MMIR framework

Multimodal Mobile Interaction and Rendering Framework

Aaron Ruß, Ehsan Gholamsaghaee, Florian Petersen
23.08.2013

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).

Pro:

- 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.

Contra:

- 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 users 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.

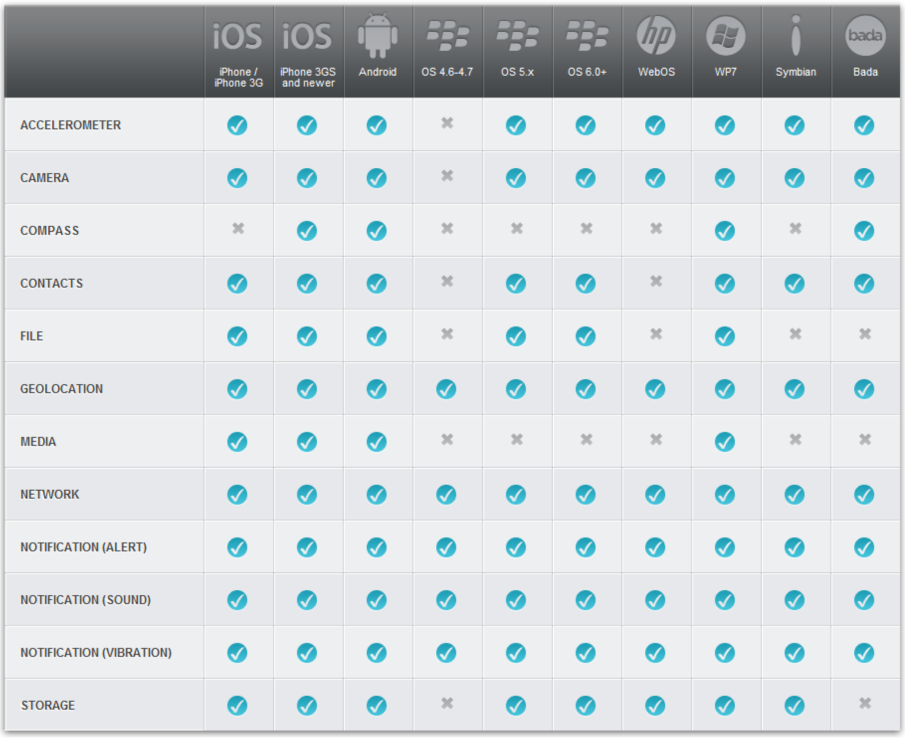
Hybrid Approach

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/.



	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	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗

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

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 *(Multimodal Mobile Interaction and Rendering)* 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 1 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.

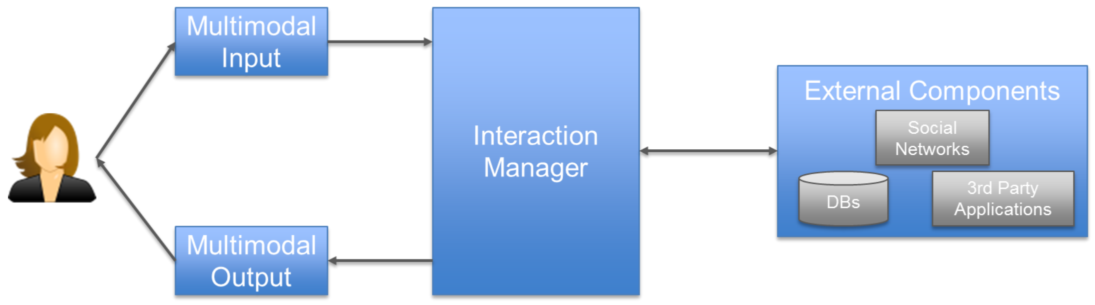


Figure 1: Basic Components of MMIR Framework

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

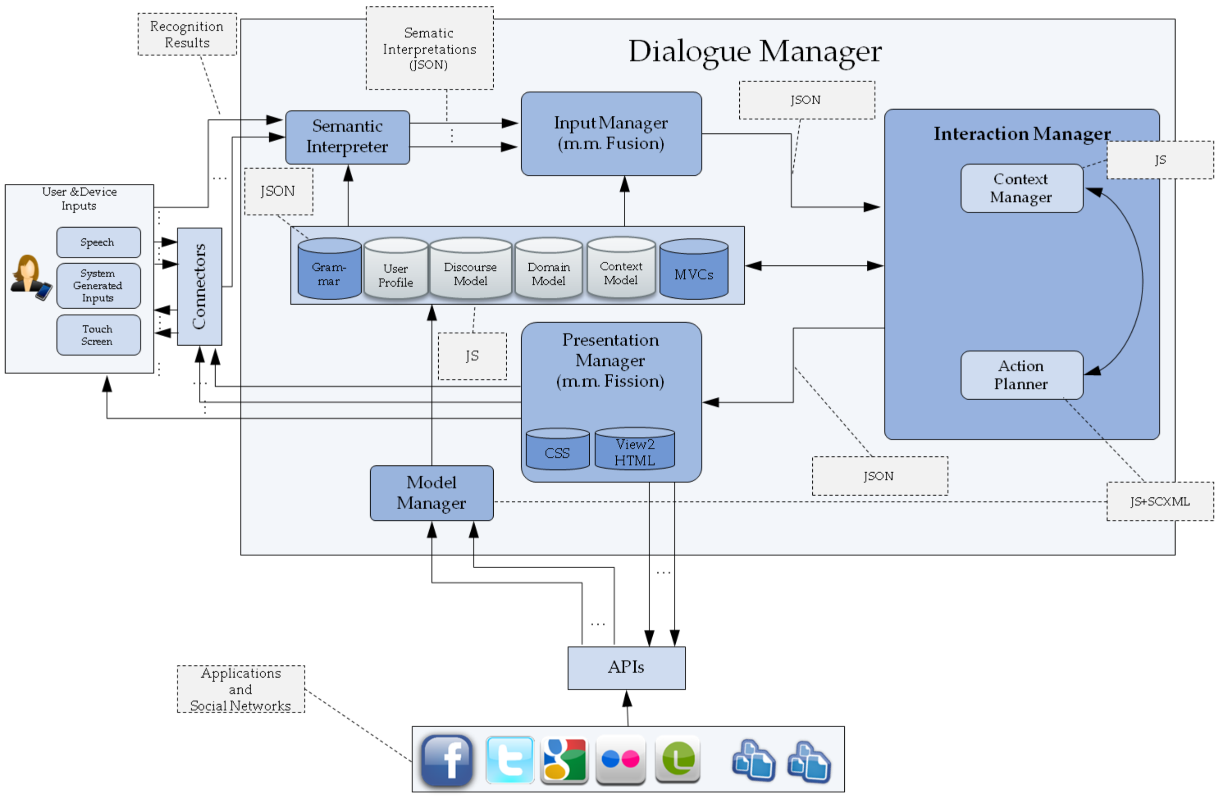


Figure 2: MMIR architecture

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

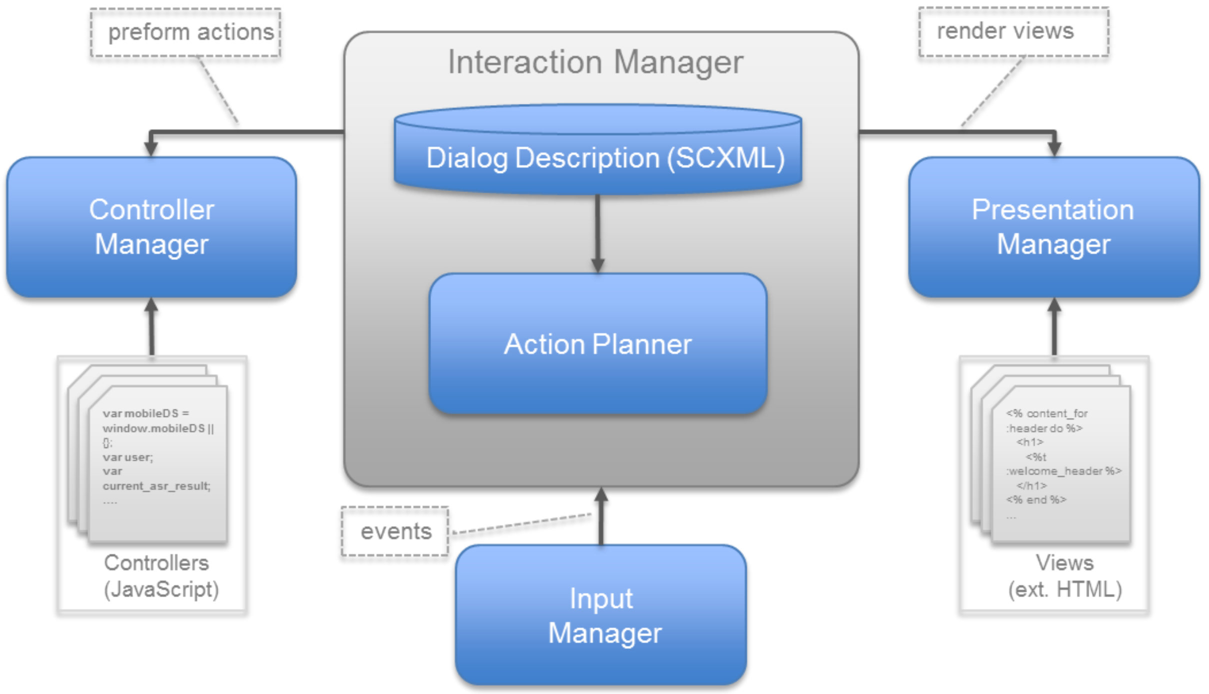
The language manager handles language specific resources. Often this is referred to as localization or internationalization. In general, the language manager handles texts for the user interface. For instance, translations for different languages to allow localization of the application during run time.

The language manager also handles other language specific resources, e.g. speech grammars (for processing speech input) and speech synthesis (TTS: text to speech).

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.



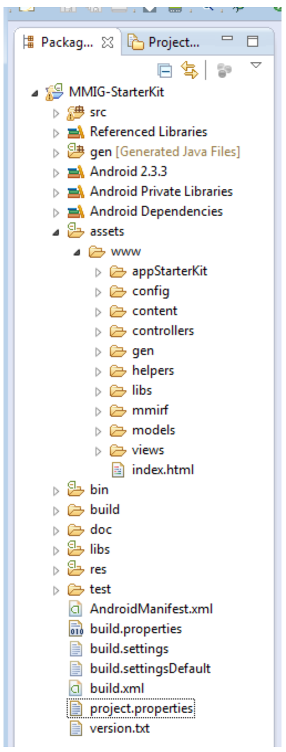
```
var mobileDS =
window.mobileDS ||
{};
var user;
var
current_asr_result;
....
```

```
<% content_for
:header do %>
  <h1>
    <%t
:welcome_header %>
  </h1>
<% end %>
...
```

Figure 3: Interaction manager

MMIR Project Structure

This chapter describes the structure of MMIG Starter KIT (MSK). Figure 10 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).



```
MMIG-StarterKit
  src
  Referenced Libraries
  gen [Generated Java Files]
  Android 2.3.3
  Android Private Libraries
  Android Dependencies
  assets
    www
      appStarterKit
      config
      content
      controllers
      gen
      helpers
      libs
      mmirf
      models
      views
      index.html
  bin
  build
  doc
  libs
  res
  test
  AndroidManifest.xml
  build.properties
  build.settings
  build.settingsDefault
  build.xml
  project.properties
  version.txt
```

Figure 4: Basic directory structure for a MMIR application

Naming Conventions for Models, Views, and Controllers

The framework uses naming conventions in order to keep the configuration simple. Following this convention means, that you use the same name for controller-files (which hold the implementation for the controller), controller-class (the JavaScript class of the controller implementation), and the controller's view directory (the directory which holds all view definitions for the controller) etc.

As a result, you do not have to make additional configurations that tell the framework for instance "view XY for controller AB can be found in location IJ". Instead the framework derives all this information from the controller's name.

	Description	Example
[controller name]	the controller's class name	`CalendarExample`
\controllers\[controller name].js	File name for controller implementation	`calendarExample.js`
\views\[controller name]\[view name].ehtml	A view definition for the controller	`calendarExample\main.ehtml`
\views\layouts\[controller name].ehtml	Layout definition for the controller's views	`calendarExample.ehtml`

Table 2: Naming convention for controllers and views

The controller's class name should always start with an upper case letter. The file in which the controller's class is specified may begin with a lower case letter, but except for this, must have the same name.

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.

- `config/languages/`: Holds language specific configurations; see the following section *Application Language Configurations (`www/config/languages`).*
- `config/statedef/`: Contains the SCXML files for the `DialogEngine` and the `InputEngine`; see the following section *Application State Definitions (`www/config/statedef`).*
- `directories.json`: This is a well-formatted JSON file, which contains a list of files that hold the implementations of the models, views, and controllers of the application, as well as language-specific resources (e.g. dictionaries, grammars).
  The framework uses this information, to automatically load the models, views, controllers, plugins, and language resources at startup. The `directories.json` file can be created using the project's **ANT** `build.xml`; see the following section *Building MSK.*

Application Language Configurations (`www/config/languages`)

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, support for internationalization is provided by a mechanism for abstracting all string-messages, static text in your views, etc. – i.e. separating it from your application code. This is done by creating dictionaries for the targeted languages. For example, if you want to support French in your application, you should define a folder `fr` in `config/languages` where you create your French dictionary. Dictionaries are well-formatted JSON-Files. MSK provides two example 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. 26).

The language directories also hold the grammar definitions for parsing speech input: `grammar.json`.

The `grammar` file contain the speech grammar in a well-formed JSON format. If you intent to use speech recognition in your application, you should also provide a grammar file for each language. For more details see *Add a new Grammar* (p. 28).

[3] see ISO 639-1 specifiaction

The file `speaker.json` contains the configuration for the text-to-speech (TTS) component, e.g. which voice should be used when synthesizing to text to audio-output. **NOTE***: the details for this configuration may be subject to change.*

Application State Definitions (`www/config/statedef`)

This directory contains the SCXML definitions for the DialogEngine and the InputEngine (see also Expand Dialog Description (SCXML), p. 46).

TBD: expanded description.

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 Layout Template Expression (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`:

```
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().raise("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().raise("user_logged_in");
    },0);
};
```

Figure 5: Initial code of application controller

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 DialogEngine, 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 DialogEngine (see the example below and the API documentation and for ControllerManager and DialogEngine).
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 follow the guidelines described in section *Naming Conventions for Models, Views, and Controllers* (p. 10).

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>
```

See sections *Layout Template Expressions* (p. 15) and *View Template Expressions* (p. 17) for more template expressions that can be used within template files for views.

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 exist, 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)

Layout vs. View vs. Partial

The main concept for displaying content to the user (i.e. the GUI) is the *view*. Layouts and partials can be considered "helper constructs" that assist in rendering the view for display to the user.

The main differences between layouts, views, and partials are notably:

Layouts: are used to specify the "skeleton" for all views of a controller. The layout contains the static, non-changing content that stays the same for all views, and the layout contains the `@yield` declarations, which are named *section-declaration* that will be "filled in" by the specific view's `@contentFor` definitions.
Located: /views/layouts/[controller-name].ehtml

Views: specify the content that should be filled into the layout's `@yield`-sections; i.e. the content that is specified with the `@contentFor` expressions is rendered into the corresponding `@yield` sections of the view's layout. In difference to layouts, views can contain various template expressions for dynamically creating content, e.g. `@for`, `@if`, `@{code}@`, …
NOTE: content in a view that is not specified within a `@contentFor` expression will be ignored when rendering the view into its layout.
Located*:* /views/[controller-name]/[view-name].ehtml

Partials: are usually small "partial content" definitions (e.g. for re-usable content definitions) that are rendered "as-is", i.e. partial templates do not use `@contentFor` section definitions (as views do), but instead all the partial's content is rendered into the referring view. A partial can be included into a view by using the `@render` expression.
NOTE: the file-names of partials are prefixed with ~ (tilde). However, when referring to partials (e.g. using the `@render` expression) this character is omitted.
Located*:* /views/[controller-name]/~[partial-name].ehtml

Structuring Layouts

When MMIR renders a view, it does so by combining the view with the controller's layout. If the controller has no specific layout defined, the `Application` layout will be used instead. For defining layouts, you have access to 2 template expressions:

- Asset tags (i.e. template expressions `@script` and `@link`)
- Yield tags (i.e. template expression `@yield`)

Note that in views (and partials) additional template expression can be used, see section *View Template Expressions* (p. 17).

Layout Template Expressions

Template expressions in layouts provide methods for generating HTML that link layouts to external JavaScript and stylesheet files. 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 `@script`

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

```
@script("datebox/jpath")
```

MMIR will then output a script tag such as:

```
<script src='/libs/datebox/jpath.js' type="text/"></script>
```

Linking to CSS Files with `@style`

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

```
@style("ui_style/main")
```

MMIR will then generate the following link tag:

```
<link rel="stylesheet" hred="content/stylesheets/ui_style/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.

View Template Expressions

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.

Using @localize

The framework allows an easy way of localization. It uses a dictionary to lookup the corresponding localized text for a supplied key:

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

Note that the language must be set and a dictionary exist, before the localization can be used properly.

On rendering the `localize` part is replaced by the corresponding value for the currently set language. For the setting `en` as language and an example dictionary, presented in section Add a new Dictionary, the result would be:

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

Commentary @ … *@*

Commentary is surrounded by `@*` and `*@`. Comments will not be displayed in the rendered view in any way. All content that is within a comment section is not processed by the parser and will be removed before the view is rendered.

A little example of a commentary in action:

```
<html>
   <head>
     @* Here comes the title of the view *@
     <title>A simple page</title>
   </head>
   <body>
     @* and here is the body contents *@
     <p>Hello, MMIG! </p>
   </body>
</html>
```

This will result in:

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

The commentary is simply removed.

Template Dynamic/Code Expressions

Using template variables @var

Template variables can either be displayed in the view (see section ***Rendered javascript***) or manipulated inside a JavaScript code segment – see following sections. The access to the variable is limited to the surrounding contentFor block of the view (see section *Using `@contentFor`*). If the variable is used within a partial, it is accessable throughout the whole partial as a partial is an implicit contentFor section.

```
@var(title_string)
@{@title_string = "Title String"}
<html>
   <head>
     <title>@(@title_string)</title>
   </head>
   <body>
     <p>>@(@title_string)</p>
   </body>
</html>
```

The use of `@var` and the `@` prefix makes the variable's scope local for this template. This also means that no global variable with the same name is overwritten or the value from a global variable is used instead of a local created variable.

The variable will only be replaced by its value on rendering, if it is used inside a (rendered) javascript section – see section ***Rendered javascript code @(…)***. Simply adding @variable to a view will just result in rendering @variable at this position – instead of the value of var.

If a variable is used inside a javascript code block (unrendered or rendered), it should be initiated with `@var()` and referenced with `@` to ensure that the variable is not overwriting a global variable. Note that variables are not permitted inside a layout description.

Unrendered javascript code @{ … }

This expression evaluates the containing javascript code but does not render the return value. Inside the code block standard javascript can be used. See also the section about variables for a brief explanation about the scope of variables.

The following snippet is an example of how to use the unrendered javascript code:

```
@{@page_name = "simple page";}
@{
   @dots="";
   for (@i = 0; @i < 30; @i = @i + 1){
      @dots = @dots + ".";
   }
}
<html>
   <head>
     <title>A simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
   </body>
</html>
```

The first line simply assigns a value to the variable `page_name`. The section beginning on the second line creates a string with 30 dots. All those commands are simply executed and nothing is going to be rendered in the final view. It will just look like this:

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

Note that the unrendered javascript code block can contain multiple lines of code.

Rendered javascript code @(…)

This expression evaluates the containing javascript code and displays the return value into the location of the template expression. Inside the code block standard javascript can be used. See also the section about variables for a brief explanation about the scope of variables.

This example shows the usage of a rendered javascript code:

```
@{@page_name = "simple page";}
<html>
   <head>
     <title>A simple page with the name @(@page_name)</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
   </body>
</html>
```

The variable `page_name` is replaced with the string assigned to it, resulting in the following code:

```
<html>
   <head>
     <title>A simple page with the name simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
   </body>
</html>
```

As seen above, this construct is used to display template variable values.

Note that the rendered javascript code block cannot contain multiple lines of code – it is mainly used for the display of template variables. In contrast, the unrendered javascript code can contain multiple lines of code.

Template View/Controller Expressions

Calling a helper function @helper

If a helper method is defined for a controller, it can be called from within a view. The helper's return value is then rendered into the view at the position of the helper call.

The syntax of the helper call is:

`@helper(`functionName [, params]**`)`**

The function name is the name of the helper function. No controller has to be supplied – it is always the controller of the current view. The helper call can also be supplied with arguments via `params`.

As the view is rendered, the helper method is called and its return value embedded in the currently processed view.

If a `params` argument is passed to a helper method via the `helper` call, the data can be referenced inside the called helper by using the `arguments` variable. This variable holds the arguments of the helper call.

This snippet calls a helper method named `createListElements` with some arguments and puts the return value in place of the method call:

```
<html>
   <head>
     <title>A simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
     @helper('createListElements', {obj1 : 'value1', obj2 : 'value2'})
   </body>
</html>
```

Rendering a partial @render

To insert a partial anywhere in the view, the `@render` statement is used. This processes the referenced partial and displays the rendered result at the position of the call.

The syntax for the `@render` statement is:

`@render(`controllerName, partialName, data**`)`**

The `controllerName` is the name of the controller which is associated with the partial, while the `partialName` is the name of the partial.

The `data` argument allows for the passing of arguments to the partial, e.g. ascertained by a helper method. The type of the data argument can be one of the following: a string / number literal, template variable or a JSON like object, which itself can consist of an array, string / number literal and a JSON like object.

If a data argument is passed to a partial via the `render` call, the data can be referenced inside the called partial by using the `@argument` variable. This variable holds the arguments of the render call.

This render call of a partial may display a menu to select a language and supply a title as well as a personal salutation to the user:

```
<html>
   <head>
     <title>A simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
     @render('Application', 'languageMenu', {displayTitle : true, salutation
: "Bob"})
   </body>
</html>
```

Inside the partial `languageMenu`, the data of the render call, `{displayTitle : true, salutation : "Bob"}`, may be referenced via the `@argument` variable.

Template Control Expressions

TBD: add details and examples for template expressions (see example for partial in /views/application/~languageMenu.ehtml).

Conditional rendering @if

For the control of the rendered view output, the `@if` construction can be used to display parts of the code only if a defined condition is true.

The syntax is:

`@if` (javascript condition) **`{`**part to display if condition is true – view expressions or HTML**`}@`**[**`@else{`** part to display if condition is false – view expressions or HTML **`}@`**]

The else-part is optional. Inside the if segments all view expressions, including HTML, can be used.

An example of a conditional rendering:

```
@{@debug = true;}
<html>
   <head>
     @if(@debug == true){
       <title>Debug mode</title>
       @{@debug_time = new Date();}
     }@ @else {
       <title>A simple page</title>
     }@
   </head>
   <body>
     <p>Hello, MMIG! </p>
     @if(@debug == true){
     <div id="debug_info"></div>
     }@
   </body>
</html>
```

This would result in the rendered view:

```
<html>
   <head>
       <title>Debug mode</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
     <div id="debug_info"></div>
   </body>
</html>
```

The if statement can be used hierarchically – just like in javascript.

Loops with @for

For the generation of similar view sections, e.g. menu items, the `@for` statement can be used. The `@for` statement can be used in two ways: to iterate over an object or it is used as a counter.

The syntax is:

`@for (`initialization; condition; afterthought**`) {`** loop body **`}@`** or

`@for (`item **`in`** object**`) {`** loop body **`}@`**

The following snippet creates a list of the work days of the week by using a counter:

```
<html>
   <head>
     <title>A simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
     @{ @daysArray = ["mon", "tue", "wed", "thu", "fri"]; }
     <ul>
     @for(@i=0, @size = @daysArray.length; @i < @size; ++ @i){
        <li>@(@daysArray[@i])</li>
     }@
     </ul>
   </body>
</html>
```

This results in:

```
<html>
   <head>
     <title>A simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
    <ul>
         <li>mon</li>
         <li>tue</li>
         <li>wed</li>
         <li>thu</li>
         <li>fri</li>
     </ul>
   </body>
</html>
```

The same can be achieved by using the iterative way of the `@for` statement:

```
<html>
   <head>
     <title>A simple page</title>
   </head>
   <body>
     <p>Hello, MMIG! </p>
     @{ @daysArray = ["mon", "tue", "wed", "thu", "fri"]; }
     <ul>
     @for(@i in @daysArray){
         <li>@(@daysArray[@i])</li>
     }@
     </ul>
   </body>
</html>
```

Special Statements

At sign @@

If the at sign, @, should be displayed in the view, it must be “escaped” by another “@”: “`@@`” will display “`@`”.

Accessing function arguments @argument

Inside a called helper function or rendered partial, the supplied data can be referenced by either referencing the `arguments` variable inside the helper function or referencing the `@argument` template variable inside a partial.

A partial can reference the supplied arguments easily with the `@argument` template variable:

```
<div>The supplied arguments are: @(JSON.stringify(@argument))</div>
```

Inside a helper function, the supplied arguments can be referenced by the `arguments` variable:

```
var args = arguments;
for (var a in args){
   // process the arguments
}
```

Render data @data

If a call to render a view is also supplied with arguments, these can be accessed with the @data template variable. Calls to render views are typically generated by the state machine and can be supplied with additional parameters to create contextual rendered output.

A call to render the loginManager view may look like this:

```
dialogManager.render('Application', 'login', {languageMenu : true});
```

Inside the view `login`, the property `languageMenu` can be accessed by referencing to `@data.languageMenu`, which will yield `true`.

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/languages/…` and automatically loads all found dictionaries. MSK provides two example dictionaries for English and German language support (`config/languages/en/dictionary.dic`, `config/languages/de/dictionary.dic`).

Dictionaries are well-formatted JSON-Files. The example below illustrates the English dictionary of MSK:

```
{
	"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/languages/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:

```
{
	"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>
```

Add a new Grammar

If you want to use speech interactions in your application, you should also provide grammars for all languages. Grammars are used to process the ASR from a speech recognizer: it “translates” *natural language* input into “programmatic instructions”, e.g. for input phrase `“please find movie XY”`, the result of executing the grammar could be something like

```
{
   search: "XY",
   displayResult: true
}
```

This is only an example, the concrete result is defined within the grammar and generally depends on the application, i.e. you will have to encode the mapping `phrase` → `result` by specifying the grammar.

Grammar definitions in MMIR are similar to *context free grammars*: input sentences are matched against grammar rules (in MMIR: utterances/phrases). The grammar rules (MMIR: phrases), may refer to other rules and/or to tokens-definitions. The following example shows a pseudo grammar for illustration:

```
TOKEN1:      "some","few"
TOKEN2:      "thing","things","object","objects"
TOKEN3:      "else"
…
RULE1:       TOKEN1 TOKEN3 TOKEN1
RULE2:       RULE1 TOKEN2
```

MMIR searches for grammars in `config/languages/…` and automatically loads all found grammars. MSK provides an example grammar for German (`config/languages/de/grammar.json`).

Grammars are defined in the form of well-formatted JSON files. The following example is an excerpt from the German grammar.json of MSK and illustrates a possible grammar definition for “translating” natural language phrases like *“bitte abspielen”* (*please play*), *“spiele ab”* (*play*) etc. into an event object `Play` (i.e. the result that would be returned, when the grammar matches the phrase: `{semantic: Play{}}`). This event object can then be used for further processing (this example is an excerpt from the German `de/grammar.json` of MSK):

```
{
   "stop_word": [
        "bitte",
        "doch",
        "der",
        "m__oe__chte",
        …
    ],
    "tokens": {
        "PREPOSITION": [
            "an",
            "um",
            "am",
            "ab",
            …
        ],
        "V_PLAY_IMP": [
            "spiel",
            "spiele"
            …
        ],
        "V_PLAY_INF": [
            "spielen",
            "abspielen",
            "h__oe__ren",
            …
        ],
        …
    },
    "utterances": {
      "PLAY": {
          "phrases": [
                "V_PLAY_INF",
                "V_PLAY_IMP PREPOSITION"
          ],
          "semantic": {
            "Play": {}
          }
      }
        …
    }
}
```

TBD: descriptions for following details

- grammar definition details:
  - stopwords, tokens, utterances, semantic-resuts
  - lower-case “restriction”: token values should all be lower case
  - umlaut encoding (e.g. ä → __ae__)
- grammar loading/selection mechanism
- grammar compilation (compile grammar.json with build.xml)

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

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): in Eclipse select the **Java** perspective and open from the menu **Window** ► **Android Virtual Device Manager**; for detailed instructions 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.

Use **Window** ► **Android SDK Manager** for installing the SDK of the Android version you are targeting with your application; at least one SDK has to be installed (we recommend to use at least ver. 2.2+).

Setting up Eclipse

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

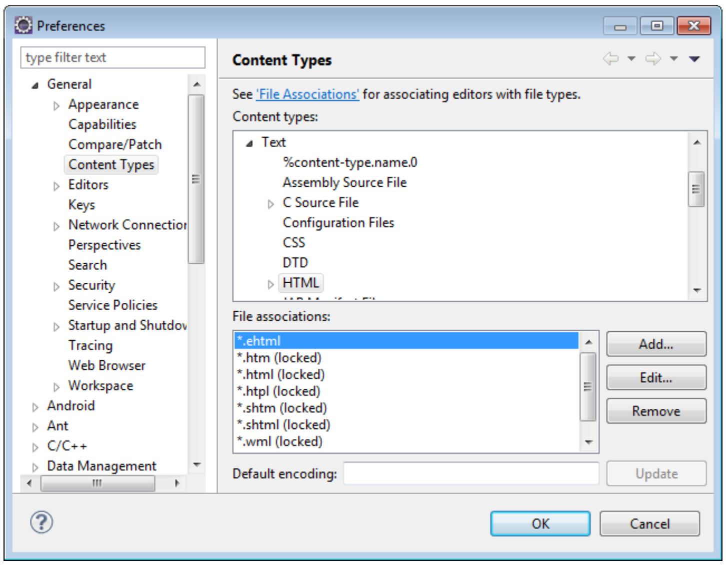
First, you should download and install the Android SDK; you should note the directory where the SDK is installed into, as you may need this information later. Next, install the ADT Plugins (e.g. in Eclipse, using the update site); see also chapter Requirements (p. 31), above.

HTML Editing

Hint: If your Eclipse environment brings an HTML editor (e.g. as the **EE Developer edition** does)[4] 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.

[4] *Eclipse Classic* does not bring an editor HTML by default; you can install an editor manually e.g. using your Eclipse's update page (*Install New Software…*) *http://download.eclipse.org/releases/[version name]*, expand the entry *Web, XML and Java EE development* and select *Web Page Editor*.



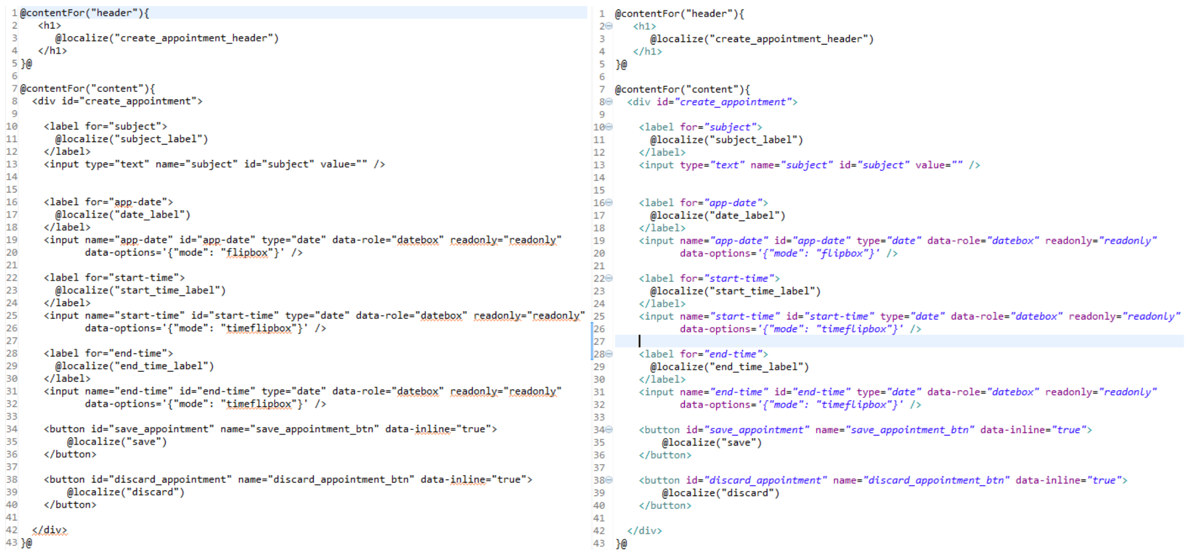
See 'File Associations' for associating editors with file types.
Content types:
Text
%content-type.name.0
Assembly Source File
C Source File
Configuration Files
CSS
DTD
HTML

File associations:
*.ehtml
*.htm (locked)
*.html (locked)
*.htpl (locked)
*.shtm (locked)
*.shtml (locked)
*.wml (locked)

Default encoding:

Figure 6: 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**.



```
@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" readonly="readonly"
           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" readonly="readonly"
           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" readonly="readonly"
           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>
}@
```

Figure 7: *.ehtml file without (left) and with (right) syntax highlighting for content type HTML in Eclipse.

Android Source Code

For debugging purposes, you may want to configure Eclipse to have access to the Android source code. For Android 4.x you can use the SDK Manager, e.g. in Eclipse in the Java perspective, open **Window** ► **Android SDK Manager**, expand the entry for the corresponding Android 4.x version and select **Sources for Android SDK** for installing the source. When prompted with No sources attached... for Android code (e.g. during debugging), select the `[Android SDK directory]/sources/[version]` subdirectory for the version currently used in your project from the Android SDK installation directory, e.g. `/android-sdk/sources/android-16`.

For some of the older Android versions[5], the plugin from http://code.google.com/p/adt-addons/ ► **Android Sources** can be used to integrate source code support. You can use the update site

[5] Note, that currently no source code is available for Android ver. 3.x.

http://adt-addons.googlecode.com/svn/trunk/source/com.android.ide.eclipse.source.update/ for installing the **Android Sources** plugin.

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.

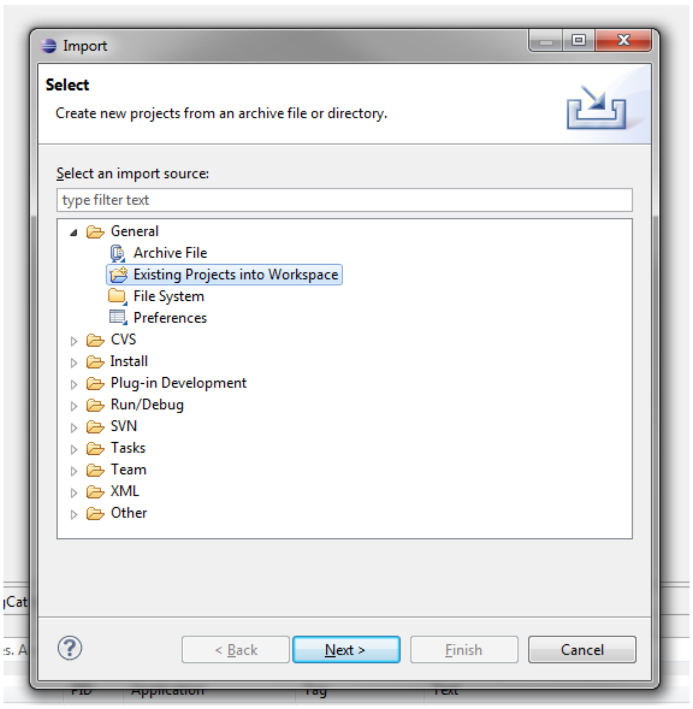


Figure 8: Import an Existing Project into Eclipse

3. Click the radio button next to **Select archive file** and click the **Browse** button on the following dialog.

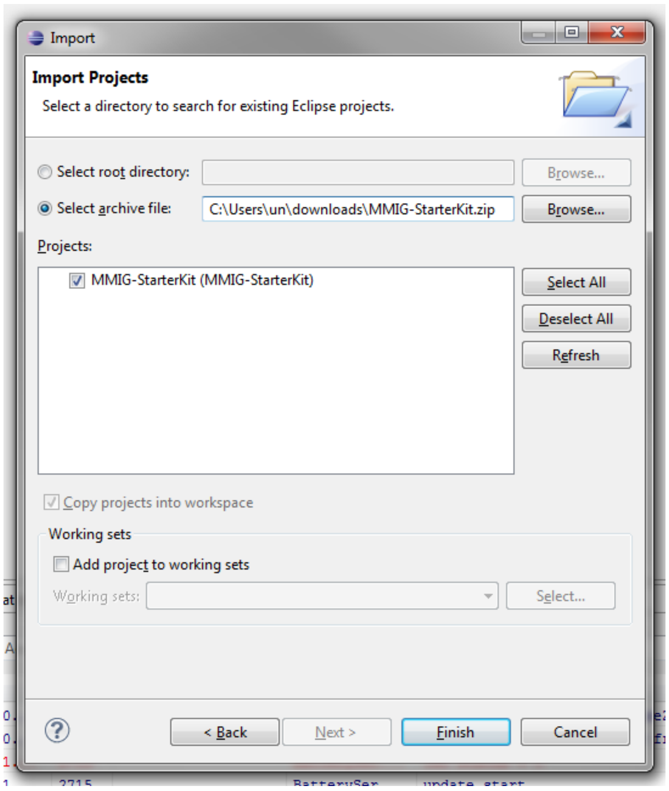
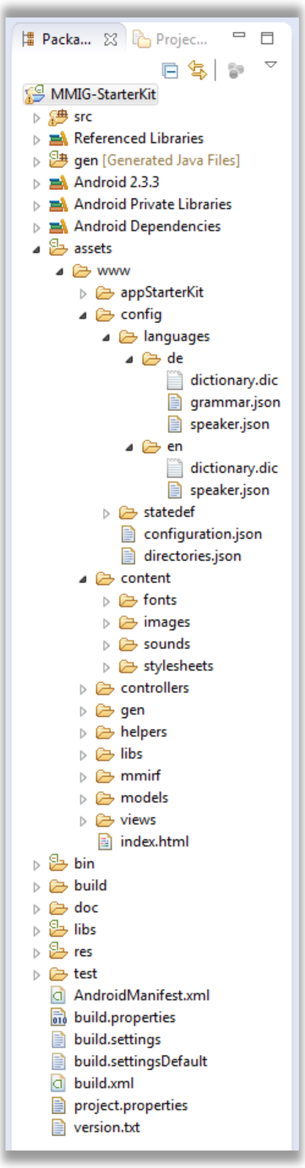

Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
C:\Users\un\downloads\MMIG-StarterKit.zip
Browse...
Projects:
MMIG-StarterKit (MMIG-StarterKit)
Select All
Deselect All
Refresh
Copy projects into workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel

Figure 9: 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 and has almost the same structure as a Cordova project. Figure 10 illustrates the structure of MSK in Eclipse.



```
Packa...  Projec...
MMIG-StarterKit
  src
  Referenced Libraries
  gen [Generated Java Files]
  Android 2.3.3
  Android Private Libraries
  Android Dependencies
  assets
    www
      appStarterKit
      config
        languages
          de
            dictionary.dic
            grammar.json
            speaker.json
          en
            dictionary.dic
            speaker.json
        statedef
        configuration.json
        directories.json
      content
        fonts
        images
        sounds
        stylesheets
      controllers
      gen
      helpers
      libs
      mmirf
      models
      views
      index.html
  bin
  build
  doc
  libs
  res
  test
  AndroidManifest.xml
  build.properties
  build.settings
  build.settingsDefault
  build.xml
  project.properties
  version.txt
```

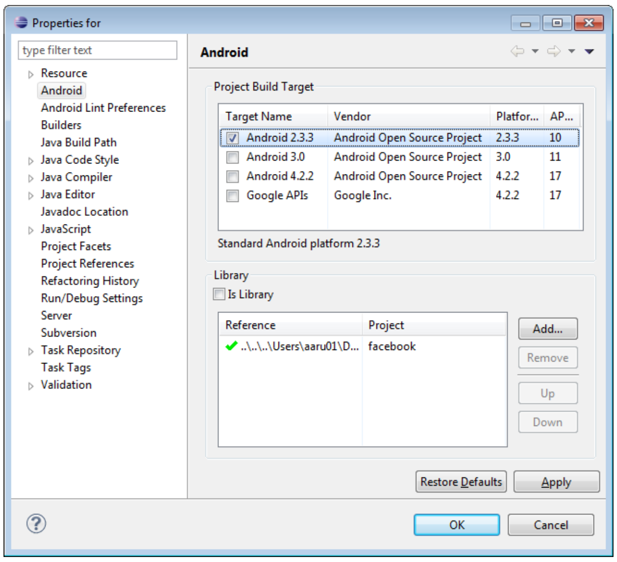
Figure 10: Structure of MSK

Common Problems

The following sections gives help for some common problem and issues that may occur when importing the MMIG-StarterKit into Eclipse.

If you experience compilation errors, some of the following steps may help:

- After importing, set the correct Android version: open the project's **Properties** ▶ **Android**, in the section **Project Build Target** select the Android version you target. If you reference other Android projects, ensure that they are correctly linked in the **Library** section; if in doubt, go through the steps to **Remove**, **Apply**, and re-**Add…** for the referenced projects.



Android

Project Build Target

Target Name	Vendor	Platfor...	AP...
✓ Android 2.3.3	Android Open Source Project	2.3.3	10
Android 3.0	Android Open Source Project	3.0	11
Android 4.2.2	Android Open Source Project	4.2.2	17
Google APIs	Google Inc.	4.2.2	17

Standard Android platform 2.3.3

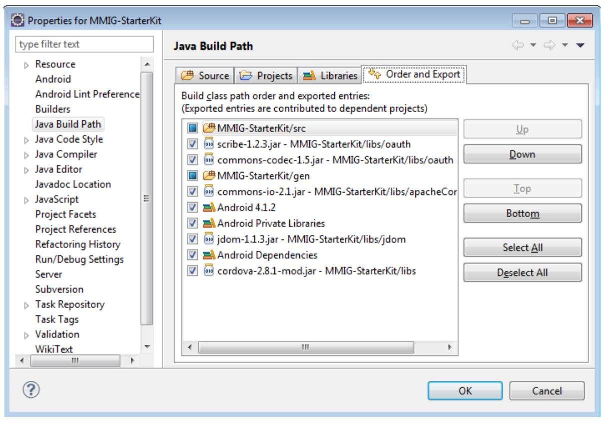
Library

☐ Is Library

Reference	Project
✓ ..\..\..\Users\aaru01\D...	facebook

Figure 11: Selecting the Android version for the project

- Ensure that the correct Java compiler version is used: in the project's **Properties** ► **Java Compiler**, set the **Java compiler compliance level** to 1.5 (5) or 1.6 (6).
- For Android ADT version 22+: If you experience problems that classes cannot be found which should be available by referenced JAR libraries, go through the following steps:
  - ensure that all referenced libraries are located in the project folder `/libs` (or in a subfolder thereof)
  - open the project's **Properties** ► **Java Build Path**
  - select the **Libraries** tab and ensure that all required JAR files are referenced (note that there is an entry **Android Private Libraries** which older ADT versions do not show)
  - Go to the **Order and Export** tab and check all boxes for the referenced JAR libraries (i.e. export them, see Figure 12)



Java Build Path

Build class path order and exported entries:
(Exported entries are contributed to dependent projects)

■ MMIG-StarterKit/src
✓ scribe-1.2.3.jar - MMIG-StarterKit/libs/oauth
✓ commons-codec-1.5.jar - MMIG-StarterKit/libs/oauth
■ MMIG-StarterKit/gen
✓ commons-io-2.1.jar - MMIG-StarterKit/libs/apacheCor
✓ Android 4.1.2
✓ Android Private Libraries
✓ jdom-1.1.3.jar - MMIG-StarterKit/libs/jdom
✓ Android Dependencies
✓ cordova-2.8.1-mod.jar - MMIG-StarterKit/libs

Figure 12: Enable export for all libraries

- If you remove or replace a JAR library in `/libs`, you should *"physically"* first remove the library, rebuild the project, and –in a case of replacement– then add the new library (and rebuild). Simply removing the library in the project's **Properties** ► **Java Build Path** may not be enough to work (in this case you may experience some error messages containing something similar to `Conversion to Dalvik format failed with error 1` in the **LogCat** output console).

Deploy to Simulator

- Right click the project and select **Run As ▶ 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 select **Run As ▶ Android Application**.

Deployment for Web Site

- Open `assets/www/index.html` in a web browser *(see also Notes on compatibility below)*
- Alternatively, you can also host the contents of `assets/www/` on a web server

Notes on Browser **compatibility**:

- Google Chrome has additional restrictions for accessing local files. For Web Site Deployment as local file (i.e. directly opening /index.html in a browser) an additional command-line switch is required in order to allow the MMIR framework access to its local files:
  `--allow-file-access-from-files`
  Note, that
  - This command-line option will only work, if no other instances of Chrome are running, when starting the application with this switch
  - Afterwards, this command-line option applies to all (newly opened) instances of Chrome
  - This option should not be used, when browsing the Internet, but only for development purposes, since it may pose a security risk

Notes on support for **speech** interactions:

- Currently, only Google Chrome (version 25.x.x) supports access to the microphone
  - To enable Chrome for accessing the microphone activate the **Web Audio Input** in `chrome://flags/`
  - 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. 26), you should see one of the screens depicted in Figure 13.

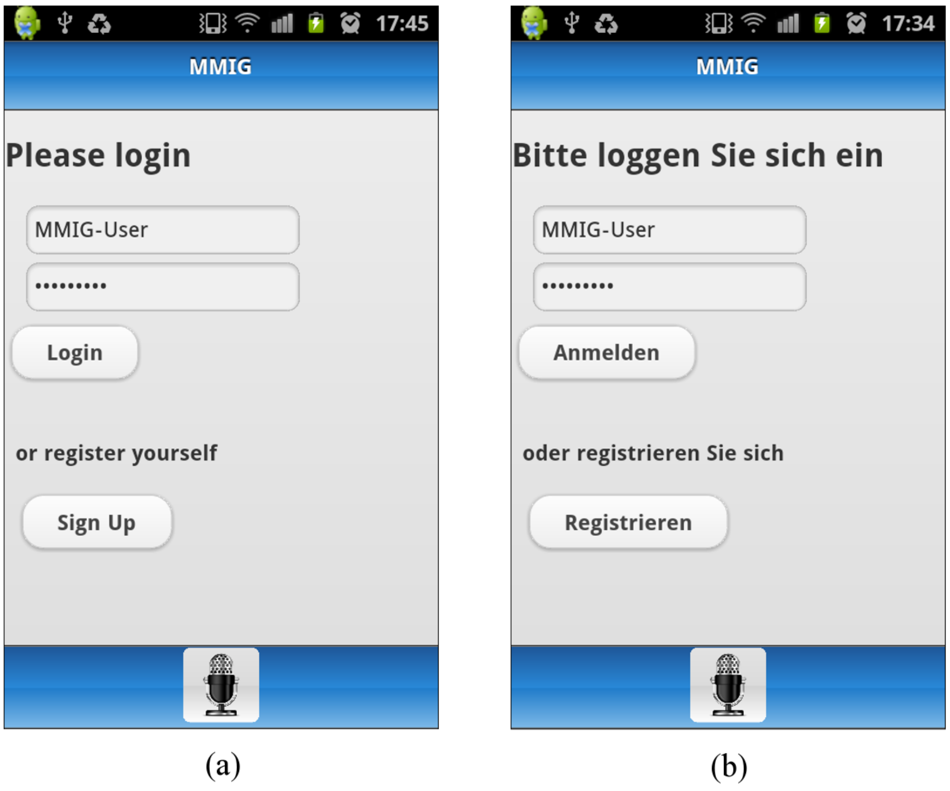
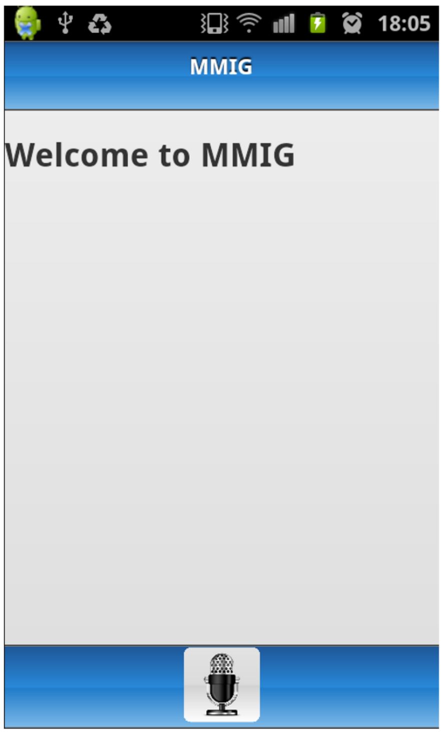
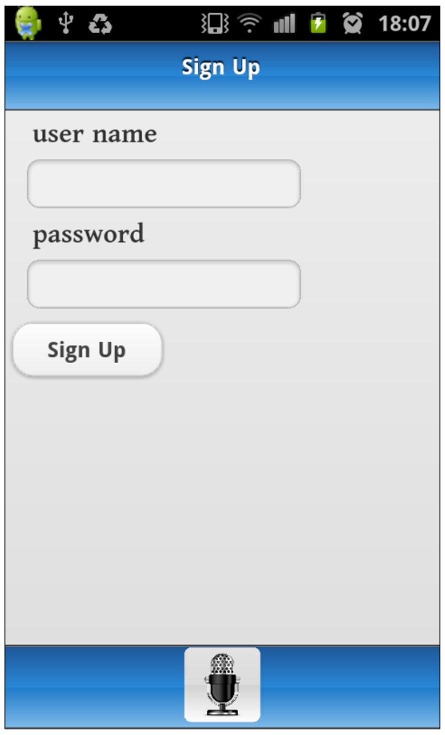


Figure 13: Start screen of MSK. (a) for English (b) for German

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 14 depicts the welcome and sign up screen of MSK.



(a)



(b)

Figure 14: *Welcome* (a) and *sing up* (b) screens of MSK

Building MSK

The MSK project contains an **ANT** `build.xml` file for automating repetitive build tasks.

Building in context of the MMIR framework mainly means, to keep the `directories.json` file up to date – it needs to be updated every time a controller, view, model, or language resource file (e.g. dictionary, grammar) is added or removed. This can be done, using the `build.xml` file of the project.

Build Configuration and Additional Requirements

The configuration file for `build.xml` is `build.settings`: if you do not see the file `build.settings`, either copy `build.settingsDefault` and rename it or try to run build.xml once (and refresh the view in **Eclipse**) which will create `build.settings` if it does not exist yet.[6]

This file only needs to be configured, if

(1) you want to create or modify speech grammars
(in `www/config/languages/[lang]/grammar.json`)

(2) you want to compile `dialogDescriptionSCXML.xml` or `inputDescriptionSCXML.xml`
NOTE: explicitly compiling the SCXML files into JavaScript files is not necessary any more, since the used SCION library interprets the XML files directly – however, compilation can be useful e.g. for finding syntax errors in the SCXML files.

If you do not want to do (1) or (2), no additional configuration is needed.

[6] the file `build.properties` contains general settings for `build.xml`, which do not need to be customized to the specific project environment.

For compiling speech grammars (1), we highly recommend the use (and configuration) for **NodeJS**[7] since this is considerably faster than, e.g. using Mozilla **Rhino**. If you want to use **NodeJS**, you need to edit `build.settings` to set the correct path and executable name to **NodeJS** (after you have downloaded/installed **NodeJS**).

NOTE: If you remove a speech grammar, you also should run the clean task in build.xml (or delete the compiled grammar file manually).

For compiling `dialogDescriptionSCXML.xml` or `inputDescriptionSCXML.xml` (2) you need the Open Source project Apache SCXML Commons JS and Maven; edit `build.settings` and set your paths etc. for Maven and the SCXML Commons JS project.

The MMIR framework knows about all components (e.g. JavaScript implementations, view templates, etc.) that it needs to load by looking into the JSON file `/assets/www/config/directories.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** – this also compiles the grammar definitions (`grammar.json`) into executable grammars; if you want to only update the file-list run the specific target, e.g. by using **Run As ► Ant Build…** and selecting the target `generateFileListJSONFile`.

You should update the information in `directories.json` always after creating for deleting 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/mmirf/plugins/`
-	A dictionary in	`www/config/languages/`
-	A grammar in	`www/config/languages/`
-	A speech config. in	`www/config/languages/`

[7] see http://nodejs.org/

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 into `calendar.js`

```
//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:

```
Calendar.prototype.create_appointment = function (ctrl, data){
  var container_id = data.container_id;
  var container = $("#" + container_id);
  var subject = $("#subject", container).val();

  var enteredDate = $("#app-date", container).datebox('getTheDate');
  var year = enteredDate.getFullYear();
  var month = enteredDate.getMonth() + 1;
  var day = enteredDate.getDate();

  var startTime = $("#start-time", container).datebox('getTheDate');
  var start_h = startTime.getHours();
  var start_m = startTime.getMinutes();

  var endTime = $("#end-time", container).datebox('getTheDate');
  var end_h = endTime.getHours();
  var end_m = endTime.getMinutes();

  var note = $("#note", container).val();
  if(typeof note !== 'string'){
        note = '';
  }
  else {
        note = note.escapeDoubleQuotes()
                   .replaceAll('\r\n','\\r\\n').replaceAll('\n','\\n');
  }

  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 +
                  '","note":"' + note + '"}';

  var jData = jQuery.parseJSON(eventData);
  var cb_func = function(){
    alert("STUB: appointment successfully created!\n\n"
             +JSON.stringify(jData, null, 2)
    );
  };
  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 to add a new `.ehtml` file to its view folder (`www/views/<controller name>`). For example, if we want to add a new view for application controller, which will be shown when the registration of a new user failed, we would add a new file named `reg_failed.ehtml` to `www/views/application` which would contain the view definition.

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 `directories.json` in `assets/www/config/`.

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 first letter). We name the new model `calendarModel.js` which should contain the following code:

```
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 `directories.json` in `assets/www/config/` by invoking the default Ant task `generateFileListJSONFile` in `/build.xml` (see chapter Add views of a new controller, p. 43).

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 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/dialogDescriptionSCXML.xml`. This description is automatically loaded and utilized via the `DialogEngine`: The dialog engine handles events and dispatches them to controllers' actions and/or renders controllers' views based on transition specified in dialog description.

Besides the dialog description, there is also `www/config/statedef/inputDescriptionSCXML.xml` which is utilized by the `InputEngine`. The `InputEngine` is meant to receive the *"raw"* input events from different modalities/input devices (e.g. touch input, speech input, gesture input), and then relaying *"meaningful application events"* to the `DialogEngine` (e.g. by *modality fusion*). For instance, the InputEngine could process the speech event "`show me information`" and the touch event "`select(obj1)`", and map dispatch an *application event* "`show information about obj1`".

It is important to note, that the SCXML files are application-specific. The SCXML files of the MSK are only an example of how to use these, specifically in context of this example application (i.e. the StarerKit).

Touch input events for MSK will first raise an event on the `InputEngine` that signal the modality (`touch_input_event`), and then raise the specific event with additional data: which object/element was touched and other data if necessary.

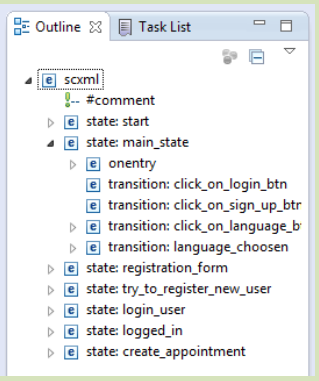
For speech input events in MSK, the modality and data is transferred to the `InuputEngine` with one single event using `speech_input_event`.

Figure 15 shows the dialog description of MSK. This description is depicted in Figure 17 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="click_on_login_btn" target="login_user" />
    <transition event="click_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="click_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="click_on_appointment_btn" target="create_appointment" />
    <transition event="back" name="back" target="main_state" />
  </state>

</scxml>
```

Figure 15: Dialog Description of MSK

The `name` attributes in `state`-, `transition`- ect. tags are non-functional, i.e. can be safely omitted. However, if you use Eclipse, its XML editor will use these `name` attributes in the outline view (usually displayed on the right), which can proof useful when navigating within the file.



```
Outline ⊠   Task List
◢ e scxml
    #comment
  ▷ e state: start
  ◢ e state: main_state
    ▷ e onentry
      e transition: click_on_login_btn
      e transition: click_on_sign_up_btr
    ▷ e transition: click_on_language_b
    ▷ e transition: language_choosen
  ▷ e state: registration_form
  ▷ e state: try_to_register_new_user
  ▷ e state: login_user
  ▷ e state: logged_in
  ▷ e state: create_appointment
```

Figure 16: Outline view for SCXML file with name attributes in STATE and TRANSITION tags

NOTE that Eclipse may have several XML editors installed. If there is no outline view available when you open an SCXML file, you may need to select a specific XML editor, e.g. by opening the context menu for the file, and then **Open With ▶ Android Common XML Editor** (or **Open With ▶ Other…** and select an appropriate editor in the **Editor Selection** dialog that opens).

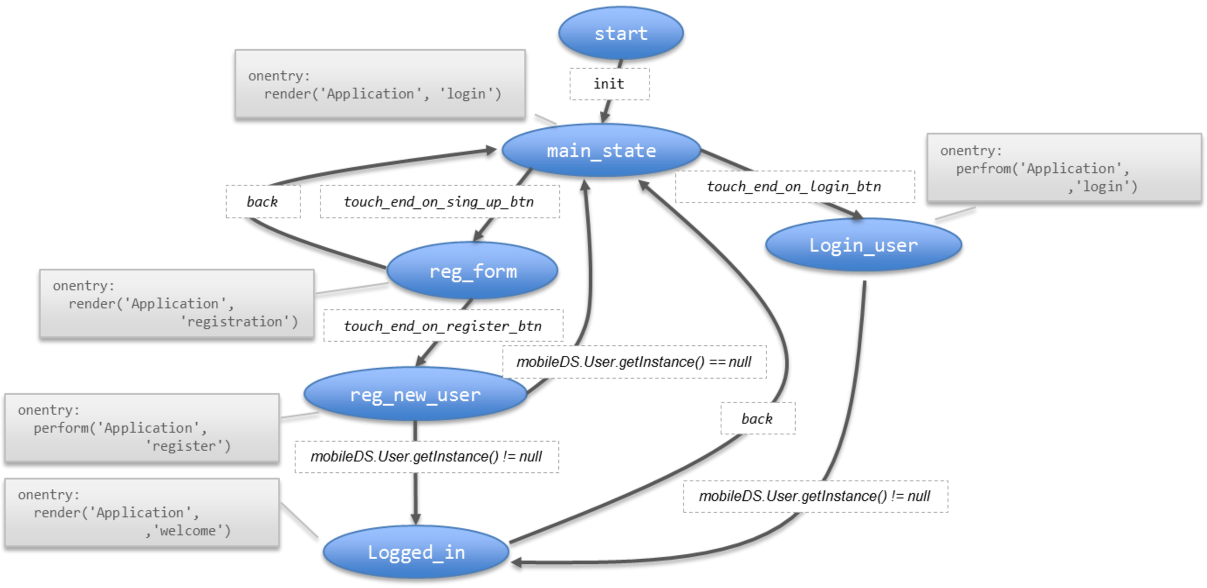


Figure 17: Dialog Description of MSK as FSA

Expanding MSK with models/views/controllers alone does not extend the behavior of your application. For this, you also have to extend the dialog description of MSK. Lets start by adding some modifications in the `welcome` view of the application controller, that will create the GUI element. As you have seen in Figure 14, after logging in, there is no more interaction available to 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>
}@
```

Each HTML button element in the MSK view descriptions MUST have a unique name.

For HTML button elements, MSK automatically raised the following events (synchronously)

- `touch_start_on_name`
- `touch_end_on_name`
- `click_on_name`

where *name* is the name of pressed button. For example by pressing the appointment button in the `welcome` view, MMIR would first raise `touch_start_on_appointment_btn`, then `touch_end_on_appointment_btn`, and lastly `click_on_appointment_btn`.

This mechanism can be specified by setting a callback function via the `setOnPageRenderedHandler` function in `DialogEngine`. In MSK this callback function is specified by the function `exectueAfterEachPageIsLoaded` in `mmirf/main.js` (which is passed to `set_on_page_loaded` during the initialization process).

Next, we need to extend the `config/statedef/inputDescriptionSCXML.xml` file, so that it can handle the touch event of the newly created button. Note that by default the added button will trigger 2 events on the InputEngine for each 'touch_start', 'touch_end', and 'click'-event: first an event `touch_input_event`, and then the corresponding event for the button, i.e. in this case `touch_start_on_appointment_btn`, `touch_end_on_appointment_btn`, and `click_on_appointment_btn`.

When we look at the `inputDescriptionSCXML.xml` we can see, that the first `touch_input_event` will bring the `InputEngine`'s state machine into the `touch_input` state, which in turn automatically redirects to its inner state `start_touch`:

```
  <transition event="touch_input_event" traget="touch_input" />

…

  <state id="touch_input" initial="start_touch" name="touch_input">
     <state id="start_touch" name="start_touch">
…
```

For extending the inputDescription to handle the button's event, we first add a transition for the `start_touch` state, that will be triggered when the click-event off button is fired:

```
<state id="start_touch" name="start_touch">
…
   <transition      event="click_on_save_appointment_btn"
                    target="save_appointment"></transition>
```

Then we create the transition's target state `appointment`, within the `touch_input` state, which finally handles the event: on entering the `appointment` state, the "application event" for the button will be raised on the `DialogEngine`:

```
<state id="touch_input" initial="start_touch" name="touch_input">
  <state id="start_touch" name="start_touch">
…
  </state>
…
  <state id="appointment" name="appointment">
    <onentry>
     <script>
       mobileDS.DialogEngine.getInstance().raise(
           'click_on_appointment_btn'
       );
     </script>
    </onentry>
   </state>
…
```

Now we have to handle the new event in the `dialogDescription`. For this, we add a transition in our dialog description which will be triggered after raising the `click_on_appointment_btn` event. To do this, we first open the `config/statedef/dialogDescriptionSCXML.xml` file and find the `logged_in` state, and the add the following transition to it:

```
<transition event="click_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.

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

  <transition event="click_on_save_appointment_btn" target="save_appointment" />
  <transition event="click_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 `InputEngine`. 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 `InputEngine`, e.g. `click_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/inputDescriptionSCXML.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. Figure 18 shows the new screens of calendar application.

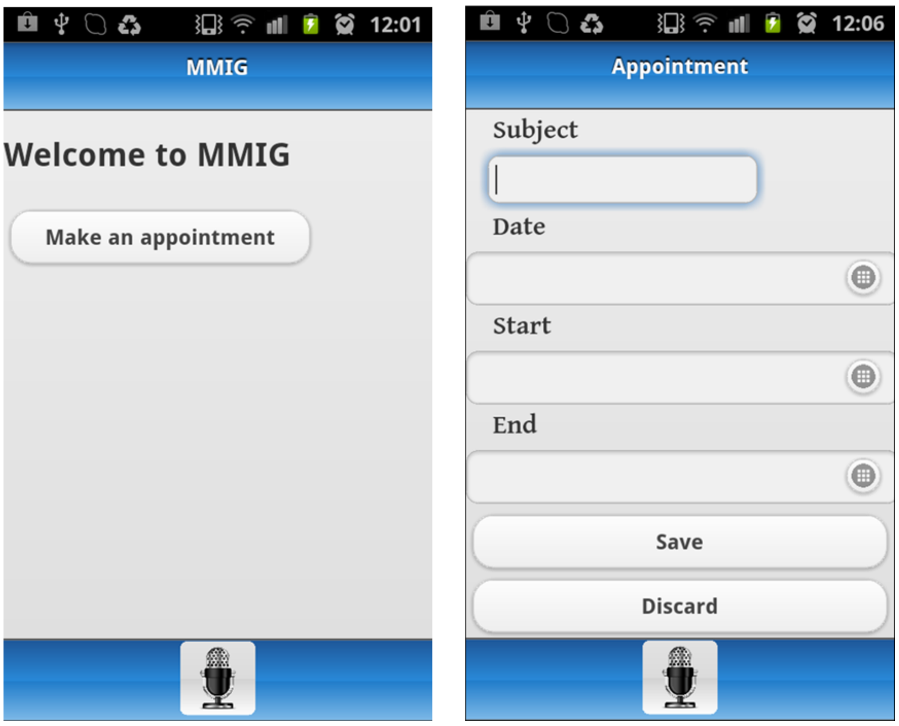


Figure 18

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`.