Westfälische
Wilhelms-Universität
Münster

MD²

Project Seminar
Model-driven Mobile Development

MD² – Handbook

Group Members:
Jan Christoph Dageförde
Julia Dittmer
Andreas Fuchs
Carolin Gülpen
Holger Koelmann
Malte Möser
Tobias Reischmann

Supervisors:
Prof. Dr. Herbert Kuchen
Jan Ernsting

Group for Practical Computer Science
University of Münster

29th March 2015

Contents

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 MD^2 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.

Modeler's Handbook

This chapter serves as a handbook for people who want to use the MD^2 framework to develop mobile business applications. It consists of

- installation instructions (cf. 1),
- a guideline for building single Apps (cf. 2.2), providing information about the
  - DSL and its language constructs,
  - the process of model creation using MD^2,
  - as well as the code generation,
  - and the deployment process of the developed App,
- information needed for the development of multiple Apps (cf. 3)
- and information about more advanced additional features of the language (cf. 4).

In general, the MD^2 framework provides the modeler with an understanding of how to generate a multitude of apps and a corresponding backend out of a single model source. Using Eclipse as an IDE, convenient features, like the auto formatting of the model code by pressing `STRG+SHIFT+F`, can be used.

1 Installation

1.1 Setting up your MD^2 Model Development Environment

The following steps will provide you with the software required to enable modelling of MD^2 models:

- Download a current Eclipse IDE with support for Java EE development (e. g., Luna).
- Install a current version of the Xtext redistributable using the Xtext Update Site.
- Install the MD^2 features from the archive that you obtained together with this documentation.

1.2 Setting up your map.apps Development Environment in NetBeans

As a prerequisite, ensure that the following software is installed:

- map.apps 3.1.0
- NetBeans EE (e. g., Version 8.0)

- Apache Tomcat 7.0 with a running map.apps runtime[1]

1. Set up your map.apps development environment in NetBeans.
   a) Extract the `sampleProjRemote` project from the map.apps distribution and open it in NetBeans.
   b) In its `pom.xml`, set the `mapapps.remote.base` directive to the URL where the map.apps runtime is installed.
   c) Start the Jetty web server from the project's context menu.
2. Deploy the generic MD^2 runtime bundles by copying the MD^2 runtime bundles into the directory `src/main/js/bundles/` within the project:

   ```
   md2_formcontrols
   md2_list_of_open_issues
   md2_local_store
   md2_location_service
   md2_runtime
   md2_store
   md2_workflow_store
   onlinestatus
   ```

1.3 Setting up GlassFish to Run Generated Backends

To deploy the generated backends, follow the subsequent steps:

1. In Eclipse, open the "Servers" tab.
2. Right-click it and choose "New" → "Server".
3. If the GlassFish adapter is not installed yet (look for "GlassFish" in the list of types), click "Download additional server adapters" and install the entry "GlassFish Tools".
4. From the list of types, choose "GlassFish" → "GlassFish 4.0" and click "Next".
5. For the Glassfish Server Directory field, navigate to the `glassfish/` subdirectory in your installation. If you entered the correct path, it should output something similar to

   "Found GlassFish Server version 4.0.0".

   Otherwise, follow the assistant's hints.
6. Click "Finish".

[1] For details on their installation, please refer to the map.apps documentation.

2 Getting Started

2.1 Creating a Project

1. Initialize a new project by navigating to `File > New > Project...`. There, choose `Other > MD2 Project` and click `Next`.

2. Choose a project name that does not contain whitespace or other non-alphanumerical characters. If necessary, choose a location different from the proposed default.

3. After clicking on `Finish`, a default project structure is generated which you can extend as you need.

The new project contains the following folders:

- `src/` contains the MD^2 models of your application and is initialized with a simple default model.
- `resources/` contains resources that the generator will copy into the generated applications.
- `src-gen/` contains all artifacts that the generator derives from your model. This folder will contain multiple subfolder for different platforms. For example, there might be a folder for the backend and for map.apps. Note, that everything in this directory will be overwritten when the generator is run.

2.2 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. At its core, MD^2 is structured according to the MVC-pattern. MVC stands for Model-View-Controller and ensures the principle of single responsibility for these classes. In order to enable the modelling and generation of worklows within across apps in MD^2, this architecture was extended to include an additional layer, the workflow layer (cf. Figure 1).

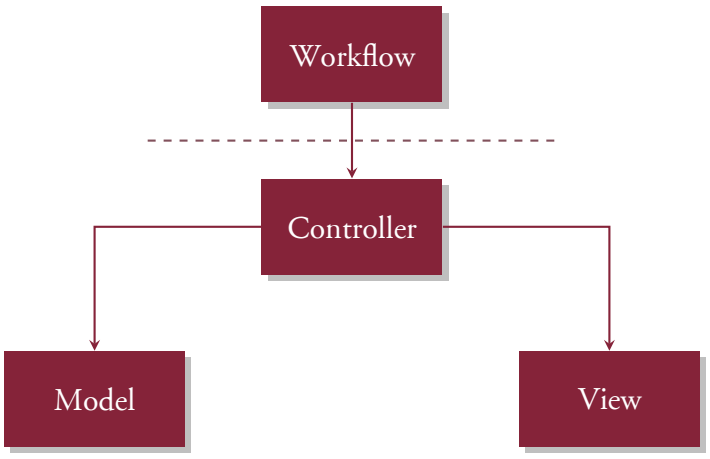

Figure 1: Architecture of MD^2 models

All components in MD^2 are organized in a package structure that corresponds to the aforementioned structure. Documents have to be placed in these packages (views, models, controllers or workflow). For example, all view files are expected to be in the package `any.project.package.views`. In every MD^2 file the package has to be defined as follows:

```
package PACKAGE_NAME
```

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

2.2.1 Workflow

The workflow layer provides abstraction on top of the controller layer. While the controller contains most of the business logic, the workflow layer allows to specify the general course of action of one or more apps using a few simple and easily understandable model language constructs. Furthermore, the 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, workflow elements can not only be started but the workflow can also be terminated using the keywords `end workflow`. A workflow element in the workflow layer typically looks as shown in Listing 1.

Listing 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 application that executes the workflow elements. As shown in Listing 2, an app consists of its ID, a list

of workflow elements that are used in the app and a name that is used as the app's title. In the scenario where only a single app is modeled, all workflow elements can be included in the app. However, it is also possible to have unused workflow elements.

Listing 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 with `startable` in the app specification. In the final application 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 a label or description for the button.

A complete workflow specification for one app will be structured as shown in Listing 3. Note, that MD^2 does not explicitly define 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 3: Workflow definition in MD^2

```
package <ProjectName>.workflows

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

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

2.2.2 Model

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

2.2.2.1 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 (i. e. true or false)
- `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]]`
- `file` – a file to be uploaded and stored in an entity field

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 empty values the parameter `optional` must be set. Further supported parameters depend on the used data type and are available 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 an info icon, such as 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". An exemplary entity is depicted in Listing 4.

Listing 4: 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[]
}
```

2.2.2.2 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, as it is depicted in Listing 5. Other data types are not supported.

Listing 5: Exemplary enum element to specify weekdays

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

2.2.3 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.

2.2.3.1 Container Elements

Grid layout panes 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 one is automatically calculated by MD^2 during the generation process. 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 pane* arranges elements (containers or content elements) either horizontally or vertically. By default all elements are arranged vertically 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]>
  <...>
}
```

2.2.3.2 Content Elements

Input elements can be used to manipulate model data via mappings (see Section 2.2.4). At the moment BooleanInputs, TextInputs, IntegerInputs, NumberInputs, TimeInputs, DateInputs, DateTimeInputs, OptionInputs and FileUploads 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` provides a boolean for whether an input field is enabled or disabled. 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 are not given, the displayed options depend on the enum type of the attribute that has been mapped on the input field (see Section 2.2.4).

```
OptionInput NAME {
  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 exist 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 noted in parentheses directly after the label definition. The optional style can either be noted directly, or by referencing an existing style definition (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

While the `Image` construct can only be used to display images that have a fixed URI, the *UploadedImageOutput* can be used to display images that are stored in a field of an entity and were uploaded before. `UploadedImageOutput` features the same attributes as `Image` and is specified as follows.

```
UploadedImageOutput NAME {
  height INT
  width INT
}
```

FileUpload is an input element for files. Using this construct, a button having the specified attributes and allowing for uploading a file is displayed on the respective UI form. `FileUpload` can be specified using the following attributes.

```
FileUpload NAME {
  label STRING
  text STRING
  tooltip STRING
  style <[style] | style>
  width INT%
}
```

To be able to display an uploaded image using the `UploadedImageOutput` construct, it is necessary to map the respective view elements to the corresponding entity fields in the controller as it is applicable for all other view content elements.

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 generated spacers, i. e. the number of occupied cells in a `GridLayoutPane`, can be specified.

```
Spacer (INT)
```

The *AutoGenerator* is used to automatically generate view elements to display all attributes of a related entity and the corresponding 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 content provider 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 an 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.

Sixteen default web colors are available as named colors: `aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, yellow`.

Elements can not only be defined at the place they should be used, but there is also a mechanism to define an element once and *reuse* it several times. Instead of defining a new element, another element can be referenced – internally this leads to a copy of the actual element during the preprocessing. 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
```

2.2.4 Controller

2.2.4.1 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 [RemoteConnection]
  defaultConnection [RemoteConnection]
  fileUploadConnection [RemoteConnection]
}
```

The attributes are explained as follows:

- `appVersion` – a string representation of the current app version, e. g., "RC1"
- `modelVersion` – 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, so that it is not necessary to specify the same connection in each content provider
- `workflowManager` – the remote connection to the workflow manager, which may be different from the default backend connection
- `fileUploadConnection` (optional) – a remote connection to the file upload server must be specified if a file is expected to be uploaded or an uploaded file to be displayed. This connection can be similar to the default connection, but storing large amounts of files on a separate server is encouraged. Specifying the connection is obligatory if at least one `UploadedImageOutput` or `FileUpload` is present in the views.

Furthermore, the controller layer is subdivided into one or more workflow elements. While a 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 allow to define a sequence of steps inside the workflow element and are further described in Paragraph 2.2.4.3. 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 be derived from it.

The `onInit` block is used to define everything that is supposed to happen upon the 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 during the initialization. This is enforced by through a validator that raises an error in case the modeler tries to throw an event here.

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`.

2.2.4.2 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 exists 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 perform functionality:

- 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 4.1).
- The LocationAction allows to extract an address from content provider fields and 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 (<parameters>)
  on|from
  <[ContainerElement] | [ContentElement]> <...>
```

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

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

```
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 StringRangeValidator 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> <...>)
```

2.2.4.3 Process Chains

A ProcessChain is used to define several steps in which the workflow element can currently be. It is possible to define several process chains. Process chains can be nested and there can be only one active process chain.

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

Each ProcessChainStep specifies a view that will be displayed once the process chain step becomes the current process chain step of the active process chain. Additionally, conditions can be defined that restrict switching to the next or previous process chain step. Events can trigger changing to the next or previous process chain step.

Instead of the default of proceeding and reversing along the process chain, another step in the process chain will become active as successor or predecessor, if referenced using `goto` or `returnTo`.

```
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
```

Process chains can be refined using sub process chains.

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

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

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

2.2.4.4 Content Providers 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) or 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.

2.2.4.5 Remote Connections A remote connection allows specifying a URI for the backend communication as well as a path to specify the storage location of files to be uploaded. The backend must comply with the MD^2 web service interface as specified in Appendix C.

```
remoteConnection NAME {
  uri URI
  storagePath PATH
}
```

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.

2.3 Deploying a Single App

2.3.1 Backend

The following steps will start the GlassFish server that is used to deploy the backend. Note, that in order to access the server you might need to grant additional privileges in the configuration of your firewall.d.

1. Within the GlassFish installation directory, navigate to `glassfish/bin/`.
2. Run the `asadmin` utility (Windows: Double-click on `asadmin.bat`, Linux/OS X: Open a terminal in that directory and run `./asadmin`).
3. In the GlassFish administration utility, type `start-database` to start the Derby database for the backend. In a Unix environment, you can combine both steps by running `./asadmin start-database`.
4. Start Eclipse and import the generated project `<PROJECT_NAME>.backend` by choosing "General" → "Existing Projects into Workspace" in the import wizard.
5. In the Project Explorer tab, right-click the project in Eclipse, choose "Properties", and navigate to "Targeted Runtime".
6. Deselect all runtimes and click "Apply".
7. Select the item "GlassFish 4.0" and click "Apply".
8. Correct JRE-related build path problems, if any, by resorting to the default JRE.
9. Confirm by clicking "OK".
10. In the Servers tab, right-click the "GlassFish 4.0" entry, and choose "Add/Remove".
11. Add the backend project to the server.
12. Start the server.

2.3.2 map.apps

You have two options to deploy a map.apps app in your NetBeans development environment:

- Copy it into the `src/main/js/app/` directory of the project, or
- use a symbolic link to reference apps from another location (see Appendix 1).

An benefit of the second variant is that newly generated code, that results from changes to your model, will automatically be available in your map.apps development environment. If the server is running, the code will also be deployed to the server automatically.

3 Development and Deployment of Multiple Apps

The MD^2 framework 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 6. Apart from that, the workflow will look as usual, with the sequence of workflow elements being determined via events. 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. A user can simply click on a listed issue to continue the workflow in the appropriate workflow element.

Listing 6: 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 2.3 for each app.

4 Additional Features

4.1 Calling RESTful Web Services from an 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 (currently GET, POST, PUT and DELETE are supported), 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 (as they will internally be converted into JSON).

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

Listing 7: Calling a web service from within a workflow

```
// Specification of the web service call
externalWebService <externalWebServiceCallOne> {
  url URL
  method (GET | POST | PUT | DELETE)
  queryparams(
    STRING : (INT | STRING | FLOAT | BOOLEAN | <ContentProviderPath>)
    STRING : (INT | STRING | FLOAT | BOOLEAN | <ContentProviderPath>)
    <...>
  )
  bodyparams (
    STRING : (INT | STRING | FLOAT | BOOLEAN | <ContentProviderPath>)
    <...>
  )
}

// Specify action to call the web service
```

```
action CustomAction <CustomActionOne> {
  call WebServiceCall <externalWebServiceCallOne>
}
```

4.2 Controlling a Workflow by calling a RESTful Web Service

The MD^2 language offers a possibility to define a RESTful web service according to Listing 8, which will start a certain workflow element. For this purpose for each workflow element marked as `invokable` a web service is generated with one endpoint for each invoke definition. When an invoke definition is placed within a workflow element of the controller model, the respective workflow element in the workflow model has to be marked as invokable as well. An event description can be added, which will be shown in the list of open issues as the event which was fired last.

Listing 8: Offer a web service to start a workflow

```
invokable at STRING using (POST | PUT) {
  <ContentProviderPath> as ALIAS
  default <ContentProviderPath> = <Value>
  set <ContentProviderPath> to <ContentProvider>
}
```

The minimally required invoke definition is simply the keyword `invokable`. The standard path where an endpoint is injected is "/". If another path should be used this must be defined after the `at` keyword. If multiple endpoints are defined, setting the path is mandatory.

The standard REST method used for the RESTful web service endpoint is `POST`. However, after the keyword `using`, the modeller can choose `PUT` instead. In the body of an invoke definition it can be defined if entities and their attributes should be set during the web service call. Three different possibilities exist for this purpose. In the following they are described in the order of their appearance within Listing 8.

- The first type allows attribute values to be set by the web service call. This means that the attribute is transformed to a parameter of the endpoint and is then set to the received value. For the name of the parameter either the name of the attribute is used or an alternative alias can be defined.
- If some attributes should always receive the same value regardless of the parameter values, they can be set to a default value using the second type. An example would be a status field which is set to "issue received" when the workflow is started.
- The last type is similar to setting a content provider to an attribute (cf. Section 2.2.4). Since the language only knows how entities are related to each other, but not their corresponding content providers, this statement is needed for every nested entity.

For each entity referenced within the definition an instance of it will be created and persisted. The type of persistence depends on whether the remote connection of the content provider equals the one of the

`workflowManager`. If they are equal, the web service call and the persistence is handled on the same server, thus the internal Enterprise JavaBeans can be used. Otherwise, the other external backend server needs to be called – this is however not implemented in the current version of the MD^2. It is not only necessary that the URLs of the remote connections are equal, but the objects also need to be identical. If the body is missing, no entities or attributes are set.

Developer's Handbook

This chapter is intended to provide MD^2 developers with detailed information necessary to further develop the framework. In its current version, this handbook contains information about the

- installation procedure (Section 1),
- DSL semantics (Section 2),
- structure of the map.apps and backend implementations (Section 3 and Section 4, respectively),
- the preprocessor (Section 5),
- and the map.apps generator (Section 6).

Regarding the implementation of map.apps in particular, Section 3 also addresses the structure and interaction of the generated and static code.

Note that changes to the DSL can have a direct effects on validators and the formatter. This should be kept in mind during the planning and implementation of future changes. The whole development project together with additional tools and example apps can be obtained from different repositories belonging to the project's GitHub organisation.

1 Installation

In order to get your development environment up and running, please follow the installation instructions of Section 1. However, do not install Eclipse as described there, but instead install the following:

- Eclipse IDE for Java and DSL Developers (e. g., version Luna, which contains Xtend and Xtext 2.7.3)
- Eclipse features:
  - GlassFish Tools for Luna
  - Eclipse Java EE Developer Tools
  - Eclipse Java Web Developer Tools
  - Data Tools Platform Extender SDK
  - JSF Tools

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 such as Android or iOS. 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 implied 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 sophisticated user or role management is not implemented in the MD^2 framework.

Every app is supposed to have a start screen which contains buttons for workflow elements that can be started in the app as well as a list of open issues (workflows in a specific state) that can be continued by the app. The belongingness of workflow elements to apps is 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 event handler, so that no backend communication is required. This is important to allow temporary off-line usage of apps in the future. Thus, the backend handles the start of new workflows *across* apps (currently implemented as EventHandlerWS) and the app-specific event handler is responsible to start 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 rather 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 biggest and most important layer in this architecture is the controller layer, which has the role to connect 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 event handler or the backend as described for the workflow layer.

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

Content provider in the controller layer are used for data provision. 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 Section 2.2.3.

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 written in JavaScript. Code generation for Android and iOS applications 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 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 across apps.

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

3.1 Static map.apps Implementation

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

3.1.1 Form Controls

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

DateTimeBoxFactory Defines a form control for the component `DateTimeInput`, which is identified by the keyword `datetimebox`. The widget shows a view element that displays 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 sets whitespace between 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 stack container within `dijit / layout/StackContainer`. It provides a view element 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 non-editable 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`, identified by the keyword `uploadimageoutput`. The widget is able to display an image within your app, which is uploaded/specified by the user.

Special dataform elements enable the use of uploaded files. The `UploadedImageOutput` displays images which have been uploaded by an app's user using a `FileUpload` input element. Given that both elements are mapped to an entity's attribute of type `file`, these elements retrieve an image from, or store an image on the server, respectively. For this procedure, a specialised `remoteConnection` needs to be defined by

the modeller, thus defining the remote location of this service and a local path where this service is able to store files (`fileUploadConnection`). Consequently, this remote connection can be different from every other remote connection used, e. g., by content providers.

As a further consequence, the uploaded file is not directly stored in the database. Instead, when called by a `FileUpload` element, the upload service stores the file on disk at the specified path (`storagePath`) and returns an identifier string of this file to the calling client (cf. Figure 1, step 1). This identifier is then used throughout the model, particularly as the value of a corresponding attribute of type `file` (cf. Figure 1, step 2).

When such an identifier is encountered as the value of an `UploadedImageOutput`, this element calls a servlet at the `fileUploadConnection`, passing the identifier as a parameter (cf. Figure 1, step 4). That way, the image is downloaded for the client just as soon as it is needed, instead of loading it at every initialisation of entities by a content provider. Note that currently only JPEG images should be uploaded, since the servlet always tries to output any given file using the content type `image/jpeg`.

3.1.2 List of Open Issues

The `md2_list_of_open_issues` comprises all code necessary to display the list of open issues within the app. This list shows all workflow instances, whose state is at a workflow element that belongs to the current app. Currently, the data listed in this widget are the the workflow element name, the last fired event and

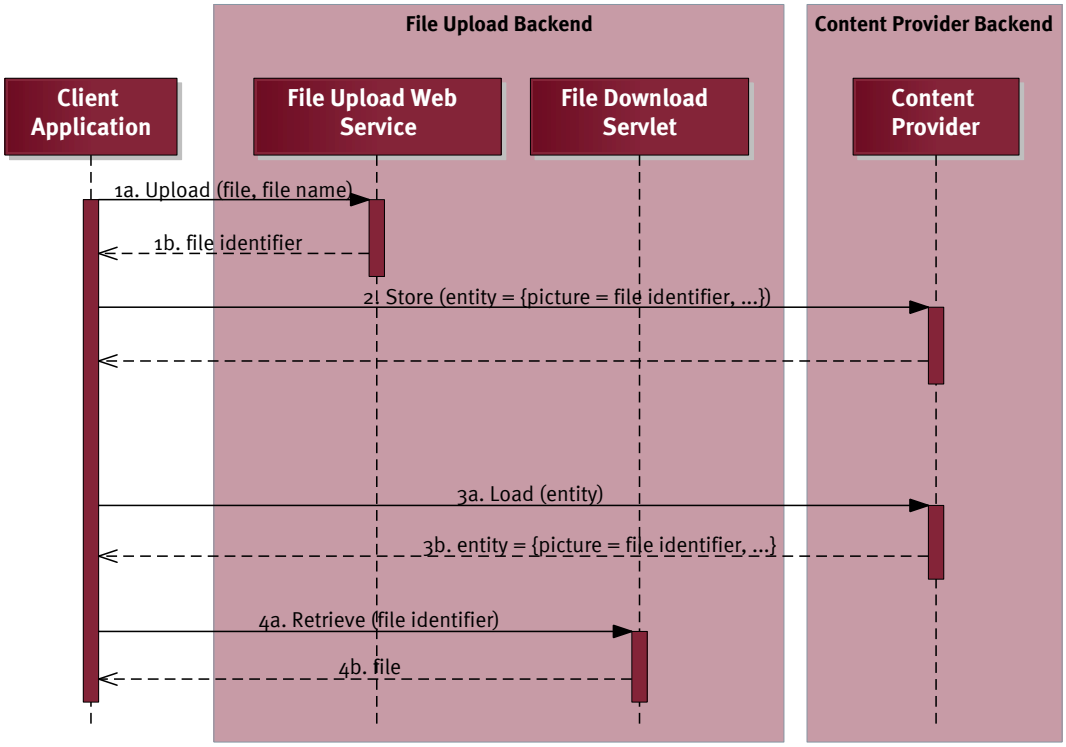

Figure 1: Procedure of uploading and retrieving user-uploaded images

a timestamp for the last modification of the workflow state. The list is included as `dijit_Widget` and is listed as a `Tool` in the app.json under the bundle specifications of the `toolset`. In the `ListOfOpenIssues Controller` a `DataView` is created, which uses the workflow store as a `DataViewModel`. The workflow store is described in Section 3.1.7. Workflow instances are not only listed, but it is possible to start the workflow element by clicking on the respective entry. Then, the `ListOfOpenIssuesController` handles the event `onClicked` and calls the function `startWorkflow` of the respective `MD2MainWidget`. The workflow instance ID is retrieved in combination with the content provider IDs of its current state. With these, the content provider referenced in the `workflowStateHandler` are set to their respective values.

3.1.3 Local Store

The local store within the bundle `md2_local_store` is one of three stores used in the context of map.apps within MD^2. This store implements some of the guidelines from the `dojo/Store` interface, which means, that it offers the methods `query`, `get`, `put`, `add`, and `remove`. The local store can be used by a content provider (set to local within the controller model). This store saves all data as cookies in the browser. Thus, the store is not meant for consistent data storage.

3.1.4 Location Service

The `LocationStoreFactory` provides methods to convert a longitude and latitude pair into an address (i.e. a country, city, street, postal) and vice versa. In the first case, the method `_getAddressForLocation` is used, which takes two parameters for the longitude and latitude value. In the second case, the method `_getLocationForAddress` is used. This method takes a single string as input, which contains all the address information (e. g., *Schlossplatz Münster 48149*). For both methods, the result is a JavaScript object.

ArcGIS is the underlying API that is used for this (reverse-) geocoding. The URL to use this service is specified in the `manifest.json` of that bundle. Currently, the URL is:

`http://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer`.

3.1.5 Runtime

This bundle contains the main logic of the MD^2 map.apps framework, which is mainly based on the `MD2MainWidget` object. Most other sub bundles just enhance the functions of this widget.

3.1.5.1 MD2MainWidget The `MD2MainWidget` is for example responsible for the opening and closing of views. Each workflow element (see Section 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

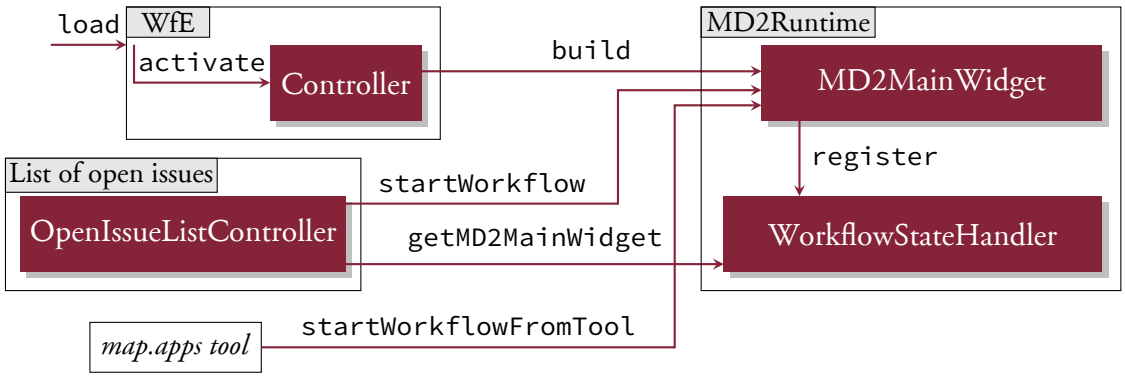

Figure 2: Initialization of the MD^2 main widget in order to start a workflow

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:

- A workflow can be started directly from the map.apps tool bar. Each workflow element marked as `startable` in the model gets its own tool. After clicking on it, the method `startWorkflowFromTool` is started, which first resets all current workflow elements and the workflow handlers and then calls the `startWorkflow` function.
- A workflow can also be started from the list of open issues (see section 3.1.2). Therefore, the `startWorkflow` is directly started from the `OpenIssuesListController`.

When the workflow element window is closed and the same operation to start a workflow element is called again (reopening the same entry of the list of open issues or restart the same tool) the same window with all data entered before will be opened. However, all changes are dropped if another workflow instance is opened in between! Each `MD2MainWidget` contains a runtime variable $, which contains important objects needed within the context of the widget. While many objects are created anew for each widget, some objects such as the `WorkflowStateHandler` (cf. Paragraph 3.1.5.13) are globally used and thus should only be created once. Therefore, many of these objects are injected as singletons within the `mainfest.json` of the bundle `md2_runtime`. Figure 2 depicts this initialization process.

3.1.5.2 Actions Each action type defined in the MD^2 DSL must also exist in the MD^2 runtime bundle. Individual actions are stored in the subfolder `simpleactions`. Moreover, an `ActionFactory` must provide a 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 a city's name for a `LocationAction`. The `ActionFactory` is instantiated in the method `build` of the `MD2MainWidget` (cf. Figure 2). Thus, every workflow element can access and use this factory.

3.1.5.3 Content Provider The content providers are responsible for saving and persisting the state of one or multiple objects. While the generated code only provides factories for the creation of content providers, the actual code is located in the static bundles. Each content provider is instantiated with a unique name, the app ID it belongs to, a store (either remote or local), the information whether it is a provider for a list of objects, a filter restricting the queried items and the information whether it is a remote or a local store. Besides functions to get or set the content of the provider, functions are offered to access the injected store and persist the data. Additionally, each content provider can inform other components of the app about changes within attributes. This can be used for example to refresh the values of view elements. The function `restore` and `reset` are used within the `WorkflowStateHandler` to influence the state of the content provider, for example when the workflow instance is changed.

3.1.5.4 Data Mapper Whenever a value within a content provider changes it is necessary to inform the view elements to be able to refresh them. This is one within the classes of this folder.

3.1.5.5 Data Types In this folder all data types known by the MD^2 map.apps framework are listed. Each data type provides additional functions for working with the data objects such as cast or compare operations. The `TypeFactory` is used to instantiate an object according to its data type and is injected to the runtime variable $ within the `MD2MainWidget`.

3.1.5.6 Entities This folder contains all internal entities known to the MD^2 map.apps framework. Currently, this is the `Location` entity and the abstract class `_Entity`, which is inherited by all generated entities.

3.1.5.7 Events In the MD^2 modelling language it is possible to define actions based on events. Examples for such events are changes or clicks on view elements. To be able to map this behaviour in map.apps, each possible events has its own class which subscribes a topic associated with the type of event it represents. The `EventRegistry` has a list of all possible events and the root event classes enable to register actions to the specific events.

3.1.5.8 Handler This folder contains handlers for global events. These mainly display results in info or warning messages. Certain actions are bound to events by subscribing to topics. One example is the data action bound to the topic `"md2/contentProvider/dataAction/${appId}"`. If any component is publishing a status (`"success"` or `"error"`) for an action (e. g., `"load"` or `"save"`) a respective info message is shown in the lower right corner of the application.

3.1.5.9 Resources This folder contains images and style files.

3.1.5.10 Templates This folder contains the root html file of the `MD2MainWidget`.

3.1.5.11 Validators In the model validators can be defined for view elements. This folder contains all existing validators, which can be created using the `ValidatorFactory` which is injected in the runtime variable $.

3.1.5.12 View The `MD2MainWidget` is responsible for creating the views for the workflow elements. For this purpose it uses a `ViewManager` which creates the view elements based on the `view` entries within the manifest.json of the respective workflow element. The type of a view element is therefore linked to a dataform component either contained in the bundle `md2_formcontrols` or within the external conterra bundle `base/dataform`.

3.1.5.13 Workflow The `WorkflowStateHandler` and the generated `WorkflowEventHandler` are responsible together for managing the state of the current workflow instance. This includes the transitions between workflow elements within the context of one app as well as firing workflow events to the backend. For each started workflow instance a unique ID is generated and assigned to that instance. This is done in the method `startWorfklow` of the `MD2MainWidget`. A `WorkflowStateHandler` provides methods to set and get the currently active workflow instance ID in a global context. This information is needed to suspend a workflow and to resume that workflow at a later time. The variable `_lastStartedTool` provides information about how the workflow instance has been started. This is important when it comes to decide for a new `startWorfklow` evaluation, if the current workflow instance has to be resumed or a new one should be created.

To change the current workflow element, the `WorkflowStateHandler` provides the methods `change WorkflowElement` and `fireEventToBackend`. The first one opens another `MD2MainWidget`, as the workflow is continued in the same app. The latter one will exit the workflow instance for the current app and close all windows, as the workflow is supposed to continue in another app. Additionally, the backend is informed about this step by calling the `fireEventToBackend` method of the `WorkflowStore` (cf. Section 3.1.7). Since the `WorkflowStore` is supposed to send the content provider IDs, this leads to a problem. The saving operation of the content providers is usually done shortly before the workflow event is fired. Since the content provider IDs are not set until the backend has answered to the web service calls, they may not be accessible yet. For this purpose, the `WorkflowStateTransaction` is used. With each new workflow instance ID a new transaction is created. It keeps a list of all started content provider saving operations and only allows to fire a workflow event when no save operations are in progress. Additionally, the transaction is informed via a subscription about the termination of each content provider operation and will then retry to fire a workflow event, if one was queued before.

3.1.6 Store

The `md2_store` bundle provides the second of the three stores. It could also be called remote store, since it provides access to external data storage. It is again an implementation of the `dojo/Store` and implements all necessary functions. The store is used within a content provider to query the current state of the objects belonging to the current workflow instance. In contrast to the local store, data which is saved within this store is persisted throughout the whole application landscape. For this purpose each store needs to be provided with an URL, pointing to the respective backend server. The data is then queried and stored using REST web services. One implementation of such a backend is automatically generated by the backend generator described in Section 4. For the store to be able to find its respective backend, the URLs need to match each other (see Section 2.3.1).

3.1.7 Workflow Store

The workflow store within `md2_workflow_store` differs from the other stores provided by the static bundles. While the local (cf. Section 3.1.3) the remote store (cf. Section 3.1.6) are used within a content provider, the workflow store has its own purpose. It is used to save and query the status of all workflow instances. It is injected in the `WorkflowStateTransaction` of the `md2_runtime` bundle (cf. Paragraph 3.1.5.13). There, it is responsible of informing the backend of any changes within the workflow instance. In the model this action can be represented by the `FireEventAction`. Beside the information which event has been fired and which was the current workflow element and instance ID, the backend gets a list of all IDs registered in the content providers. This is necessary in order to be able to restore the state of the content providers in another app.

The workflow store is also referenced in the list of open issues. The workflow store implements, similar to the other stores, the functions of the `dojo/Store`. This makes it possible to hand the workflow store over to a `DataView`, which then displays the information retrieved from the store. The `query` function enables to query the whole list of the list of open issues.

For the workflow store to work correctly, the app.json needs to provide an appropriate configuration for the REST location and the current app ID, for which the web service should filter. The following snippet (with equivalent values) is automatically generated:

```
"md2_workflow_store": {
  "MD2WorkflowStore": {
    "uri": "http://localhost:8080/ReferenceProject.backend/service/workflowState
       /",
    "app": "Citizenapp"
  }
}
```

3.2 Generated map.apps Implementation

The generated code of the map.apps applications contains everything, that can not be generalised in the static implementation. This includes mainly the defined content providers, entities, and workflow elements, as well as the workflow mechanism described in the workflow model. Additionally, some components of the static map.apps implementation are injected with model specific settings.

It is important to note that each app definition within the workflow model will result in its own app. Each app gets its own content provider, model, and workflow element implementation. However, files for those objects are only created when they are necessary within the context of the specific app.

3.2.1 Content Providers

The content providers are created as an individual bundle within each app. Each content provider is represented within its own file. However, the file does not incorporate the content provider itself, but merely a content provider factory. It is used to create an instance of the content provider class included in the static `md2_runtime` bundle (cf. Paragraph 3.1.5.3).

Each content provider is created using the name defined in the model, the app ID and an instance of the respective local or remote store. The stores are created using a store factory injected within the manifest.json. Additionally, the remote stores are initialised with the URL of their corresponding remote connection.

Besides the content providers defined within the model, two local content providers are created (namely `__returnStepStackProvider` and `__processChainControllerStateProvider`). They enable the usage of process chains within a workflow element. Therefore, they save the steps taken and their respective state so that it is possible to return to a previous step.

3.2.2 Models

The entities are grouped within the bundle `md2_models`. Each defined entity and enum in the model gets its own file. Additionally, the two entities `__ProcessChainControllerState` and `__returnStepStack` are created to map the state of the two content providers listed above. Each entity inherits from the static class `_Entity` described in Paragraph 3.1.5.6. They describe their own data type as well as the types of all their attributes, or, in case of an enum, the possible values. Additionally, the entities have an `_initialize` function to create empty attribute types for all attributes, except for the referenced entities. For those the values have to be set manually using a return value of another content provider later in the code.

3.2.3 Workflow Elements

For each workflow element of the app an individual bundle is created. This bundle specifies a `Controller`, which is the instance factory for a `MD2MainWidget`. The `MD2MainWidget` itself is further specified within

the manifest.json. Here, all view elements are defined in addition to the app ID, the workflow element ID, the webserviceBackendURI, the window title, and the action called upon initialization of the workflow element. This `onInitialized` action is executed by the `MD2MainWidget` when opening the widget. Besides the `Controller`, all actions which inherit from the static class `_Action` are specified. The `CustomActions.js` implements a class that serves as an instance factory for all actions defined within the workflow element.

3.2.4 Workflow

This bundle contains the `WorkflowEventHandler`, which keeps a list of all `MD2MainWidget`. Additionally, it contains the specification for which workflow follows which, and for the differentiation between the case when an event has to be fired to the backend or a workflow element must be changed within the app. After exiting the context of a workflow instance, the event handler has the capability to reset all `MD2MainWidget` instances registered in order to be prepared for a new instance.

3.2.5 App.json

The app.json is used to inject further information into the static bundles. These settings are mainly backend URLs and app IDs, which are needed in this components. Besides that, the app.json contains a list of all bundles used within the app context.

4 Backend

The MD^2 backend is implemented in Java. It is automatically generated by from the MD^2 model. Note, that some parts of the backend are static while others are contingent upon the model.

4.1 Beans

For entities that are used in at least one remote content provider a stateless session bean is generated. Such a bean provides basic methods to create or manipulate entities of its type. The Java Persistence API (JPA) is used for the persistent storage of the data. The persistence configuration file is located at `META-INF/persistence.xml`. Currently, EclipseLink is used as persistence service.

Additionally, a static session bean is generated for the workflow state of the instances of a workflow. Internally, a workflow instance is identified by an unique ID that is represented as an integer. However, since a workflow instance is generated at the client side, a single client cannot assure that a specific number is not used by another workflow instance of another client. Therefore, every client generates its own `instanceId` that is a hash value of the current time and other variables, and thus supposed to be unique. This generated hash value is represented as a string value.

4.2 Datatypes

The datatypes used in the backend are static and implemented as simple wrappers. For instance, every entity has a unique internal identification number (`internalID`) that is an integer value. The respective wrapper implementation is depicted in the following listing.

Listing 9: An integer wrapper for the internal identification number

```
@XmlRootElement(name = "internalId")
public class InternalIdWrapper {

  @XmlElement
  protected int __internalId;

  protected InternalIdWrapper() {
    // no-arg default constructor necessary
  }

  public InternalIdWrapper(int integer) {
    this.__internalId = integer;
  }
}
```

4.3 Entities

The entities and enumerations defined in the MD^2 model are generated into the subpackage `entities.models`. Moreover, two static Java classes are generated: `RequestDTO` and `WorkflowState`. The former is used as an encapsulation for all client requests, e. g., to create a corresponding REST request. The latter is used as a representation of the state a particular workflow instance has. A workflow can consist of multiple workflow elements, that in turn can fire different events. Thus, every started workflow (i. e. every workflow instance) must keep track of its current workflow element and the last event fired in it.

4.4 File Download Servlet

This servlet is used to deliver uploaded files to a requesting client. It is accessible at `/DownloadFile` below the web root of your deployed project.

In a `GET` request to this servlet you need to set the parameter `file` to the identifier of an uploaded file, which was returned by the file upload web service before (cf. Figure 1). All files are stored in the file system. Therefore, the download servlet needs to look for files using the identifier in a central directory, which is also referenced by the upload web service. This location is defined in the generated `Config`.

UPLOAD_FILE_STORAGE_PATH, which is derived from the storagePath element in the file upload remote connection in your model.

Note, that currently only images can be delivered as the download servlet assumes the content type to be image/jpeg. This could be changed in the future by storing the correct content type during upload and retrieving it in this servlet.

4.5 Web Services

Similar to the generation of the stateless session beans, a web service is only generated for those entities that are used in at least one remote content provider. Those web services provide simple access to entity data. Additionally, some static web service are generated that are used for specific features. Those are explained in the following.

4.5.1 Calls to External Webservices

As a simple way to interact with external services, a web service CallExternalWebServiceWS in the backend allows to call another web service, that might be on a different system or server. The web service in the backend provides a method that takes a JSON-encoded object as an input. This object must contain the URL, the REST method type and the set of parameters of that method. For example, the following listing depicts how such a JSON object is constructed in map.apps.

Listing 10: JSON-encoded object containing information to call an external web service

```
data: json.stringify({
  "url": this._url,
  "requestMethod": this._method,
  "queryParams": this._queryParams,
  "body": this._bodyParams
})
```

queryParams, as well as bodyParams, are basically key-value pairs. However, the Java library used for the REST endpoint was not able to map these key-value pairs to a Java HashMap. For that reason a workaround was introduced: Instead of a single JSON object containing all key-value pairs, an array of JSON objects needs to be transmitted, where each object contains exactly one key-value pair. On the backend, these are mapped to an ArrayList<CustomHashMapEntry>. The class CustomHashMapEntry (and, therefore, each JSON object) consists of the attributes key and value.

4.5.2 Offer Webservices to Start Workflow

Besides the possibility to start a workflow through an app it is possible to invoke it using a webservice. The description of the corresponding model language is described in Section 4.2. For each invokable

workflow element a webservice is created and for each invoke definition a webservice endpoint is specified, including the defined parameters and the creation of the required entities. After the entities are saved using the internal beans, a `workflowState` is persisted using the workflow element the webservice belongs to. Additionally, the `lastEventFired` is set to the defined text specified in the workflow model after the `invokable` keyword, or to a default if not specified. The entity IDs returned by the internal beans are then injected as the content provider IDs. Directly afterwards, the workflow instance is accessible within the list of open issues of all app that are allowed to view the invoked workflow element. Since a new workflow instance is created, the backend is creating a new random UUID for each webservice call.

For each endpoint a method `@POST` or `@PUT` can be defined. The used parameter types are `@FormParam`. The path which has to be used to call the webservice endpoint consists of the workflow element name and the specified path in the invoke definition.

4.5.3 Event Handler

For the communication across apps, the backend offers an event handler web service. This web service handles all workflow events that are fired in one app and need to start a workflow element in another app. Required parameters for this web service are

- the instance ID of the workflow instance,
- the event which was fired,
- the content provider IDs,
- and the current workflow element which fired the event.

The event handler web service uses these parameters to perform adjustments in the workflow state of the current workflow instance. This includes setting the last event fired and the current workflow element. Furthermore, the content provider IDs are stored in the workflow state so that subsequent apps can load data from content providers using these IDs.

4.5.4 Workflow State

The workflow state web service allows to retrieve open workflow instances or add new ones. Whenever the list of open issues is opened in an app, this app sends its name to the workflow state web service. The web service then returns all workflow states whose current workflow element is part of the app with the given app name. For this purpose, the belongingness of workflow elements to apps is originally derived from the DSL model and stored in a hashmap in the backend.

Furthermore, for every new workflow instance, a new workflow state needs to be created. To do so, the workflow state web service is called as soon as the app which started the workflow hands the control over to the backend. This app generates a globally unique identifier (as described in Section 4.1) and provides it in the web service call.

4.5.5 File Upload

As another web service, a REST endpoint for uploading files is provided. In contrast to the other web services, it expects an input format of `MULTIPART_FORM_DATA`, thus allowing image uploads from HTML forms.

Given an uploaded file, it creates a file with a unique file name using the `File.createTempFile()` interface. The file is stored in the location specified in `Config.UPLOAD_FILE_STORAGE_PATH` (or `storagePath`) and the generated file name is returned to the invoking client. No further information about the file is stored or checked, i. e. original file name and content type are lost.

4.5.6 Version Negotiation

This web service can be used by generated apps to check whether they were generated from the same model version as the backend. Consequently, this is only useful if the modeller updates the model version after making changes to the data model.

5 Preprocessor

For each target platform a generator class is created that implements the interface `IPlatformGenerator`. Each generator class is registered in the `MD2RuntimeModule` and gets injected into the framework automatically. During modelling the Xtext Builder participant starts the building process every time a model file is changed and regenerates the platform files. Prior to generating the source code, the model gets transformed and is passed on to the implementing generator classes. Basically, this step can be considered a model-to-model transformation with the aim to simplify the generation process.

5.1 Model Simplification

When MD^2 models are defined across multiple files, the resulting model will also be split across multiple `MD2Models`. To avoid that these models have to be recollected over and over again throughout the preprocessing process, all controllers, models, workflows and views are combined into single model prior to the actual preprocessing.

5.2 Autogenerator

The autogenerator is a feature that allows to easily create view elements from model definitions. During the preprocessing, the references to content providers are resolved and content elements are created based on the model attribute types that the content provider declare. The schema for generation is as follows:

- `IntegerType, FloatType, StringType` become text input fields

- `DateType, TimeType, DateTimeType` become text input fields with corresponding time type attribution (later depicted as date/time pickers)
- `BooleanType` becomes a checkbox field
- `ReferencedType`

  `Enum` becomes an "option input" field

  `Entites` are processed recursively and all elements are wrapped in a "flow layout pane"

5.2.1 Remarks

With each content element, a new `MappingTask` is created that maps the content element to the attribute provided by the content provider. For this purpose a "custom action" called autoGenerationAction is created that the platform generators need to parse manually. If the view element is unmapped on startup or a user-specified mapping is found, the auto-generated mapping is removed.

In case the model attribute, from which a content element is created, has the optional parameters `name` or `description` set, these values are converted to label and tooltip representations for text inputs, option inputs and checkboxes. If no name is given by the modeller, the ID of the content element will be assigned as a label name with each uppercase letter preceded by a whitespace (e. g., "myAddress" will be transformed into "My Address").

5.2.2 Cloning and References

The MD^2 language allows the modeller to define certain view elements once and then reuse them multiple times. Internally, these elements are copied and references pointing to them are resolved during the pre-processing. The same behavior applies to view elements that are referenced in a view container (e. g., the flow layout pane) but are defined outside of this container. The following example shall serve to illustrate these two use cases.

```
TabbedPane mainView {
  customerPane -> customerView(tabTitle "Customer")
  generalPane -> generalView(tabTitle "General")
}
FlowLayoutPane customerPane(vertical) {
  headerPane
  TextInput myTextInput
}
FlowLayoutPane generalPane(vertical) {
  headerPane
  // Some view elements
}
FlowLayoutPane headerPane {
```

```
  Image logoImage("./capitol.png")
}
```

The code excerpt shows a tabbed pane that references two container elements. These containers again reference the same sub-container twice. After preprocessing the model appears to the code generators as follows:

```
TabbedPane mainView {
  FlowLayoutPane customerView(tabTitle "Customer", vertical) {
    FlowLayoutPane headerPane {
      Image logoImage("./capitol.png")
     }
    TextInput myTextInput
  }
  FlowLayoutPane generalPane(tabTitle "General", vertical) {
    FlowLayoutPane headerPane {
      Image logoImage("./capitol.png")
    }
  }
}
```

5.2.3 Resolving References

Because view elements can be reused and referenced across the whole model, MD^2 needs to provide means on how to use these virtual elements in behavioral elements like actions. The problem is that Xtext only allows to reference the originating element, but not the reference itself. In the example provided this means that only `headerPane` can be referenced, but any `headerPane` in a `customerPane` or `generalPane` can not, because there is no such element during runtime. To solve this issue MD^2 has the language type `AbstractViewGUIElementRef` that deals with these kind of references. This element allows to chain references to any element of the model in an arbitrary order. However, the scope has to be restricted to offer just the possible elements during autocompletion. During preprocessing the pseudo-referenced elements are converted into real elements and any reference to them will be resolved.

The reference to `myTextInput` that resides in `customerPane` and is part of the tabbed pane `mainView` can be referenced as `mainView.customerView->customerPane.myTextInput`. Basically, two elements are chained with `->` as the delimiter.

The modeller also might want to reference an auto-generated content element. This is done by referencing the autogenerator followed by the model element in square brackets. In the scenario modelled in Listing 11 the text input field that is generated for the customer's first name shall be referenced. The reference will then be `myPane.customerAutoGenerator[Customer.firstName]`.

Listing 11: Model using automatically generated view elements

```
FlowLayoutPane myPane {
  AutoGenerator customerAutoGenerator {
    contentProvider customerContentProvider
  }
}
contentProvider Customer customerContentProvider {
  providerType default
}

entity Customer {
  firstName: string
}
```

5.2.4 Remarks

When a view element, which is referenced or reused at a different place, is copied, each event binding, mapping or validator binding pointing to this view element will be copied as well. The copied version of the controller elements will be redirect to the copied version of the view elements. This does not work the other way around: any behavior element pointing to a referencing view element does not apply to the original element.

5.3 Validators

To ensure proper data integrity and to validate user input, each content element can be attributed with validators in a ValidatorBindingTask. When a view element is mapped to a model attribute some validators can be inferred automatically based on the attribute type (e. g., for integers) and its parameters (e. g., optional attributes). In these cases a validator is automatically created and bound to the view element in question. However, the modeler can still overwrite or unbind these validators in any actions that are performed upon startup.

5.3.1 Type-Specific

- `IntegerType` enforces a StandardIsIntValidator being bound
- `FloatType` enforces a StandardIsNumberValidator being bound
- `StringType` enforces a StandardStringRangeValidator being bound

5.3.2 Type Parameter-Specific

- Omitting the `optional` keyword will result in a StandardNotNullValidator being bound

- Setting `min` or `max` values for any kind of attribute will result in an appropriate validator being bound that ensures the range for this data type

5.4 Replacements

5.4.1 Enums

The MD^2 language allows to declare enums explicitly and implicitly. Internally all enums will be treated as explicitly defined enums and converted accordingly.

Before:

```
entity User {
  gender: {"male", "female"}
}
```

After:

```
entity User {
  gender: User_Gender
}
enum User_Gender {
  "male", "female"
}
```

5.4.2 Process Chains

Nested process chains will be flattened.

Before:

```
processChain pc1 {
  step firstStep:
    view mainView.customerView
  step nestedStep:
    subProcessChain pc2
  step thirdStep:
    view mainView.resultView
}
processChain pc2 {
  step secondStep:
    view mainView.InputView
```

```
}
```

After:

```
processChain pc1 {
  step firstStep:
    view mainView.customerView
  step secondStep:
    view mainView.InputView
  step thirdStep:
    view mainView.resultView
}
```

5.4.3 CombinedAction

The sole use of combined actions is to trigger execution of other actions. This behavior can also be achieved by using call tasks in custom actions. Hence, all combined actions will be converted to custom actions internally.

5.4.4 Miscellaneous

Some minor adjustments are made to the model:

- Check for existence of the `flowDirection` parameter for all flow layout panes.
- Duplicate spacer according to the specified number of spacers.
- Replace all named colours by their hex colour equivalents.
- Replace custom validators with standard validator definitions.

5.4.5 TestGenerator

Usually model transformations are transparent to the modeller and even for the developer hard to trace. With the test generator, though, there is a way to get a glimpse of the model's state as XMI definition before it gets passed to the platform generators.

6 map.apps Generator

The functions which generate map.apps source code are grouped into classes based on the kind of file they create. All map.apps generator code is located in the package `de.wwu.md2.framework.generator.mapapps` and written in Xtend. Furthermore, the class `util.MD2MapappsUtil` contains a few extension methods that are used in multiple generator classes.

As an entry point, the `MapAppsGenerator#doGenerate(fsa)` method is called from the development environment. It cleans the target directory and prepares for creating each app defined in the model.

For each app, its relevant paths and names are defined. Afterwards, the generator invokes the `generateWorkflowElementBundle` for each workflow element to generate individual bundles. Afterwards, three further methods of this class are called, generating a model bundle, a content provider bundle, and a workflow bundle as bundles for the app.

Last but not least, every generated app is packaged into a .zip archive, to enable uploading the app into a production map.apps system.

6.1 AppClass

This class is responsible for creating the `app.json` as described in Section 3.2.5. Furthermore, it creates the `bundles.json` file which maintains references to all local bundles of an app.

6.2 ModuleClass

This class specialises in creating the `module.js` file for all modules. It is therefore invoked by the `MapAppsGenerator` for each module, but with slightly different produced templates.

6.3 EntityClass, EnumClass, and ModelsInterfaceClass

These three classes are called while creating the models bundle for an app. The classes specialise in the kind of model that is created.

6.4 ContentProviderClass

Local and remote content providers are created by this class and put into the corresponding bundle of the app.

6.5 EventHandlerClass

This class generates the logic of the workflow event handler, which decides whether a combination of event and sending workflow element should cause another, local workflow element to be started. If that is not the case, the workflow will be terminated locally and all relevant data is transmitted to the backend, thus creating an entry in the list of open issues of another app.

6.6 Expressions, CustomActionClass, and CustomActionInterfaceClass

The last three classes are responsible for creating logic code for the workflow elements. This includes the generation of JavaScript code for actions that are run during the initialisation of an app or when mapped actions are activated.

Note, that process chains are not considered here, since the preprocessor has turned them into actions, which are handled by the `CustomActionClass` accordingly.

The `Expressions` class is used to store intermediate values used inside the actions, such as entity attribute values or comparisons of values. It is also used for storing literal values, which were hard-coded into the model, in a variable for use in later comparisons.

A Known Issues and Suggestions for Future Development

Since the generators for iOS and Android were not further development, they need to be adapted in the future in order to be compatible with the new version of the DSL and support all new features. Special attetion should be paid to the former workflows which were renamed as process chains during the course of the project seminar. In the iOS and Android implementations, these naming adjustments also need to be performed.

Furthermore, the division into workflow elements allows for a very modular architecture of the created apps. To exploit this advantage, a renaming of workflow elements should be allowed in the model in a way that different apps can use the same workflow element using different names for them. This way, it would be possible to determine very precisely which workflow element in which app is to be started rather than starting a workflow element which can be processed by any app that has the respective workflow element assigned.

Another possible extension is to allow return values when external webservices are called. This would enable programmers to include almost arbitrary functionality in the model without the need to offer it as explicit construct in the DSL. The only requirement is that return values comply with the data types supported by the MD^2 framework to ensure that they can be used in a purposeful way.

In addition, there are still improvement possibilities in the DSL using validators. One possibility is to check whether the fire event action in a workflow element is the last action and warn if it is not. This can be helpful, for example, when a save action follows a fire event action. In this case, saving data will not be performed, since the fire event action immediately forwards control to the next workflow element. Another possibility is to throw warnings or errors when a controller maps something to view elements which are never used by the controller, e.g. mapping something to a view which only appears in a different app. Finally, in the case that several content providers are used, a validator can check whether all content providers are saved, in order to ensure that a modeller does not forget any save actions.

Other possible features are

- call other external applications apart from REST,
- extend the location features (rather map.apps specific):
  display locations on a map,

convert a click on a map into location data,

convert coordinates into an address,

- allow the definition of custom icons for startable elements,
- support white spaces in project names,
- support temporary offline usage,
- generate only relevant content for each app, e.g. only generate entities which are used by the apps,
- provide build scripts for MD^2,
- provide auto formatting in the IDE,
- access foreign apps such as the phone, camera and GPS.
- customizable columns within the list of open issues.

B Further Improvements to your Development Environment

1 Reference a Generated App in the Development Project

By using a symlink, a running NetBeans instance will automatically notice changes to the generated app. Consequently, if the Jetty server is running, the newly generated app will automatically be published and made available in the browser.

1. Open a terminal and navigate to the map.apps NetBeans project directory (e.g. the extracted `sample ProjRemote` from step 1a in Section 1.2).
2. Navigate to the `apps/` directory using `cd src/main/js/apps`.
3. Create a symbolic link using an appropriate command (where `<PROJECT_NAME>` is the name of your MD^2 Project in Eclipse, `<ECLIPSE_PROJECT_LOCATION>` is its location, and `<APP_NAME>` is the name of the generated app(s)):
   a) Windows:
   ```
   mklink /j <APP_NAME> <ECLIPSE_PROJECT_LOCATION>\<PROJECT_NAME>\src-gen\
   <PROJECT_NAME>.mapapps\<APP_NAME>
   ```
   b) Linux / OS X:
   ```
   ln -s <ECLIPSE_PROJECT_LOCATION>/<PROJECT_NAME>/src-gen/<PROJECT_NAME>.
   mapapps/<APP_NAME> <APP_NAME>
   ```
4. Repeat step 3 for every generated app that you would like to have refreshed automatically.

2 Jetty: Allow Serving of Symbolically Linked Files

On Linux/OSX, Jetty by default does not serve symbolically linked files due to security concerns. To override this setting (which is not recommended in a production environment), put the provided file `jetty-web.xml` into the folder `/src/test/webapp/WEB-INF/` of your map.apps project.

C Backend Connection Specification

1 Resource Paths

Format:

VERB – Path – Request body <Status> - <Response body>

Entities

Load

```
GET - /<entity.name>/?filter=<filter>                    200 OK - List<Entity>
GET - /<entity.name>/first?filter=<filter>      200 OK - Entity or 404 NOT FOUND
```

Save

```
PUT - /<entity.name>/ - List<Entity>     200 OK - List<{ “”__internalId: <id> }>
```

Delete

```
DEL - /<entity.name>/<id>                                  200 OK or 404 NOT FOUND
```

Remote Validations

```
GET - /md2_validator/<remoteValidator.name>/ - Entity
                                                    200 OK - ValidationResult object

GET - /md2_validator/<remoteValidator.name>/?attrName1=content&attrName2=content
... &attrNameN=content
                                                    200 OK - ValidationResult object
```

attrNameX is a fully qualified name, having
`contentProviderName.path.to.attribute`

Filter Parameter

```
not <Attribute> (equals|greater|smaller|<=|>=) (<Int>|<Float>|<String>|<
    InputField>)
((and|or)(not)? <Attribute> (equals|greater|smaller|<=|>=) (<Int>|<Float>|<
    String>|<InputField>))*
```

Resource for Model Version Checks

The model version should be checked by the apps for all remote connections. Requests are only valid if the server accepts the current model version.

```
GET /md2_model_version/current                                  200 OK - <version>


GET /md2_model_version/is_valid?version=<version>
                                                  200 OK - { "isValid": (true|false)}
```

2 JSON Format Conventions

List<Entity>:

```
{
  "entityName": [
  {
    "attribute": <Value type see below>,
    [...]
  },
  {
    "attribut": <Value type see below>,
    [...]
  } [...]
  ]
}
```

Having <Entity> = Entity without root node

Entity:

```
{
  "entityName": [
    "attribute": <Value type see below>,
    [...]
  }
```

```
}
```

Validation Result:

```
{
  "ok": (true|false),
  "error": [
  {
    "message": "Allgemeine Fehlermeldung",
    "attributes": ["attribut1", "attribut2"]
  },
  {
    "message": "'cant be blank",
    "attributes": ["forename", "surname"]
  }
  ]
  [...]
}
```

Mapping for data types (language data type -> JSON type for attribute values)

```
Enum -> Int (index of the currently selected value)
Int -> Number

Float -> Number
<Everything else> -> String
Date -> String im Format yyyy-mm-ddThh:mm:ss+hh:mm
```

3 Examples

GET `/customer/first` returning one customer

```
{
  "customer": {
    "__internalId": "0",
    "firstName": "Ulrich",
    "lastName": "M\u00c3\u00bcller",
    "membership": "1",
    "professionalCategory": "0"
```

```
  }
}
```

GET `/customer` returning multiple customers


```
{
  "customer": [
  {
    "__internalId": "0",
    "firstName": "Ulrich",
    "lastName": "M\u00c3\u00bcller",
    "membership": "1",
    "professionalCategory": "0"
  },
  {
    "__internalId": "0",
    "firstName": "Hans",
    "lastName": "Dampf",
    "membership": "1",
    "professionalCategory": "0"
  }
  ]
}
```