User's Handbook

map.apps with MD2

Project Seminar

Model-driven Mobile Development

(University of Münster)

12th March 2015

Contents

1 Introduction

MD^2 is a framework for model-driven development of mobile business applications. It provides a domain specific language (DSL) for the specification of a textual model. Such a model can describe characteristics of a business scenario, including the communication between different apps and sequences of actions within each app, the views to be displayed and the data to be stored. A complete DSL model consists of four files, each representing a certain perspective - model, view, controller as well as workflow - on the application(s) to be generated.

In addition to the DSL, the MD^2 framework comprises a generator, which uses models specified in the DSL to generate source code. The code generation process creates source code for all apps that are specified in the model as well as a backend that handles their communication via web services as well as the communication with external applications. Originally, the MD^2 framework supported code generation for Android and IOS, but now includes a generator for a JavaScript-based web-platform called map.apps. Updates for Android and IOS are intended in the future.

This handbook comprises a modeler's handbook and a developer's handbook. The modeler's handbook is targeted towards people who want to use the MD^2 framework to create models and generate code from them. It describes the constructs provided by the DSL and how to use them. Furthermore, it explains how the applications generated by the framework can be deployed. The developer's handbook aims to provide all information that is necessary for further development of and with the MD^2 framework. This includes the structure of the code generator for map.apps as well as the structure and interaction of the generated and static code.

2 Modeler's Handbook

2.1 Installation

The following software is required prior to the next steps:

- Eclipse IDE for Java and DSL Developers (e.g. Version Luna)
  - Xtend und Xtext v.2.7.3
  - GlassFish Tools for Luna
- GlassFish 4.+
- map.apps 3.1.0
- NetBeans EE (e.g. Version 8.0)
- Apache Tomcat 7.0 with running map.apps runtime

Umstellung auf den neuen Glasfish, Luna usw. (vgl. Facebook) bitte kontrollieren. Fehlt etwas?

2.2 Getting Started

2.2.1 Developing a Single App

This section describes how an application can be developed based on the current (March 2015) state of the MD^2 DSL. To this end, it is structured according to the different layers that constitute a MD^2 model, which are shown in Figure 2.1. Aside from the conventional MVC layers – model, view and controller – the architecture includes an additional layer which enables and models the generation of workflows within and across apps and was added to MD^2 during the project seminar.

All components in MD^2 are organized in a package structure that represents the aforementioned structure. All documents have to be placed in corresponding packages (views, models, controllers or workflow). For example, all view files are expected to be in the package any.project.package.views. The package has to be defined in each MD^2 file as follows:

```
package PACKAGE_NAME
```

The package name has to be a fully qualified name that reflects the actual folder structure.

Den MapApps installation Guide hier einbinden oder im Anhang referenzieren?

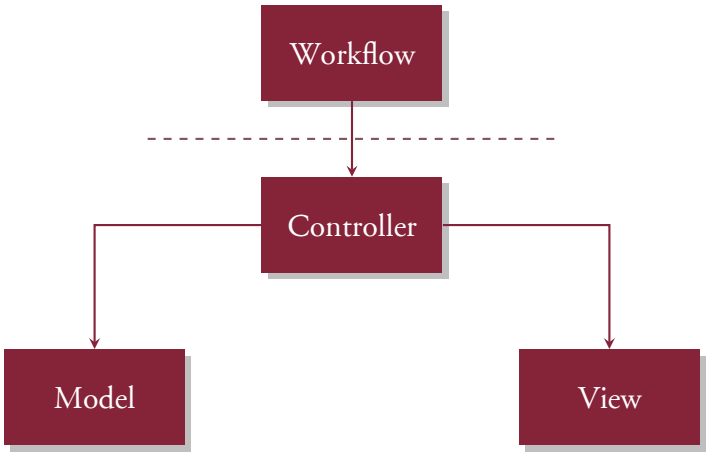

Figure 2.1: Architecture of MD^2 Models

Workflow

The workflow layer provides abstraction on top of the controller layer. It allows to specify the general course of action of one or more apps using a few simple and easily understandable model language constructs. Furthermore, this abstract workflow representation is intended to serve as a basis for communication with customers, e.g. for requirements engineering and collaborative app development through rapid prototyping.

Workflows are specified as a (possibly cyclic) directed graph of workflow elements in the workflow file of an MD^2 model. Workflow elements represent encapsulated functionality which needs to be further specified in the controller layer. The workflow layer merely references the workflow elements from the controller layer to define their interaction.

Workflow elements are linked to each other via events. For each workflow element one or more events can be specified that can be fired. However, at runtime a workflow element can fire only one of these events, i. e. a parallel processing of the workflow is not intended. Similar to the workflow elements, workflow events in the workflow layer are references to workflow events that are created in the respective controller.

In addition to the events that can be fired, the workflow element also specifies which workflow element is to be started in response to a fired event using the keyword `start`. Moreover, when an event is fired, not only workflow elements can be started but the workflow can be terminated by using the `end workflow` keyword. A workflow element in the workflow layer typically looks as shown in Listing 2.1.

Listing 2.1: Workflow Elements in the Workflow Layer

```
WorkflowElement <NameOfWorkflowElement>
      fires <NameOfEventOne> {
            start <NameOfSubsequentWfeOne>
      }
      fires <NameOfEventTwo> {
```

```
            end workflow
      }
```

After defining the sequence of workflow elements, the workflow also requires the specification of an app. As shown in Listing 2.2, an app consists of its ID, a list of workflow elements that are used in the app and a name that is to be used as the app title. In the scenario where only a single app is modeled, all workflow elements can be listed in the app. However, it is also possible to have unused workflow elements.

Listing 2.2: App Definition in MD^2

```
App <AppID> {
      WorkflowElements {
            <WorkflowElementOne>,
            <WorkflowElementTwo> (startable: STRING),
            <WorkflowElementThree>
      }
      appName STRING
}
```

A workflow has one or more entry points, i. e. startable workflow elements. These are marked as `startable` in the app specification. During code generation this will result in a button on the app's start screen that starts the corresponding workflow element. In addition, an alias needs to be provided which is used as label or description for the button.

Finally, the complete workflow specification for one app will be structured as shown in Listing 2.3. Note that MD^2 does not differentiate between different workflows. However, it is possible to implicitly create multiple workflows by using two or more startable workflow elements that start independent, disjunct sequences of workflow elements.

Listing 2.3: Workflow Definition in MD^2

```
package <ProjectName>.workflows

WorkflowElement <WorkflowElementOne>
[...]
WorkflowElement <WorkflowElementTwo>
[...]
WorkflowElement <WorkflowElementThree>
[...]

App <AppID> {
      [...]
}
```

Model

In the model layer the structure of data objects is being described. As model elements Entities and Enums are supported.

Entity An entity is indicated by the keyword `entity` followed by an arbitrary name that identifies it.

```
entity NAME {
<attribute1 ... attribute n>
}
```

Each entity may contain an arbitrary number of attributes of the form

```
ATTRIBUTE_NAME: <datatype>[] (<parameters>) {
name STRING
description STRING
}
```

The optional square brackets [] indicate a one-to-many relationship. That means that the corresponding object may hold an arbitrary number of values of the given datatype. Supported complex data types are:

- Entity
- Enum

Supported simple data types are:

- `integer` – integer
- `float` – float of the form #.#
- `boolean` – boolean
- `string` – a string that is embraced by single quotes (') or double quotes (")
- `date` – a date is a string that conforms the following format: `YYYY-MM-DD`
- `time` – a time is a string that conforms the following format: `hh:mm:ss[(+|-)hh[:mm]]`
- `datetime` – a date time is a string that conforms the following format: `YYYY-MM-DDThh:mm:ss [(+|-)hh[:mm]]`

Parameters are optional and will be transformed into implicit validators during the generation process. They have to be specified as a comma-separated list. On default each specified attribute is mandatory. To allow null values the parameter optional can be set. Further supported parameters depend on the used data type and are explained as follows:

- `integer` supports
  - `max INTEGER` – maximum allowed value of the attribute
  - `min INTEGER` – minimum allowed value of the attribute
- `float` supports
  - `max FLOAT` – maximum allowed value of the attribute

`min` `FLOAT` – minimum allowed value of the attribute

- `string` supports

  `maxLength` `INTEGER` – maximal length of the string value

  `minLength` `INTEGER` – minimal length of the string value

Optionally, attributes can be annotated with a name and a description which are used for the labels and the tooltips in the auto-generation of views. If a tooltip is annotated a question mark will be shown next to the generated input field. If no name is annotated, a standard text for the label will be derived from the attribute's name by transforming the camel case name to natural language. E.g. the implicit label text of the attribute firstName is "First name".

Exemplary entity that represents a person:

```
entity Person {
      name: string
      birthdate: date {
            name: "Date of Birth"
            description: "The exact day of birth of this person."
      }
      salary: float (optional, min 8.50, max 1000)
      addresses: Address[]
}
```

Enum An enumeration is indicated by the keyword `enum` followed by an arbitrary name that identifies it. Each enum may contain an arbitrary number of comma-separated strings. Other data types are not supported. Exemplary enum element to specify weekdays:

```
enum Weekday {
"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
}
```

View

View elements are either ContentElements or ContainerElements that can contain other content or container elements. Furthermore, basic styles for some content elements can be defined.

Container Elements *Grid layouts* align all containing elements in a grid. Elements can either be containers or content elements. The grid is populated row-by-row beginning in the top-leftmost cell.

```
GridLayoutPane NAME (<parameters>) {
      <Container | Content | [Container] | [Content]>
      <Container | Content | [Container] | [Content]><...>
```

```
}
```

For each grid layout at least the number of rows or the number of columns has to be specified. If only one of these parameters is given, the other is calculated by MD^2 on generation time. In case that both parameters are specified and there are too few cells, all elements that do not fit in the layout will be discarded. The following comma-separated parameters are supported:

- `columns INTEGER` – the number of columns of the grid
- `rows INTEGER` – the number of rows of the grid
- `tabIcon PATH` – if the layout is a direct child of a TabbedPane, an icon can be specified that is displayed on the corresponding tab. See section on TabbedPanes for more details.
- `tabTitle STRING` – if the layout is a direct child of a TabbedPane, a text can be specified that is displayed on the corresponding tab. See section on TabbedPanes for more details.

A *flow layout* arranges elements (containers or content elements) either horizontally or vertically. By default all elements are arranged in a left-to-right flow.

```
FlowLayoutPane NAME (<parameters>) {
      <Container | Content | [Container] | [Content]>
      <Container | Content | [Container] | [Content]>
      <...>
}
```

The following comma-separated parameters are supported:

- `vertical` or `horizontal` (default) – flow direction
- `tabIcon PATH` – if the layout is a direct child of a TabbedPane (which are described subsequently), an icon can be specified that is displayed on the corresponding tab.
- `tabTitle STRING` – if the layout is a direct child of a TabbedPane, a text can be specified that is displayed on the corresponding tab.

A *tabbed pane* is a special container element that can only contain container elements. Each contained container will be generated as a separate tab. Due to restrictions on the target platforms, tabbed panes can only be root panes, but not a child of another container element. By default the title of each tab equals the name of the contained containers. By using the tabTitle and tabIcon parameters the appearance of the tabs can be customized.

```
TabbedPane NAME {
      <Container | [Container]>
      <Container | [Container]>
      <...>
}
```

Content Elements *Input elements* can be used to manipulate model data via mappings (see Section 2.2.1). At the moment BooleanInputs, TextInputs, IntegerInputs, NumberInputs, TimeInputs, DateInputs, DateTimeInputs and OptionInputs are supported. All input elements support the optional attributes `label`, `tooltip`, `type`, `disabled`, `default` and `width`. `type` allows to specify the type of the field. `disabled` was intended to provide a boolean for whether an input field is enabled od disabled, but is not regarded by the generator in the current implementation. Default values can be set using `default` and the width using `width`.

TextInput fields can be used for freetext as well as date and time inputs.

```
TextInput <name> {
      label STRING
      tooltip STRING
      type <TextInputType>
      disabled <true|false>
      default STRING)
      width <width>
}
```

OptionInputs are used to represent enumeration fields in the model. In addition to the aforementioned attributes, OptionInputs support the optional `options` attribute. This can be used to populate the input with the string values of the specified Enum. If options is not given, the displayed options depend on the Enum type of the attribute that has been mapped on the input field (see Section 2.2.1).

```
OptionInput {
      label STRING
      tooltip STRING
      type <OptionInputType>
      disabled <true|false>
      default STRING)
      width <width>
      options <enum>
}
```

Labels allow the modeler to present text to the user. Often they are used to denote input elements. For the label definition there exists the following default definition

```
Label NAME {
      text STRING
      style <[style] | style>
}
```

as well as this shorthand definition

```
Label NAME (STRING){
      style <[style] | style>
}.
```

The text can either be annotated as an explicit text attribute or the label text to display can be noted in parentheses directly after the label definition. The optional style can either be noted directly or an existing style definition can be referenced (styles are described later in this section).

Tooltips allow the modeler to provide the user with additional information. For the tooltip definition there exists the following default definition

```
Tooltip NAME {
      text STRING
}
```

as well as this shorthand definition that allows to note the help text in parentheses directly after the label definition

```
Tooltip NAME (STRING).
```

Buttons provide the user the possibility to call actions that have been bound on events of the Button. For the button definition there exists the following default definition

```
Button NAME {
      text STRING
}
```

as well as this shorthand definition that allows to specify the text in parentheses directly after the button definition

```
Button NAME (STRING).
```

For the *image* exists the following default definition

```
Image NAME {
      src PATH
      height INT
      width INT
}
```

as well as this shorthand definition that allows to specify the image path in parentheses directly after the image name

```
Image NAME (PATH) {
      height INT
      width INT
```

}.

Images support the following attributes:

- `src` – Specifies the source path where the image is located. The path has to be relative to the directory /resources/images in the folder of the MD^2 project
- `height` (optional) – Height of the image in pixels
- `width` (optional) – Width of the image in pixels

A *Spacer* is used in a `GridLayoutPane` to mark an empty cell or in a `FlowLayoutPane` to occupy some space. Using an optional additional parameter the actual number of spacers can be specified.

```
Spacer (INT)
```

The *AutoGenerator* is used to automatically generate view elements to display all attributes of a related entity and the according mappings of the view elements to a Content Provider. It is possible to either exclude attributes using the `exclude` keyword or to provide a positive list of attributes using the keyword `only`.

```
AutoGenerator NAME {
      contentProvider [ContentProvider] (exclude|only [Attribute])
}
```

In case of one-to-many relationships for attributes (annotated with []) or a content provider it has to be defined which of the elements should be displayed in the auto-generated fields. The *EntitySelector* allows the user to select an element from a list of elements. The attribute `textProposition` defines which ContentProvider stores the list and which attribute of the elements shall be displayed to the user to allow him to find the desired element.

```
EntitySelector NAME {
      textProposition [ContentProvider.Attribute]
}
```

Styles can be annotated to several view elements such as labels and buttons to influence their design. They can either be defined globally as a root element in the view and then be referenced or annotated directly to the appropriate elements.

```
style NAME {
      color <color>
      fontSize INT
      textStyle <textstyle>
}
```

The following optional style attributes are supported. If a attribute is not set, the standard setting is used for each platform.

- `color <color>` (optional) – specifies the color of the element as a named color or a six or eight digit hex color (with alpha channel)
- `fontSize INT` (optional) – specifies the font size
- `textStyle <textstyle>` (optional) – the text style can be normal or italic, bold or a combination of both.

As named colors the 16 default web colors are supported: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, yellow.

Elements can not only be defined where they should be used, but there is also a mechanism of defining an element once and *reusing* it several times. Instead of defining a new element, another element can be referenced – internally this leads to a copy of the actual element. However, names have to be unique so that each element could only be referenced once. To avoid those name clashes a renaming mechanism had been implemented that allows to set new names for the actual copied element.

```
Element -> NAME
```

Controller

Main The `main` object contains all basic information about a project. Each project must contain exactly one `main` object that can be in an arbitrary controller.

```
main {
      appVersion STRING
      modelVersion STRING
      workflowManager [WorkflowBackendConnection]\todo{what exactly needs to be
         written inside the brackets?}
      defaultConnection [RemoteConnection]
}
```

The attributes are explained as follows:

- `appVersion` – a string representation of the current app version, e.g. “RC1”
- `modelVersion` (optional) – a string representation of the current model version that has to be in accordance with the model version of the backend
- `defaultConnection` (optional) – a default remote connection can be specified here, so that it is not necessary to specify the same connection in each content provider
- `workflowManager` –

describe workflow-Manager

Furthermore, the controller layer is subdivided into one or more workflow elements. While the workflow describes the interaction of workflow elements, their internal functionality needs to be specified in the controller layer. Each workflow element can be seen as an independent controller which is responsible for the successful execution of its functionality. In general, workflow elements are structured as follows

```
WorkflowElement <workflowElementName>{
      defaultProcessChain <ProcessChain>
      onInit {
            <...>
      }
      <...>
}.
```

Workflow elements work with `ProcessChains` that define a sequence of possible steps inside the workflow element and are further described in Section 2.2.1. Since several process chains can be defined in a single workflow element, the `defaultProcessChain` keyword is used to set a default process chain. This process chain will then be used as starting point for the workflow element and the first view to be shown will also be derived from the default process chain.

The `onInit` block is used to define everything that is supposed to happen on initialization of the workflow element, e.g. binding actions to buttons or mapping content to view fields. In the `onInit` block, no workflow events may be fired, as this means handing off control from the workflow element directly at the initialization. This is enforced by raising a validator error in case the modeler tries to do this.

Typically, workflow elements should be modeled in a way that they fire a workflow event after termination of their functionality to start a new workflow element or end the whole workflow. This can be done using the SimpleAction `FireEventAction` as will be described in the following.

Actions An action provides the user the possibility to declare a set of tasks. An action can be either a `CustomAction` or a `CombinedAction`.

A *CustomAction* contains a list of CustomCodeFragments where each CustomCodeFragment contains one task.

```
CustomAction <Action> {
      <CustomCodeFragment>
      <...>
}
```

For each type of task there exist a specific CustomCodeFragment that is distinguished by the keyword that introduces it.

- `bind <Action1> ... <ActionN> on <Event1> ... <EventN>`
- `bind <Validator1> ... <ValidatorN> on <ViewElement1> ... <ViewelementN>`
- `unbind <Action1> ... <ActionN> from <Event1> ... <EventN>`
- `unbind <Validator1> ... <ValidatorN> from <ViewElement1> ... <ViewelementN>`
- `call <Action>`
- `map <ViewElement> to <ContentProviderField>`

- `unmap <ViewElement> from <ContentProviderField>`
- `set <ContentProvider> = <Expression>`
- `set <ViewElement> = <Expression>`
- `if (<Condition>){ <CustomCodeFragment> }`
- `elseif (<Condition>){ <CustomCodeFragment> }`
- `else {<CustomCodeFragment>}`

The main tasks are binding actions to events, binding validators to view elements and mapping view elements to model elements. For every task there is a counterpart for unbinding and unmapping. `call` tasks can call other actions, `set` operations set the value of a content provider field or a view element. The `if`, `elseif` and `else` blocks allow to model case distictions, e.g. based on user input.

Actions are bound to events. There are several types of actions and events available. CustomActions and CombinedActions are referenced externally whereas SimpleActions are declared directly. For events, there are local event types that listen to the state of a certain view element as well as global event types. The most powerful event type is the OnConditionEvent.

SimpleActions provide a quick way to change the state of the project:

- ProcessChainProceedAction: `ProcessChainProceed`
  proceed to the next ProcessChain step
- ProcessChainReverseAction: `ProcessChainReverse`
  go back to the last ProcessChain step
- ProcessChainGotoAction: `ProcessChainGoto <processChainStep>`
  change to the given ProcessChain step
- SetProcessChainAction: `SetProcessChain [ProcessChain]`
  changes the current ProcessChain
- GotoViewAction: `GotoView (<ViewElement>)`
  change to the given view element
- DisableAction: `Disable (<ViewElement>)`
  disables a view element
- EnableAction: `Enable (<ViewElement>)`
  enables a view element
- DisplayMessageAction: `DisplayMessage (<SimpleExpression>)`
  displays a message
- ContentProviderOperationAction: `ContentProviderOperation (<AllowedOperation> <ContentProvider>)`
  perform a CRUD action (save, load, remove) on the given ContentProvider
- ContentProviderResetAction: `ContentProviderReset (<ContentProvider>)`
  resets the given ContentProvider

- FireEventAction: `FireEvent (<WorkflowEvent>)`

  fires a workflow event to the backend. In response a new workflow element will be started or the workflow terminated.
- WebServiceCallAction: `WebServiceCall (<WebServiceCall>)`

  sends a request to call an external web service to the backend (for details cf. section 2.4.2).
- In addition, the LocationAction allows to extract an address from content provider fields and to generate a punctual location (i.e. longitude and latitude) for this address. For this purpose, all input and output fields have to be defined. The calculation result for longitude and latitude will be written in their respective output fields. The LocationAction is structured as follows:

```
Location (
      inputs (
            cityInput <ContentProviderPath>
            streetInput <ContentProviderPath>
            streetNumberInput <ContentProviderPath>
            postalInput <ContentProviderPath>
            countryInput <ContentProviderPath> )
      outputs (
            latitudeOutput <ContentProviderPath>
            longitudeOutput <ContentProviderPath>).
```

There are different event types available:

- `ElementEventType` – onTouch, onLeftSwipe, onRightSwipe, onWrongValidation; preceded by a dot and a reference to a ContainerElement or ContentElement
- `GlobalEventType` – onConnectionLost
- `OnConditionEvent`

The *OnConditionEvent* provides the user the possibility to define own events via Conditions. The event is fired when the conditional expression evaluates to true.

```
event NAME {
      <Condition>
}
```

A *condition* allows to combine conditional expressions using the operators `and`, `or`, and `not`. Conditional expressions evaluate to true or false. They can be BooleanExpressions, EqualsExpressions or GUIElementStateExpressions that check the state of a ViewGUIElement. In addition, MD^2 supports mathematical expressions such as `equals`, `>`, `<`, `>=`, and `<=`.

Validators are bound to view elements. The validator can be a referenced element or a shorthand definition can be used in place.

```
bind|unbind validator
```

```
      <[Validator]> <...>
      on|from
      <[ContainerElement] | [ContentElement]> <...>
```

The shorthand definition has the same options but does not allow reuse.

```
bind|unbind validator
      <IsIntValidator | NotNullValidator | IsNumberValidator | IsDateValidator
         | RegExValidator |NumberRangeValidator | StringRangeValidator
         (<params>)
      on|from
      <[ContainerElement] | [ContentElement]> <...>
```

A detailed description for validator type can be found in the validator description in the following. The available parameters at params are identical to those of the validator element.

View elements are *mapped* to model elements that are in turn accessed through a ContentProvider.

```
map|unmap
      <[ContainerElement] | [ContentElement]>
      to|from
      <[ContentProvider.Attribute]>
```

CallTasks call a different Action.

```
call
      <[CustomAction] | [CombinedAction] | [SimpleAction]>
```

CombinedActions allow the composition of Actions.

```
CombinedAction NAME {
      actions <Action> <...>
}
```

Validators are used to validate user input. For each validator type corresponding parameters can be assigned. The `message` parameter is valid for every type and will be shown to the user if the validation fails.

The RegExValidator allows the definition of a regular expression that is used to validate the user input.

```
validator RegExValidator NAME (message STRING regEx STRING)
```

The IsIntValidator checks whether the user input is a valid integer.

```
validator IsIntValidator NAME (message STRING)
```

The IsNumberValidator checks whether the user input is a valid integer or float value.

```
validator IsNumberValidator NAME (message STRING)
```

The IsDateValidator allows to define a format that the date at hand shall conform to.

```
validator IsDateValidator NAME (message STRING format STRING)
```

The NumberRangeValidator allows the definition of a numeric range that shall contain the user input.

```
validator NumberRangeValidator NAME (message STRING min FLOAT max FLOAT )
```

The StringRange allows the definition of a string length range. The length of the `STRING` input by the user will be checked against this range.

```
validator StringRangeValidator NAME (message STRING minLength INT maxLength INT)
```

The NotNullValidator makes the input field required.

```
validator NotNullValidator NAME (message STRING)
```

The *RemoteValidator* allows to use a validator offered by the backend server. By default only the content and id of the field on which the RemoteValidator has been assigned are transmitted to the backend server. However, additional information can be provided using the `provideModel` or `provideAttributes` keyword.

```
validator RemoteValidator NAME (message STRING connection <RemoteConnection>
    model <ContentProvider>) |
validator RemoteValidator NAME (message STRING connection <RemoteConnection>
    attributes <ContentProvider.Attribute> <...>)
```

ProcessChain A ProcessChain is used to define several steps in which the WorkflowElement can currently be. It is possible to define several ProcessChains. ProcessChains can be nested and there is at most one ProcessChain active.

```
ProcessChain NAME {
      <ProcessChainStep> <...>
}
```

Each ProcessChainStep defines one view that is related to it and will be displayed if the ProcessChainStep becomes the current ProcessChainStep of the active ProcessChain. Additionally conditions can be defined, that restrict switching to the next or previous ProcessChainStep. Also events can be specified that trigger the change to the next or previous ProcessChainStep.

Instead of the aforementioned settings, a ProcessChain that will become active while this ProcessChainStep is the current one, can be referenced.

```
step NAME:
      view <[ContainerElement] | [ContentElement]>
```

```
      proceed {
            on <Event>
            given <Condition>
            do <Action>
      }
      reverse on <Event>
      goto <ProcessChainStep> (returnTo <ProcessChainStep>) on <Event>
      return on <Event>
      return and proceed on <Event>
      message STRING
```

ProcessChains can be refined using SubProcessChains.

```
step NAME:
      subProcessChain <ProcessChain>
```

The event definition for `EventDef` is the same as for event bindings:

```
<Container | Content> . <elementEventType> |
<GlobalEventType> |
<OnConditionEvent> <...>
```

Each *content provider* manages one instance of an entity. View fields are not mapped directly to a model element, but only content providers can be mapped to view elements. Data instances of the content providers can be updated or persisted using DataActions.

It allows to `CREATE_OR_UPDATE` (save), `READ` (load) and `DELETE` (remove) the stored instance. Which of those operations is possible is specified in `allowedOperations`. By default all operations are allowed. A filter enables to query a subset of all saved instances. The `providerType` defines whether the instances shall be stored locally or remotely.

The *remote connection* allows to specify a URI for the backend communication. The backend must comply with the MD2 web service interface as specified in the appendix.

```
remoteConnection NAME {
      uri URI
}
```

Furthermore, the MD^2 framework also provides a *location provider*, i.e. a virtual content provider for locations. The entity which is automatically handled by this content provider contains attributes such as latitude, longitude, street, etc. In map.apps the location provider can be used to get coordinates from a map and resolve the corresponding address.

2.2.2 Deploying a Single App

Backend

map.apps

2.3 Development and Deployment of Multiple Apps

The MD^2 framework also allows to model and generate workflows that involve multiple apps. For this purpose, several apps rather than just one can be specified in the workflow layer. These apps can share the same workflow elements or use different ones as shown in Listing 2.4. Apart from that, the workflow will look as usual, with the sequence of workflow elements being determined via events. After generation, each app is provided with a list of open issues in the start screen. In this list, all events are presented that were fired from another app and are supposed to start a workflow element which belongs to the current app in action. A user can simply click a listed event to continue the appropriate workflow element.

Listing 2.4: Workflow Definition for Multiple Apps

```
package <ProjectName>.workflows

WorkflowElement <WorkflowElementOne>
<...>
WorkflowElement <WorkflowElementTwo>
<...>
WorkflowElement <WorkflowElementThree>
<...>

App <AppID1> {
      <WorkflowElementOne> (startable: STRING),
      <WorkflowElementTwo>
}

App <AppID2> {
      <WorkflowElementOne>,
      <WorkflowElementThree>
}
```

The deployment of multiple apps is similar to that of a single app. In this case, however, not just one but all created apps have to be deployed, e.g. by setting the corresponding symbolic links as described in Section 2.2.2 for each app.

2.4 Additional Features

2.4.1 Uploading and Displaying Files

The current MD^2 version allows for uploading and displaying files such as images to or in an application. Therefore, in addition to conventional data types (e.g. string), the DSL comprises a data type representing files. In the model file of a MD^2 model this type can be assigned to an attribute, which can be marked as optional (cf. listing X). Moreover, in the view file the input element for files – the FileUpload construct – needs to be used. Using this construct a button having the specified attributes and allowing for uploading a file, is displayed on the respective UI form. In the controller file a remote connection specifying the location to which the file should be uploaded needs to be defined and listed in the main block of the controller.

Aside from uploading files, they can be displayed in an application which is done by using the UploadedImageOutput construct in the view file.

Moreover, the content view elements should be linked to content provider fields in the init action to establish a connection between the UI elements and the respective data fields, as it is applicable for all other view content elements.

A specification of a sample image upload and its display, including all applicable attributes, is depicted in the following listing.

Listing 2.5: Upload and Display of Files

```
//in model file:
entity sampleEntity {
      picture : file (optional)
}

// in view file:
// provide upload button
FileUpload pic {
      label "piclabel"
      text "pictext"
      tooltip "pictooltip"
      style picstyle
      width 42%
}

// display image (here: image uploaded before)
UploadedImageOutput pic1display {
      imgHeight 2
```

```
        imgWidth 2
        width 2
}

// in controller file:
main {
        //...
        fileUploadConnection FileUploadConnection
}

remoteConnection FileUploadConnection{
        uri "http://localhost:9090/proxy?sampleUri"
        storagePath "sampleFileUploadPath"
}

// moreover: specify respective content provider and mappings in init action in
    controller
```

2.4.2 Calling RESTful Web Service from the App

With the current MD^2 version it is possible to call RESTful web services that are provided by external applications. To do so, it is necessary to specify the web service's url and REST method (e.g. GET), as well as the parameters to be transferred to it. The parameters are represented as <key, value> pairs and can be sent as query parameters and/or via the body of the request. Accordingly, depending on the option expected by the service to be called, the DSL allows the modeler to specify queryparams or bodyparams.

Aside from static values to be set at design time, it is possible to set a parameter to the value of a particular ContentProviderPath, i.e. the value of a content provider's field, which is derived at run time. If the value is set statically at design time the data types String, Integer, Float and Boolean are allowed.

An exemplary web service description based on the DSL as well as the corresponding call of the action is depicted in the following.

Listing 2.6: Calling a Web Service From Within a Workflow

```
// Specification of the web service call
externalWebService sampleWebService {
        url "http://sampleURL"
        method POST
        queryparams(
                "param1": 42
        )
```

```
        bodyparams (
                "param2": "sampleString"
                "param3": sampleProvider.sampleField
        )
}

// Specify action to call the web service
action CustomAction callWS {
        call WebServiceCall sampleWebService
}
```

2.4.3 Control of Workflow by calling a RESTful Web Service

3 Developer's Handbook

3.1 Installation

3.2 DSL Semantics

The MD^2 framework is intended to provide a cross platform solution, i.e. to generate apps not only for map.apps but also other platforms. For this purpose, this section delivers an overview about the semantics of the DSL, e.g. the different patterns targeted or forms of communication that are implicated by certain model constructs. This will enable future developers to generate apps for other platforms which provide the same functionality as the apps currently generated for map.apps.

First of all, the MVC pattern with additional workflow layer used in the DSL should also be represented in the generated code.

Workflow Layer The workflow layer defines different apps and their workflow elements. Since workflows bundle specific functionality, each app can be seen as a user role, and the assigned workflow elements represent the role's permissions. However, a sphisticated user or role management is not implemented for the MD^2 framework.

Every app is supposed to be given a start screen which contains buttons for workflow elements that can be started from the app as well as a list of open issues that can be continued by the app. The belongingness of workflow elements to apps should be represented in a map in the backend, which connects workflow elements to their apps. This is for example important for the determination of open issues which are allowed to be continued from a specific app.

Similar to that, the backend needs to know the sequence of workflow elements, i.e. which workflow elements are to be started after which event and when to end the workflow. Note that two workflow elements can fire the same event and start different workflow elements. Thus, the backend also needs to know which event/workflow element combination initializes the start of a specific new workflow element.

However, if a workflow element fires an event which starts a new workflow element within the same app, this should be handled by the app-specific EventHandler, i.e. no backend communication is required. This is important if future developments are supposed to allow for temporary off-line usage of apps. Thus,

the backend handles the start of new workflows *across* apps (currently implemented as EventHandlerWS) and the app-secific event handler handles the start of new workflow elements *within* apps.

When a workflow element is started across apps, it will appear in the list of open issues of all apps that have the respective workflow element assigned.

Model Layer The model is a quite thin layer in the overall architecture, the only components contained are entities and enumerations. In order to access core data functionality, a data model has to be setup that defines the database to be accessed later on by the content providers. This database is currently located in the backend and should therefore be accessible by apps from all platforms.

Controller Layer The major and biggest layer in this architecture is the controller layer and has the most important role in connecting the view with the model and vice versa. It consists of several workflow elements, each being an independent controller. The default process chain of a workflow element should be used as starting process chain. Likewise the first view from this process chain should be used as start view for the workflow element. Each workflow element (i.e. each controller) requires its own initialization, e.g. mappings of content to views. The required actions for initializations can be found in the onInit block in a workflow element. When a workflow element fires a workflow event, it should be terminated and the control handed to the app-specific EventHandler or the backend as described for the workflow layer.

Within the body of workflow elements, the controller behavior can be defined using actions and ProcessChains. ProcessChains will be converted to actions in the preprocessor, and therefore do not require a generator for different platforms.

ContentProvider in the controller layer are used for data provision. A webservice-based communication to the backend is required for every platform in order to store and request the data.

View Layer The view layer has not been changed during the course of this project seminar. View elements should be implemented with the functionality described in Chapter 2.

3.3 map.apps Implementation

The current implementation of the MD^2 framework generates web-based apps for a framework called map.apps, which is mainly based on JavaScript. Code generation of Android and iOS apps are also targeted, but not fully implemented yet.

The generated code for map.apps can be subdivided into three parts: static map.apps code, dynamically generated map.apps code and a backend. The static map.apps code contains the part of the code which does not depend on the models created in the MD^2 DSL. Since it is static, it does not need to be generated, but is required for the overall functionality of the generated apps. The dynamically generated part is completely

dependent on the model. The backend is implemented in Java and contains static as well as dynamic code. However, it is completely generated. The backend provides a server which offers functionality such as data storage and communication accross apps.

Each of these three parts of the code is described in detail in the following.

3.3.1 Static map.apps Implementation

The static map.apps code is split into several bundles, which are then used by the generated map.apps code. These bundles are located at `src/main/js/bundles`and are explained in the following subsections.

Keywords nur highlighten wenn freigestellt

Form controls

The form controls are defined within the bundle `md2_formcontrols`. It uses and extends the existing map.apps bundle `dataform` with additional form elements, which can be used in MD^2. Each factory defined within the bundle of MD^2, specifies how a JavaScript-object can be transformed to a data form widget. To define an own dataform or to understand the concepts of a dataform component the map.apps documentation will be helpful.

DateTimeBoxFactory Defines a form control for the component `DateTimeInput`, which is identified by the keyword `datetimebox`. The widget shows a view element showing the time and the date of a `datetime` value.

GridPanelFactory Defines a form control for the component `GridLayoutPane`, which is identified by the keywords `md2gridpanel` and `gridpanel`. The widget enables to structure multiple view elements in a grid.

ImageFactory Defines a form control for the component `Image`, which is identified by the keyword `image`. The widget is able to display a static image within your app.

SpacerFactory Defines a form control for the component `Spacer`, which is identified by the keyword `spacer`. A spacer defines some white space between some components or within the grid of a `GridLayoutPane`.

StackContainerFactory Defines a form control for the component `AlternativesPane`, which is identified by the keyword `stackcontainer`. This widget encapsulates the stackcontainer within `dijit/layout/Stac`
It provides a view elements which has multiple views, but shows only one, similar to a book or a slide show. The user can navigate between them using specific keys.

TextOutputFactory Defines a form control for the component `Label`, which is identified by the keyword `textoutput`. This widget enables to display uneditable text.

TooltipFactory Defines a form control for the component `Tooltip`, which is identified by the keyword `tooltipicon`. This widget offers a tooltip behind a question mark icon.

UploadImageOutputFactory Defines a form control for the component `UploadedImageOutput`, which is identified by the keyword `uploadimageoutput`. The widget is able to display an image within your app, which is uploaded/specified by the user.

List of open issues

The `md2_list_of_open_issues` comprises all code necessary for displaying the list of open issues within the app. This list shows all workflow instances, whose state is at a workflow element, which belongs to the current app. Currently supported data listed in this widget are the guid of the workflow instance, the workflow element name and the last fired event. The list is included as `dijit\_Widget` and is listed as a `Tool` within the app.json under the bundle specifications of `toolset`. In the `ListOfOpenIssuesController` a `DataView` is created, which uses the workflow store as a `DataViewModel`. The workflow store is described in section 3.3.1. In addition to just displaying the workflow instances it is possible to start the workflow element through clicking on the respective entry. Therefore the `ListOfOpenIssuesController` handles the event `onClicked` and calls the function `startWorkflow` of the respective `MD2MainWidget`.

Local store

Location service

Runtime

Add descriptions of folders: content-provider, datamapper, datatypes, entities, events, handler, legacy, resources, templates, util, validators, view

MD2MainWidget Each workflow element (see 2.3) has its own instance of a MD^2 main widget. This is specified in the respective controller of the workflow element bundle inside the app. That is, the `manifest.json` of the workflow element bundle references an `_md2AppWidget` for its controller. Once the controller is activated (i.e. the `activate` function is called), the respective MD^2 main widget instance is built. This MD^2 main widget is implemented in the file `MD2MainWidget`, which serves as the basic starting point to start a workflow. Thus, it provides methods to start a workflow element. There are different ways a workflow can be started. One way is to start it directly from the map.apps tool bar. Another way is to start a workflow from the list of open issues (see section 3.3.1). Figure 3.1 depicts this initialization process.

Erlaeuterung der transactionID im WorkflowStateHandler + Erlaeuterung vom WorkflowStateTrans-

Workflow For each started workflow instance an unique ID is generated and assigned to that instance. This is done in the method `startWorfklow` of the `MD2MainWidget`. A global `WorkflowStateHandler`

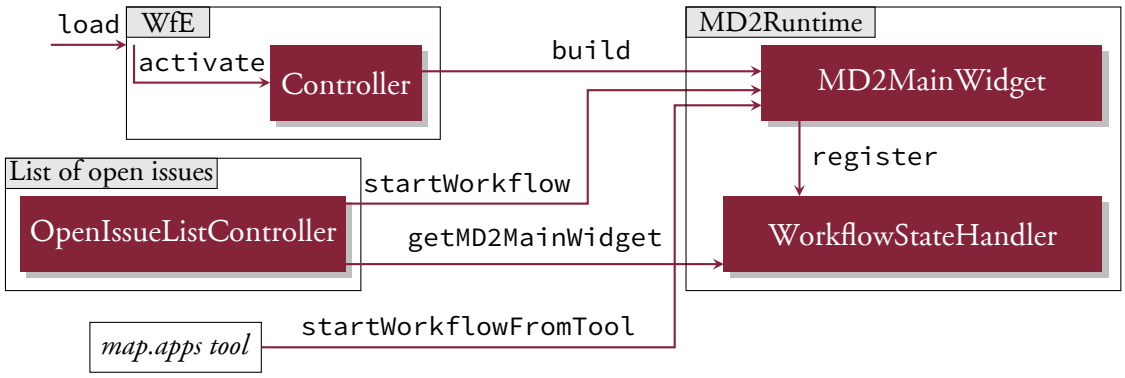

Figure 3.1: Initialization of the MD^2 main widget in order to start a workflow

provides methods to set and get the currently active workflow instance ID. This information is needed to suspend a workflow and to resume that workflow at a later time.

Actions Each action that is defined in the MD^2 DSL must also exist in the MD^2 runtime bundle in map.apps to be used. Individual actions are stored in the subfolder `simpleactions`. Moreover, an `ActionFactory` must provide a simple method that returns an instance of the respective action (e.g., `getLocationAction` for the `LocationAction`). All actions provide a method `execute` that implements the action. An individual constructor allows to initialize the action, e.g., setting the city's name for a `LocationAction`. The `ActionFactory` is instanced in the method `build` of the `MD2MainWidget` (see figure 3.1). Thus, every workflow element can access and use this factory.

Store

Workflow store

3.3.2 Dynamic map.apps code

3.3.3 Backend

3.4 map.apps Generator

A Sample Workflow