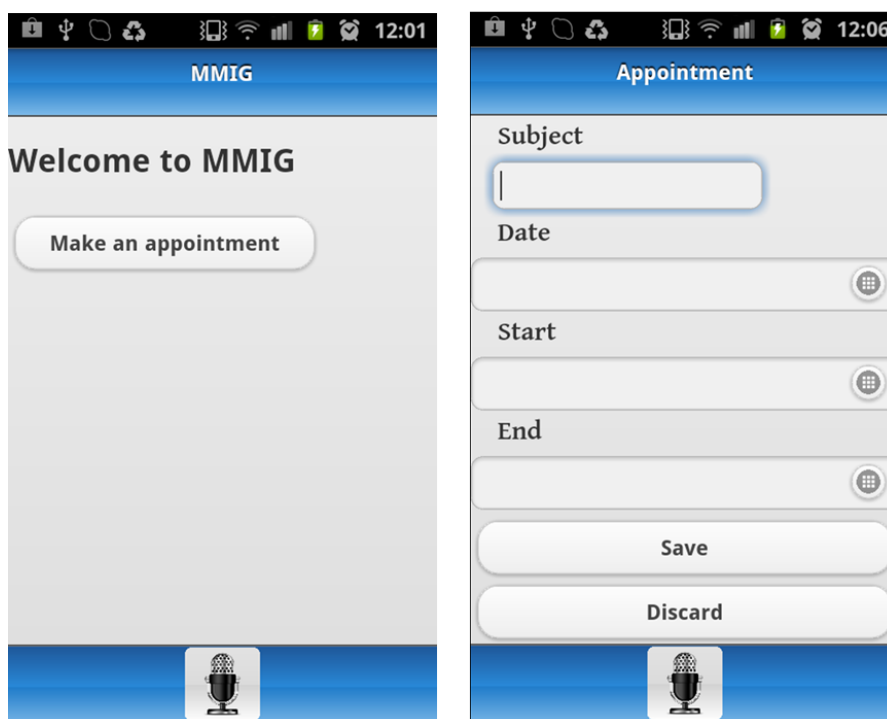

Figure 16

In order to test, if the SCXML files have valid syntax, we can explicitly compile the SCXML files (previous versions of MMIR required the resulting JavaScript files of the compilation, but the current version interprets the SCXML files directly using the `scion` library).

For this compilation we use the `common-scxml-js` tool of Apache. You can download it from here (**TBD**: the sandbox Apache project common-scxml-js is not online anymore; an alternate download location has to be prepared). Before using `common-scxml-js`, you have to edit the `/build.settings` file and modify it according to your environment, e.g. point to the directory where you have copied this tool. Then you can compile the SCXML files using the appropriate tasks in the Ant build file `/build.xml`.