MD²
Model-driven Mobile Development

Group members:
Sören Evers
Klaus Fleerkötter
Daniel Kemper
Sandro Mesterheide
Jannis Strodtkötter

Tutors:
Steffen Henning Heitkötter
Dr. Tim Alexander Majchrzak

Dr. Frank Köhne

Table of Contents

Introduction

Mobile development becomes more and more important as the usage of smart phones and tablets increases. There are different mobile operating systems requiring different apps. The traditional way to address this issue is the development and maintenance of one app per supported operating system. This approach yields high costs due to increased development and maintenance efforts. One alternative to solve this issue is to develop web apps that will be enhanced by an intermediate layer. The most prominent example is PhoneGap. Although the intermediate layer styles the apps to look like native app, the user will still encounter differences in the look and feel compared to native apps. MD² presents a new approach. MD² uses model driven mobile development to combine the advantages of both other approaches. MD² defines a domain specific modeling languages which is used to model an app. The domain specific language abstracts from platform specific concepts and allows easier and faster development of apps in a declarative way. Based on the modeled app the generators construct native apps for each of the target platforms. Thereby the developer has to maintain only one app, more precisely one model of an app, but can enjoy the benefits of native apps.

The task of this project was to proof the concept of model driven mobile development. Therefore the goal was neither to support all mobile operating systems nor to provide the ability to develop all possible kinds of apps. The goal was to proof the ideas behind model driven mobile development on their practicability. Therefore, MD² targets Android and iOS as mobile platforms and the functionality is based on a use case of the insurance area. It should be possible to generate tariff calculators. Saying this a bit more generally, with MD² it should be possible to model and generate apps, that can be used to display data to the user, collect data from the user and communicate with backend servers that perform the further processing of the data.

In the context of MD² two kinds of developers are required: App modelers and framework developers. The app modelers are the end users of the MD² framework and use MD² to model and generate apps. The third chapter of this documentation is targeted on modelers and can be used as a starting point and hand book to model apps with MD². Framework developers on the other hand develop and maintenance the framework itself. They introduce new features and fix bugs in both the language and the generators. The fourth chapter should be used by them to get to know the architecture of the MD² framework, the thoughts behind design decisions and how the different parts work. Additionally they should read the third chapter as well because the basic ideas and explanations of all language elements are required prerequisites for them as well. Besides those two technical chapters there are more general chapters in this documentation. The next chapter will introduce the approach of the project seminar used to develop the framework. Chapter five evaluates the ideas behind MD² and chapter six concludes and provides an outlook.

Approach

General

The development of the MD² framework consisted of basically three steps. First the general architecture had to be specified. Based on the architecture the language had to be written. The language provides, beside a grammar to which modelers have to comply, the metamodel of models of apps that can be modeled with MD². This metamodel is required for the third step, the development of the generators.

To develop the generators, we chose a prototype based approach. This means that we first developed prototypical apps directly for Android and iOS. These prototypes should be representative and comply with the general architecture. Later on these prototypes have been used to develop the generators. Therefore we first analyzed the prototypes on parts that are independent of a certain model and will always be the same. We transferred these parts to libraries. The other parts built the basis for the development of the generators. We took one part after another and transformed them into generatable code by identifying the model dependent pieces and letting them be generated based on the input model.

After knowing the building blocks of the development we structured how we will tackle them. First we defined the general architecture with the whole team, because this is the basis of the framework. Then we divided the team in three groups and approached the development of the language and the prototypes for Android and iOS in parallel. After these three steps have been finished we developed the two generators. Each of them was based on the language and the respective prototype. Besides the two prototypes we found a third task that had to be done. Since the language, and therefore the metamodel, became larger and more complicated as initially assumed, we introduced a preprocessing step. The preprocessing transforms a model, as defined by a modeler, to a version that is better suited for the generation. So we had again three tasks to be performed in parallel and therefore split our team respectively into three groups.

Tools

The tools that we used were mostly predefined. As development environment for all the development tasks we used Eclipse, except for the development of the iOS apps, for which we had to use Xcode. The development of the apps has been done with the recommended tools. To develop the language we used Xtext, which has been suggested to us by our tutors.

The only decision we had to make was on the tool to develop the generators with. We had two options. We could either use Xtend or Acceleo. The advantages of Xtend are a better integration with the Xtext framework, it being more flexible because templates and behavior are handled in

methods and Xtend uses a Java like language that provides many features to facilitate generator building. Drawbacks are no support for manipulation of generated code and that it uses a general purpose language which leads to a not standardized frame. The benefits of Acceleo on the other hand are a good integration in Eclipse and support of development functionality as debugging, tracking and tracing, support of protected regions that allow comfortable manipulation of generated code, native generator building and it applies to the MOF Model to Text Transformation standard as specified by the OMG. Drawbacks of Acceleo are that it is more static since it is template based and difficult to integrate in a multi model environment.

The powerfulness of the two generators is comparable. In our case the available Java know-how in the team outweighed the advantages of the better UI of Acceleo due to the limited time. For a longer-lasting project it might make sense to build-up the know-how and benefit from the better UI integration. E.g. both, Acceleo and Xtend do not provide functionality to copy static files. However, in Xtext we were more used to the code basis and thus it was easy to provide this feature. In Acceleo much more research would have been necessary to solve this problem. Protected regions may come in handy, but we had and have not really seen the urgent need for this feature and it was never formulated as a compulsory requirement.

Modeler's Handbook

Getting Started

Setting up an environment

The MD² framework will be deployed as Eclipse plugin. Therefore it is shipped as a set of three jar files. To "install" this plugin simply copy the three files into the plugin folder of your Eclipse installation folder. The Eclipse installation folder is usually located in `C:\Program Files\eclipse`. Every time a new version of the framework will be deployed, these files have to be replaced.
The MD² plugin is based on Xtext. So Xtext has to be installed for the plugin to work. The installation of Xtext is done by using the Install New Software wizard of Eclipse. To start it choose `Help -> Install New Software...` from the menu bar and `Add...`. Use http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/ as `Location`. From the appeared list choose `Xtext` in the latest version and in the sublist choose `Xtext Runtime` and `Xtext UI`.

Live Coding Example

In the folder `Screencasts` the live coding example can be found. It is a set of the following nine screencasts:

- 01 Introduction
- 02 Create project
- 03 Model
- 04 First View
- 05 Controller
- 06 Run first app
- 07 Second View
- 08 Extended Controller
- 09 Run enhanced app

Additionally the file `liveApp.zip` contains the source code, that has been developed during the screencasts.

Deploying an MD² App on Android

A common Eclipse-based development environment is recommended to debug, run or package your generated Android app. We suggest using your already set-up Eclipse installation and upgrading it. Therefore start the Install New Software wizard, add https://dl-ssl.google.com/android/eclipse/ and select `Developer Tools`. After the Developer Tools have been installed via the `Window` menu the `Android SDK Manager` and the `AVD Manager` can be startet. MD² currently supports API level 11 / Android 3.0. The corresponding SDK should be installed by using the `Android SDK Manager`. If you want to test your App on your local machine using Android's emulator, create an virtual Android device using the `AVD Manager`.

Within your MD² project, you'll find a complete Android project for Eclipse in `src-gen/<app.package.name>.android`. Import and run it like any other Android project by following these steps:

1. Choose `File > Import... > Existing Project into Workspace`
2. Choose the option `Select root directory`, select the folder `<Path to your modelling project>src-gen/<app.package.name>.android` and click `Finish`
3. While building the workspace, the Android ADT plugin takes a short while to perform additional tasks, including generating the `gen` source folder.
4. Start the app by choosing `Run > Run` or `Debug as... > Android Application`. If prompted, choose the virtual device you have created earlier, or a physical device connected via USB.

For further details on how to run, debug and publish Android applications, please refer to the chapter "Workflow" on Android Developers.

Please note that all JAR files within lib must be included in the build path and exported in order for the Dalvik compiler to see them. Normally, this is already set in the project settings and can be neglected. But bear this in mind if you experience any NoClassDefFound errors while deploying, especially if you want to use an external build tool.

Deploying an MD² App on iOS

In general, the apps generated by the MD² framework are easy to test and deploy. This comes from the fact, that for each app the according project file will be generated, which enables the direct startup of the Xcode project after the generation. This is simply done by double clicking the project file in the main folder of the generated app as depicted in Figure 1.
However, as with every iOS app, it can only be started on a Macintosh operating system and with Xcode installed.

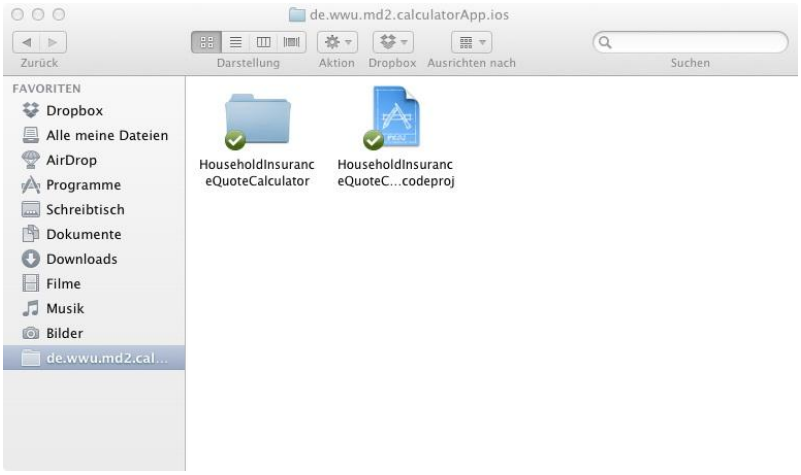


Figure 1: Project start-up by project file

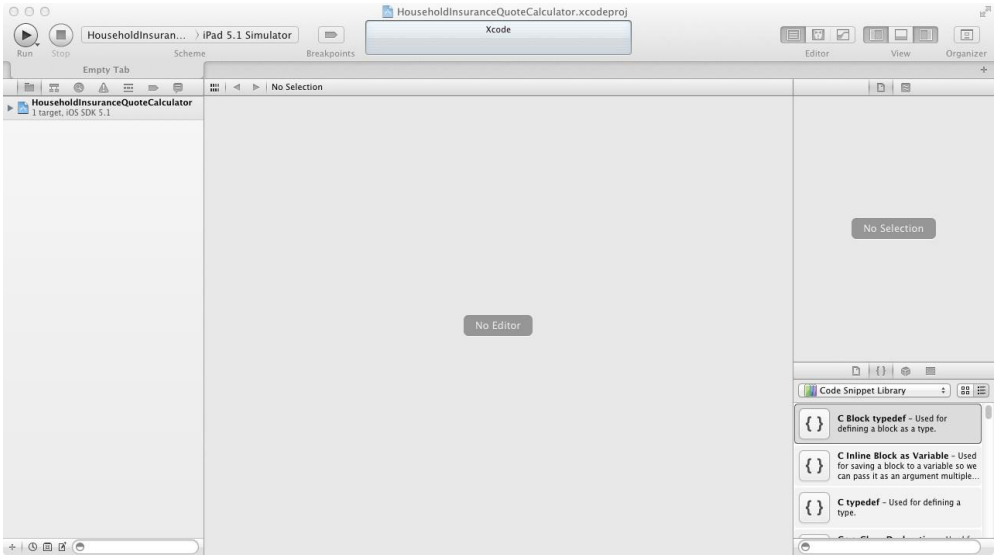


Figure 2: Started project in Xcode

After the app has been started, Xcode will appear completely configured, similarly than in Figure 2. The name of the project is the one defined in the main block as app name. Under the same name a folder has been created, which contains all necessary static and generated files in several sub folders structured by the Model-View-Controller concept. Figure 3 depicts the folder structure in all layers of the MVC concept and the supporting files additionally. The detailed structure derived from the architecture can be seen in the appropriate section.

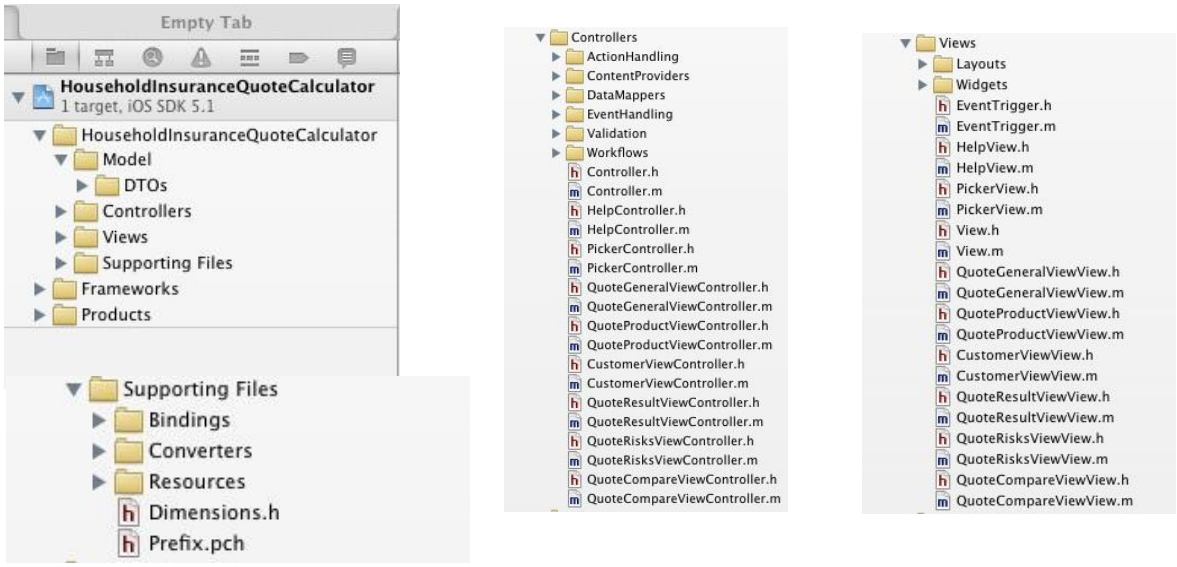


Figure 3: Folder structure of the model layer (left top) controller layer (middle), view layer (right) and supporting file (left bottom)

Additionally, the app can be deployed either on the simulator or on a specific hardware device. In order to change the deployment, the run configuration has to be changed. By clicking on the menu right next to the stop button, you can choose between an attached device, the iPad simulator or the iPhone simulator. In most cases the first two options should be sufficient. This configuration is depicted in Figure 4.

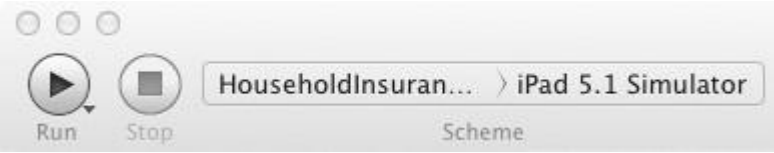


Figure 4: Run configuration for iPad simulator

Beginning with the start on the iPad simulator, you simply have to choose the second option from the run configuration and click on the run button. After that, the simulator appears with the started app, similarly to the HouseholdInsuranceQuoteCalculator app from Figure 5.

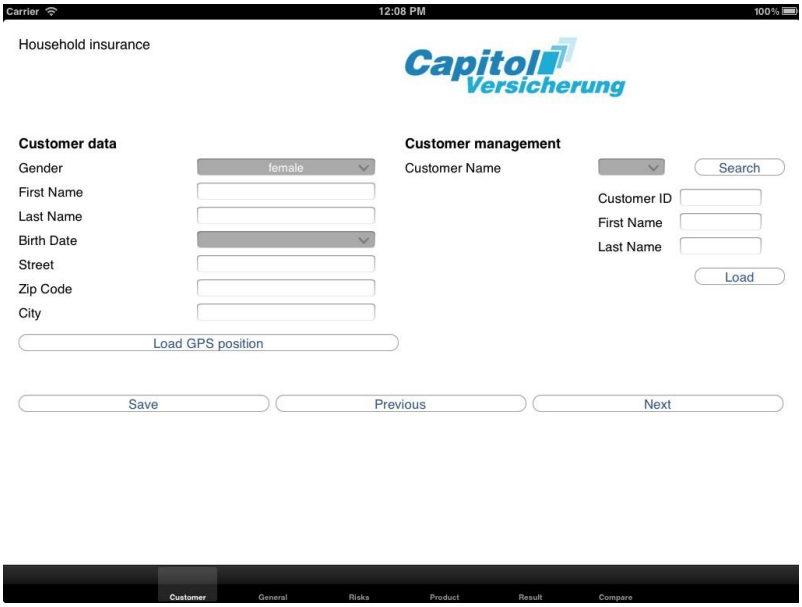


Figure 5: Started HouseholdInsuranceQuoteCalculator app on the iPad simulator

Another issue with the simulator is the occasional need to delete the local data the app automatically creates, even without local content providers. In order to clean the installation after the model has been changed, you have to delete the local data. As depicted in Figure 6 the appropriate app has to be clicked for a longer time until the app icons are shaking. After this the black x button has to be clicked and the deletion finally confirmed like it can be seen in Figure 7. The procedure is equivalent on the iPad hardware device, simply by exchanging clicks with touches.

Figure 6: Delete local data on simulator

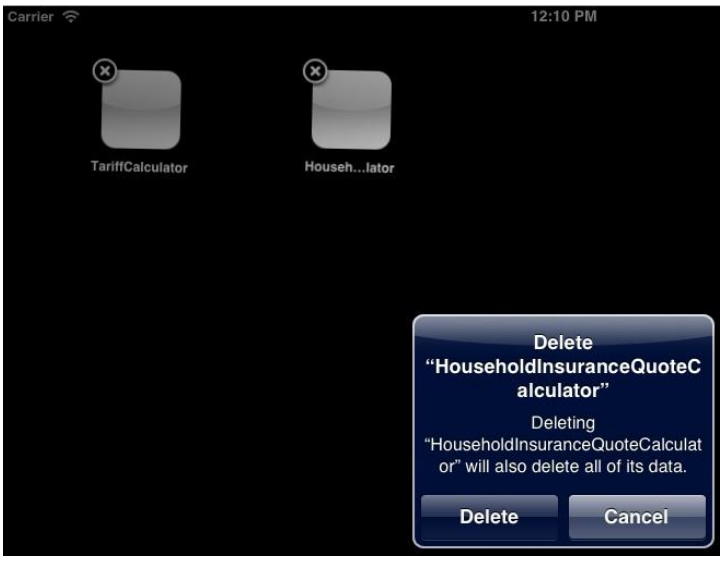


Figure 7: Confirmation to delete local data

Finally, the hardware device has to be registered for the specific licence that should be used for deployment, which can be seen in Figure 8. After this, it can be used and plugged in to the Macintosh computer via the USB port. As the device will be recognized by Xcode properly and displayed in the run configuration the first option has to be selected. Like in Figure 9 the first option changes to the actually plugged device.

Figure 8: Provisioning profile and device management

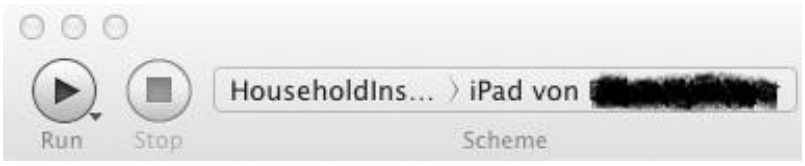


Figure 9: Run configuration for iPad hardware device

Here, the procedure to start up the app is similar to the iPad simulator so that it will not be discussed in detail here.

Language reference

The MD² framework enforces an MVC design pattern. This part sketches out the functionality covered by each component. The language uses keywords to specify components in the language; sometimes followed by a unique name that has to be assigned to that component so that other components can reference to it. Each component may have attributes to further specify its functionality.
The following formatting is used throughout this reference guide:
Language keywords: `bold typewriter font`
Attributes: `typewriter font`
References are underlined and named by the component they have to point to: `underlined typewriter font`
A common structure of an MD² file is as follows:

```
package PACKAGE_NAME

component NAME {
      attribute DEFINITION
      attribute STRING
      attribute OTHER_COMPONENT
}

component NAME {
      ...
}

...
```

Package Structure

All components in MD² are organized in a package structure that represents the MVC structure. All documents have to be placed in corresponding packages (views, models or controller). For example, all view files are expected to be in the package `any.project.package.views`. The package has to be defined in each MD² file as follows:

```
package PACKAGE_NAME
```

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

Model elements

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

Entity

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

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

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

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

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

Supported complex data types are:

- Entity
- Enum

Supported simple data types are:

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

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

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

- `float` supports
  `max FLOAT` – maximum allowed value of the attribute
  `min FLOAT` – minimum allowed value of the attribute
- `string` supports
  `maxLength INTEGER` – maximal length of the string value
  `minLingth INTEGER` – minimal length of the string value

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

Exemplary entity that represents a person:

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

Enum

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

Exemplary enum element to specify weekdays:

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

View elements

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

Container Elements

Grid Layout

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

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

For each grid layout at least the number of rows or the number of columns has to be specified. If only one of these parameters is given, the other is calculated by MD² on generation time. In case that both parameters are specified and there are too few cells, all elements that do not fit in the layout will be discarded. The following comma-separated parameters are supported:

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

FlowLayoutPane

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

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

The following comma-separated parameters are supported:

- `vertical` or `horizontal` (default) – flow direction
- `tabIcon PATH` – if the layout is a direct child of a TabbedPane, 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.

TabbedPane

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

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

Content Elements

Input elements

Input elements can be used to manipulate model data via mappings (see controller section). At the moment text inputs, dropdown fields and checkboxes are supported. All input elements support the optional attributes `label` and `tooltip` that can be used to create compound input fields with a label and a help text button added.

TextInput

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

```
TextInput NAME {
      label STRING
      tooltip STRING
      type <textfield_type>
}
```

Besides the tooltip and label attribute, a text field type can be specified to influence the appearance of the actual input field. The following text field types are supported:

- `default` - display a standard input field (this is the default)
- `date` - display a date picker
- `time` - display a time picker
- `timestamp` - display a combined date and time picker

OptionInput

Option inputs are used to represent enumeration fields in the model.

```
OptionInput NAME {
      label STRING
      tooltip STRING
```

```
        options Enum
    }
```

Besides the tooltip and label attribute, option inputs support the optional options attribute. This can be used to populate the input with the string values of the specified Enum. If options is not given, the displayed options depend on the Enum type of the attribute that has been mapped on the input field (see controller section).

CheckBox

Check boxes are used as a representation for boolean model attributes.

```
    CheckBox NAME {
        label STRING
        tooltip STRING
        checked BOOLEAN
    }
```

Besides the tooltip and label attribute, check boxes provide the optional attribute `checked` that allows to specify whether the checkbox is checked by default or not. This setting will be overruled by actual values loaded from the model.

Label

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

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

as well as this shorthand definition

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

The text can either be annotated as an explicit `text` attribute or the label text to display can be noted in parentheses directly after the label definition. The optional `style` can either be noted directly (cf. Style section) or an existing style definition can be referenced.

Tooltip

Display tooltip element. 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)
```

Button

Display button element. 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)
```

Image

Display image. 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 MD² project

- `height` (optional) - Height of the image in pixels
- `width` (optional) - Width of the image in pixels

Spacer

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

```
Spacer (INT)
```

Auto Generator and Entity Selector

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

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

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

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

Style

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

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

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

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

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

Element referencing and renaming

Elements can not only be defined where they should be used, but there is also a mechanism of defining 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. However, names have to be unique so that each element could only be referenced once. To avoid those name clashes a renaming mechanism had been implemented that allows to set new names for the actual copied element.

```
      Element -> NAME
```

Controller elements

Main

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

```
main {
      appName STRING
      appVersion STRING
      modelVersion STRING
      startView ViewContainer
      onInitialized Action
      defaultConnection RemoteConnection
      defaultWorkflow Workflow
}
```

The attributes are explained as follows:

- `appName` - The name of the application
- `appVersion` - a string representation of the current app version, e.g. “RC1”

- `modelVersion` (optional) - a string representation of the current model version that has to be in accordance with the model version of the backend
- `startView` - reference to any container element (cf. view) that should be used as the initial view after the app start-up
- `onInitialized` - reference to any action that should be executed directly after start-up, e.g. to define the initial mappings and validator bindings. A common use case is to refer to a combined action here that calls appropriate custom actions for the binding tasks.
- `defaultConnection` (optional) - a default remote connection can be specified here, so that it is not necessary to specify the same connection in each content provider
- `defaultWorkflow` (optional) - Exactly on e workflow can be active at the same time. Using this parameter it is possible to specify a workflow that is active right after the app start-up. It has the same effect as calling a SetActiveWorkflowAction in the onInitialized action.

Actions

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

CustomAction

A CustomAction contains a list of CustomCodeFragments where each CustomCodeFragment contains one task. For each type of task there exist a specific CustomCodeFragment that is distinguished by the keyword that introduces it. 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. Furthermore there are CallTasks that can call other actions.

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

Binding and unbinding actions

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.

```
bind|unbind action
      <CustomAction | CombinedAction | SimpleAction> <...>
      on|from
      <Container | Content> . <elementEventType> |
      <GlobalEventType> | <OnConditionEvent> <...>
```

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

- `NextStepAction` - proceed to the next Workflow step
- `PreviousStepAction` - go back to the last Workflow step
- `GotoStepAction (<WorkflowStep>, BOOLEAN)` - Change to the given Workflow step. The second parameter indicates whether an error message should be shown if the action fails.
- `GotoViewAction (<Container | Content>)` - Change to the given view element
- `DataAction (<AllowedOperation> <ContentProvider>)` - Perform a CRUD action (save, load, remove) on the given ContentProvider
- `NewObjectAction (<ContentProvider>)` - Creates a new object for the given ContentProvider
- `AssignObjectAction (use <ContentProvider> for <ContentProvider.Attribute, ...)` - Cross link two ContentProvider. The first parameter denotes the ContentProvider for the model element of a different ContentProvidewr defined in the second parameter
- `GPSUpdateAction (<GPSField | STRING> <...> to <ContentProvider.Attribute>)` - Links a GPS property (latitude, longitude, altitude, citystreet, number, postalCode, country, province) to a model element of a given ContentProvider
- `SetActiveWorkflowAction (<Workflow>)` - Changes the current Workflow

There are different event types available:

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

CustomEvent

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 can be defined recursively in one of the following ways. This is a simplified version of the grammar. For a comprehensive overview the grammar in the appendix can be consulted.

```
Boolean |
<Container | Content> equals not? <Container | Content> |
<Container | Content> equals not? <STRING | INT | FLOAT> |
is  not?  <valid|empty|checked|filled> <Container | Content> |
not? <Condition> and|or not? <Condition>
```

Binding and unbinding validators

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

```
bind|unbind validator
      <Validator> <...>
      on|from
       <ContainerElement | ContentElement> <...>
```

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

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

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

Mapping and unmapping view elements

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

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

CallTask

CallTasks call a different Action

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

CombinedAction

CombinedActions allow the composition of Actions.

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

Validator

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

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

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

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

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

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

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

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

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

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

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

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

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

The NotNullValidator makes the input field required.

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

RemoteValidator

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> <...>)
```

Workflow

A Workflow is used to define several steps in which the application can currently be. It is possible to define several Workflows. Workflows can be nested and there is at most one Workflow active.

```
workflow NAME {
      <WorkflowStep> <...>
}
```

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

Instead of the former mentioned settings, a Workflow can be referred to that will become active while this WorkflowStep is the current one.

```
step NAME:
      view <ContainerElement | ContentElement>
      forwardCondition { <Condition> }
      forwardMessage STRING
      backwardCondition { <Condition> }
      backwardMessage STRING
      forwardOnEvent forwardEvents <EventDef>
      backwardOnEvent <EventDef>
```

or to jump to a different Workflow

```
step NAME:
      subworkflow <Workflow>
```

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

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

ContentProvider

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

It allows to CREATE_OR_UPDATE (save), READ (load) and DELETE (remove) the stored instance. Which of those operations is possible is specified in allowedOperations. By default all operation 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.

RemoteConnection

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

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

Developer's Handbook

This section contains information for developers who want to extend the MD² framework source code. Furthermore, it also describes what needs to be regarded when implementing a back-end server for apps developed using MD².

Setup

General

First, setup your IDE and environment:

4. Install Eclipse (Indigo or newer). Preferably, use the Eclipse's JEE distribution.
5. Install via `Help -> Install New Software...` and http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/ Xtext and Xtend

Second, copy the sources to where you want to edit them. Check out the source repositories[1]:

1. Create an empty directory and within it:
2. Optional: Create a workspace directory and point Eclipse to it

[1] This guide assumes that the source code is stored in repositories of the VCS system GIT. But without a or with another VCS, the steps should be quite similar.

3. Clone the following repositories without changing their name
    - `framework`
    - For Android: `framework-lib-android`
    - For iOS: `ios-file-lister.git`
4. Optional: Add repositories to EGit using its repository view
5. Import the projects contained in these repositories using `File > Import... > Import Existing Project into Workspace`

Once your environment is ready, you can start altering the framework code:

1. `git pull` the latest changes
2. Run the workflow GenerateMD2.mwe2 (found at `src/de/wwu/md2/framework`).
3. Run the project as an "Eclipse Application" to launch a new Eclipse instance in which you can develop your MD² app.

Workspace & Requirements in Android

Generated apps rely on the runtime library `md2-android-lib.jar`. This jar contains all parts that do not rely on code generation. It is contained within the repository `framework-lib-android` and currently copied manually into the framework projects, from where it is copied as a resource into generated apps during their generation.

Building the runtime library is done using Ant. See the build.xml for details. The following targets are available:

- `ant compile` to compile the project
- `ant package` to just create a jar (Output folder is dist/)
- `ant release` to package and copy the result into the framework project. Be sure to adhere to the directory structure described in Installation.

You can run these targets from Eclipse (This requires a JDK7 to be present under `JAVA_HOME` on the `PATH`) :

- Select `build.xml`
- Click `Run -> Run as... -> Ant build...`
- Within the dialog, select the target (see above). You'll want 'package' to create the jar or 'package'
- Run it.

Workspace & Requirements in iOS

As already said the basic requirement to develop iOS applications and extend the framework by further static classes in Objective-C is a computer that runs a Mac OSX operating system. Here, it has to be considered that only versions above 10.6 contain the Apple Software Development Kit (SDK) and, thus, are able to develop iOS and Mac OSX applications with Xcode. Since now, the most sophisticated and officially supported tool for development is Xcode, although there are certain less advanced compilers that can compile Objective-C code.

FileLister

In order to add new static files to the framework the so-called FileLister has been developed, which collects all files that are marked as static from a certain path and puts them in an own folder. Figure 1 shows the static file tagging for an exemplary file.

```
// static: ActionHandling
//
//  AppDelegate.m
//  TariffCalculator
//
//  The AppDelegate represents the central hub for the whole application.
//
//  Created by Uni Muenster on 30.05.12.
//  Copyright (c) 2012 Uni-Muenster. All rights reserved.
//
```

Figure 1: Static files tagging on iOS

Before the FileLister can be executed for the first time, it has to be configured properly in order to point to the correct path the static files are collected from. Therefore, the run configuration has to be changed by right-clicking on the project, selecting "Run As" and clicking "Run Configurations...". This procedure can be seen in Figure 2. After this a new window appears and the "Arguments" tab has to be clicked. After this an argument will be added, which contains the path in double quotes (e.g. "/Users/unims/Documents/Xcode Workspace/TariffCalculator/TariffCalculator") as depicted in Figure 3.

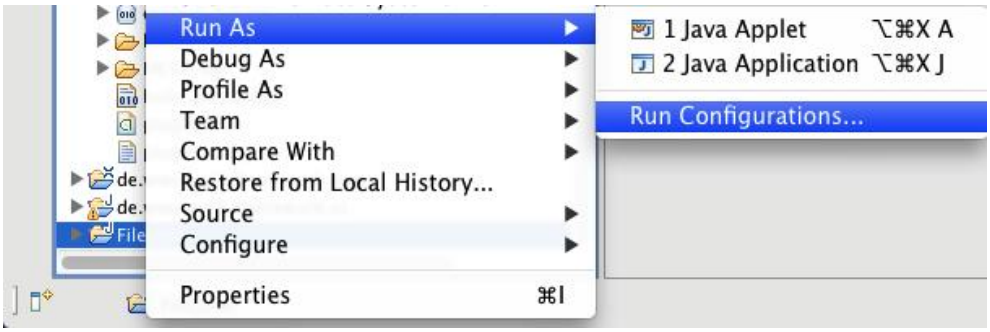


Figure 2: Configure the FileLister by its Run Configuration

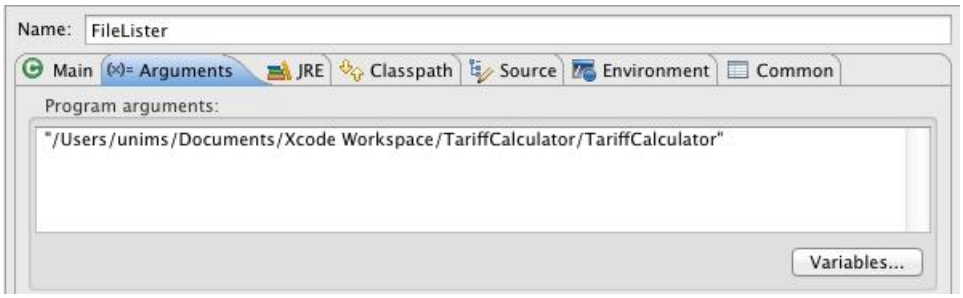


Figure 3: Set the path to the folder of the iOS static files (here: iOS prototype)

After the FileLister is correctly configured, it can be started. Therefore right-click on the project, select "Run As" and then click on "Java Application", which is described in Figure 4. During the FileLister is collecting all static files, the appropriate code that will be inserted later into the FileStructure is displayed on the console. Figure 5 shows an exemplary output of the FileLister. By

selecting all lines in the console, which begin with “StaticFiles.put”, right-clicking on the console view and clicking “Copy” (Figure 6) in the end the appropriate code for the FileStructure will be copied.

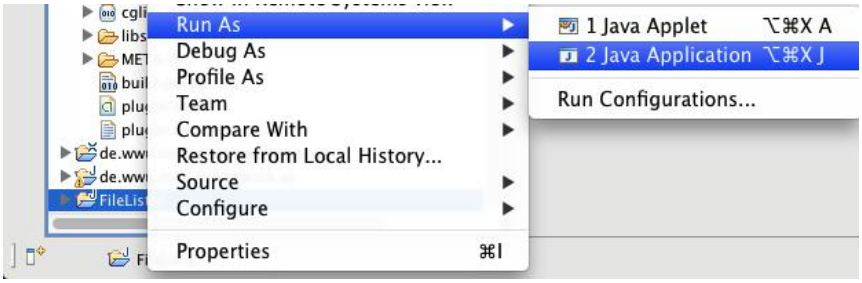


Figure 4: Start the FileLister to collect static files

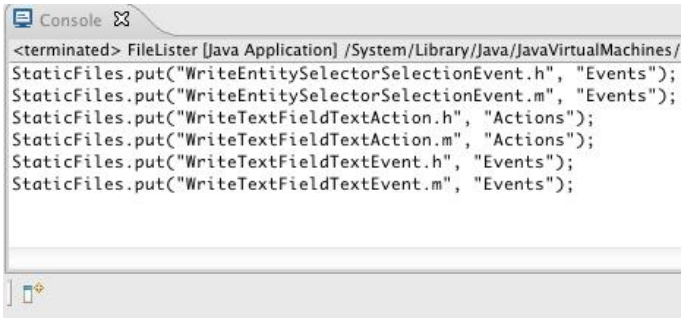


Figure 5: Output of the FileLister after execution

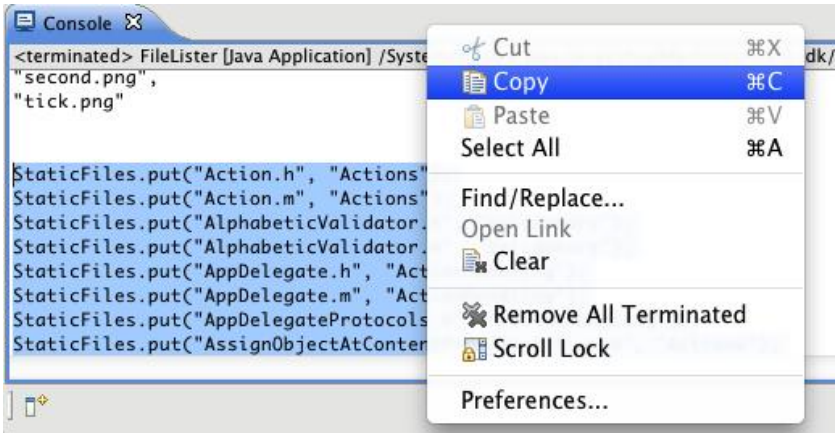


Figure 6: Copy the collected static files

FileStructure

After the static files have been selected by the FileLister these new files have to be registered to the framework. For this reason, the FileStructure class will be introduced that stores all necessary static, generated and resource files for the whole app. During the generation process it will be used to build the project file for the generated app, which contains references to all files. Like described in Figure 7 the previously copied code from the console has to be inserted. Therefore, the old code has to be selected and the new code to be pasted.

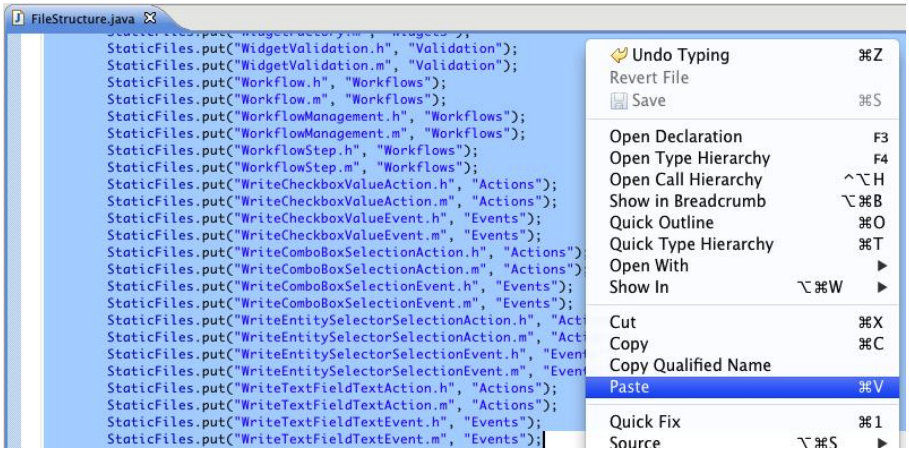


Figure 7: Paste the collected static files into the file structure class

Resources

In addition, the next step considers the update of the static file resources, which will be added to the generated app during generation. As Figure 8 describes, the old resources have to be deleted before. Afterwards, the static files collected by the FileLister have to be copied, as you can see in Figure 9. These will be contained in the automatically generated "static_files" folder and can be found under the path configured the run configuration (see Figure 3), e.g. "/Users/unims/Documents/Xcode Workspace/TariffCalculator/TariffCalculator/static_files".

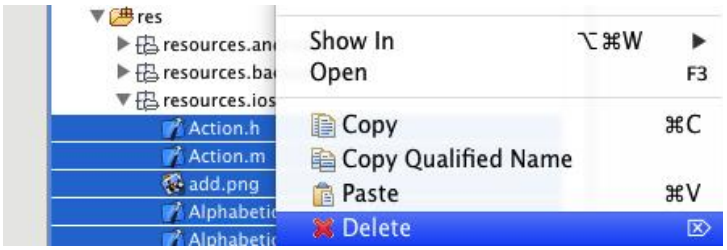


Figure 8: Delete the old static file resources from the framework project

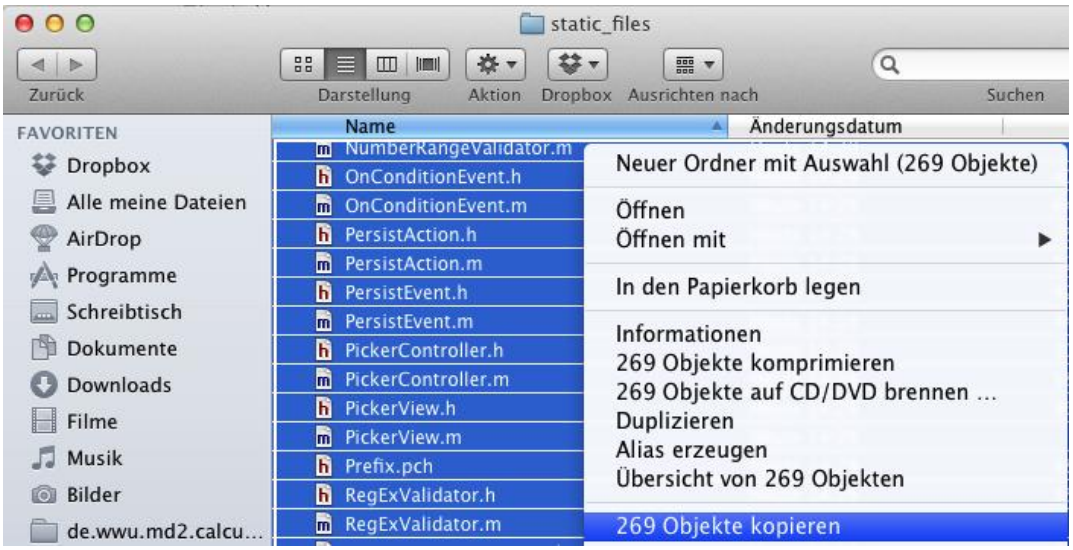


Figure 9: Copy the new static file resources from the "static_files" folder

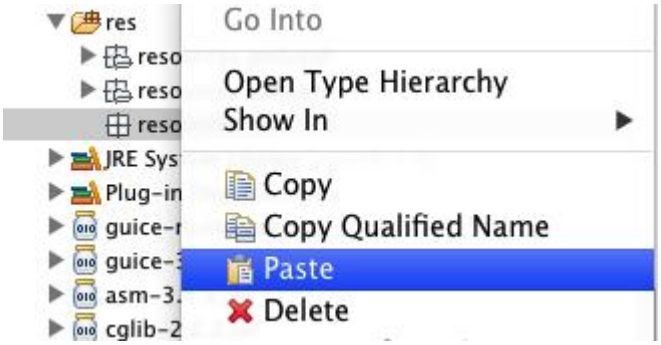


Figure 10: Insert the new static file resources to the framework project

Finally, in the last step the new static file resources will be added to the project, like depicted in Figure 10.

Exception breakpoints

Another issue that especially helps debugging the generated app is the exception breakpoint. In case of critical errors where the app is crashing completely this construct helps to find the source of the problem. With an exception breakpoint Xcode jumps to the line of code in the class at which the error occurs. Otherwise, Xcode jumps to the main class and the actual source is hidden.
Therefore, the tab selected in Figure 12 has to be chosen, clicked on the "+" button at the bottom, like described in Figure 11, and "Add Exception Breakpoint" has to be clicked.

Figure 11: Add an exception breakpoint to the generated app for debugging

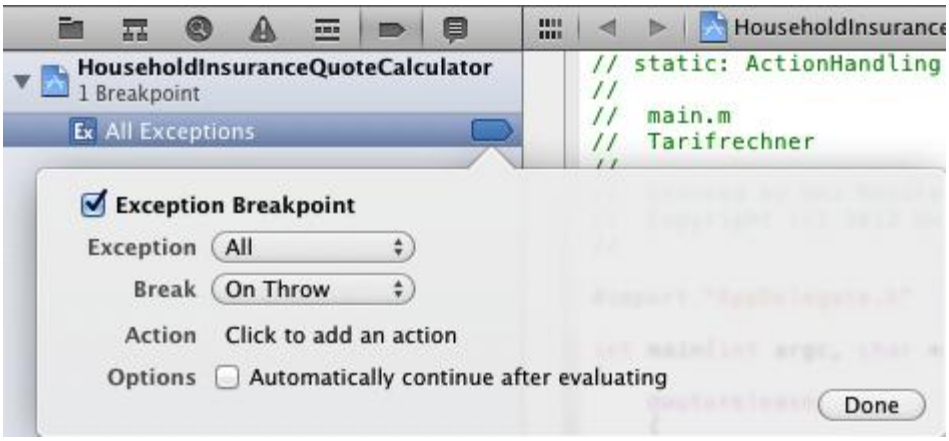


Figure 12: Configure the exception breakpoint

Afterwards, the window from Figure 12 appears and the exception breakpoint has been created and can be configured. Normally, the default configuration is sufficient and can be confirmed.

Deploying a new framework version

To deploy a new version of the framework, one simply has to select the main framework project `de.wwu.md2.framework` and choose `File -> Export...`. In the opened wizard select `Plug-in Development -> Deployable plug-ins and fragments`. In the next step all three framework projects have to be selected. Additionally the option `Directory` should be marked and the path to the folder, to which the plugin shall be deployed, has to be entered. To deploy the plugin now, one simply has to click the `Finish` button.

Implementing a server connection

Generated apps can communicate with a server using remote ContentProvider. The developers chose to employ RESTful JSON over HTTP as platform-neutral mode of transportation. It fits the aim of transporting entity instances back and forth and is easy to implement on the server-side. To further ease the server implementation, each generated app contains a stub implementation intended to serve as a facade. You can use and extend this project to implement a connection between the deployed apps and your existing application infrastructure. Otherwise, it is also possible to implement such an endpoint from scratch.

To start off, import the generated backend stub project in Eclipse:

1. Choose `File > Import... > Existing Project into Workspace`
2. Choose the option `Select root directory`, select the folder `<Path to your modelling project>src-gen/<app.package.name>.android` and click `Finish`

The project has the following IDE dependencies:

- JEE extensions

- Glassfish JEE application server[2] and plug-in (can be optained via Eclipse marketplace)

Among other contents, the project contains the following classes and resources:

- The package `<app.package.name>.backend.models` contains JPA-annotated, persistable versions of your entities.
- For each entity, a REST resource needs to be implemented. These are generated as POJOs with JAX-RS-compliant annotations which can be found within the package `<app.package.name>.backend.ws`.
- The main extension points are the stateless session beans within the package `<app.package.name>.backend.beans`. Implement the methods `getAllX`, `getFirstX`, `putXes`, `deleteX` (X being the name of the respective entity) to accordingly provide and accept entity objects. The default implementation of these methods read (and write, respectively) all instances into a default database.
- A `persistence.xml` that uses the data source with the JNDI-name `jdbc/__default`.
- A `web.xml` that is configured such that all REST resources are made available using Jersey, Glassfish's default JAX-RS implementation.

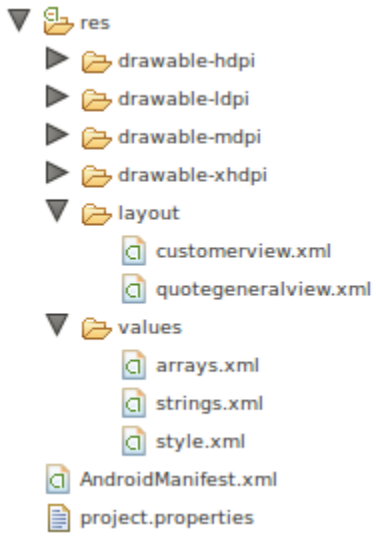
Note that currently remote validations are not implemented and only conceptually included.

When implementing your own RESTful backend and resources, please refer to the Backend connection specification in the Appendix.

App

Android

In this section, we will examine the structure and contents of the generated Android project. It expects the reader to be a Android developer and therefore assumes basic knowledge of the Android SDK.



The generated project contains three major types of files: Generated source files (in `src`), static source files contained in JAR libraries (in `lib`) and resource files (in folder `res`). Its folder of a typical project looks as follows:

Main Concepts

Although they are two conceptually different layers, the components of the app as well as of the runtime library closely

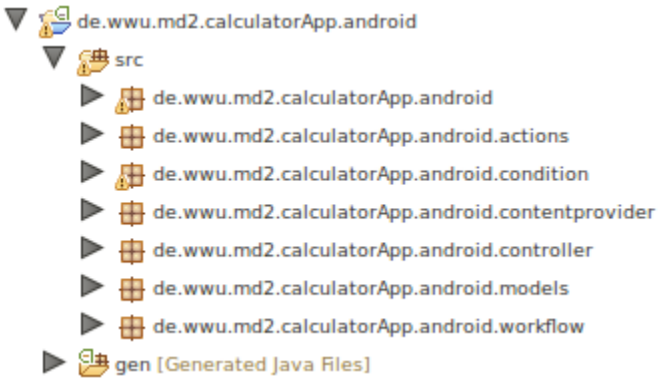
resemble the language architecture. Its concepts are mirrored, as well as applied to Android's SDK library where possible.

Ubiquitous to Android as an operation system is its process model, which imposes certain framing conditions for MD² apps: Applications are a composite of Activities rather than a monolithic process. Activities have a three-stage life cycle that is in full control of the OS. Most importantly, activities can be destroyed indeterminately in situations of hardware resources shortage. Therefore, MD² apps need and do respect Android's lifecycle concept by mapping views *on the root level* to pairs of an Android layout and an activity or fragment. This further fulfills the Android concept of dumb views, which only describe the structure of the user interface, and activities and fragments, which automate them by executing the view logic.

An important factor is how TabbedPanes are realized in MD² Android apps. Since Android 3, the `ActionBar` component plays a central role for the user experience of apps for this operating system. It is a powerful tool for displaying tabbed content, which is why it is used to display tabbed views. Since the `ActionBar` expects its tab to be fragments, The sub-container of a TabbedPane are generated as pairs of layout and fragment rather than layout and activity.

Source code structure

For the sake of keeping the source close to the style of manually written code and not bloating it, MD² uses Android's resource XML files for storing views as layout files, as well as enum values and style information as string resources. The `res` folder furthermore contains images provided by the modeler.



To gain an understanding of the basic architecture, we will discuss several notable classes and packages. All classes that are not specific to the generation context are bundled within the runtime library, which is added to each app as a JAR file.

The base package contains the *project application class*. This class serves as the global context and is available to every Activity and Fragment. It is a composition of application-wide components like the event bus. In it, basic bootstrapping actions are performed, such as registering Actions, Content Providers and Workflows.

The sub-package `action` holds representations of all Custom and Combined Actions in a command pattern style. Custom actions are composed of code fragments. A code fragment is an piece of logic that is executed within the context of an activity. Usually, these logic pieces are the different tasks modeled (for e.g. calling an action, mapping a view element, loading data etc.). When an action is executed, code fragments that cannot be executed in the current activity's context are appended to

a per-activity queue and get executed as soon as the respective activity becomes available. In addition to modeled actions, each project has a AutoGenerationAction that is executed to perform map tasks for view elements created by an AutoGenerator.

The sub-package `contentprovider` contains all content providers in use, if any. The generated classes only hold information specific to the respective content provider declarations, like the entity assignment or filter string. They inherit from concrete implementations for local and remote content providers located within the runtime library, which contain the actual connectivity functionality. Local providers use files on the device storage to persist entities. Remote content providers perform HTTP operations to implement MD²'s transmission protocol for RESTful backends. All providers use JSON as a serialization format, leveraging the external library Jackson for (de-)serialization. Since I/O operations are blocking, they are performed in a background thread using Android's AsyncTask interface.

The sub-package `workflow` contains the application-specific workflow steps, if any. These classes implement the workflow concepts by binding to their assigned view containers. Furthermore, they handle workflow-specific actions by listening to the event bus.

All activities and fragments are placed in the sub-package `controller`. Their purpose is to handle the lifetime of views, trigger actions on user input and cause the view to update.

Finally, the sub-package `model` holds Java representations of entities and enums. Both types are implemented as POJOs: Although the Java language features mighty enum types, MD²'s enums are, at their core, an array of strings of which one is selected. Since we use XML resources to store enum arrays, enum POJOs only need to keep a reference to the array resource and a selection index.

iOS

The general architecture of iOS consists of two types of classes:

- *Static* files
- Dynamically *generated* files

In most cases the generated files access the functionality of the static files.
Hereby, the aim is to outsource as much functionality into those static files as possible, in order to ease the generation process and enhance the flexibility of the framework. As with the general framework architecture, also the iOS application is structured in the core MVC concepts:

1. **Model**
2. **Controller**
3. **View**

The next chapters will step into each of these architecture components and explain their functionality and tasks as well as a distinction between the static and generated parts.

Model

As the model is a quite thin layer in the overall architecture, the only components contained are entities and enumerations. In order to access core data functionality, a data model has to be setup

that defines the database to be accessed later on by the content providers. This is an XML representation of entities, attributes and relationships and a variety of different attributes, like if it is a toMany relationship or the attribute type, can be set.

Entities

Entities will be generated in this data model including the referenced entities as well as normal attributes. For a common object hierarchy with a shared set of attributes, all entities inherit from the static entity "DataTransferObect". Here, the internal ID and the created date is stored for internal database management.

Enumerations

Enumerations on the one side are simply number objects in the data model and, in order to represent the according descriptions, will be generated in the Localizable.strings file and read and set during runtime.

Controller

The major and biggest layer in this architecture is the controller layer and has the most important role in connecting the view with the model and vice versa. Therefore, the following elements are contained:

- *Actions*
- *Content providers*
- *Controllers*
- *Data mappers*
- *Events*
- *Validators*
- *Workflows*

As you can see the basic concepts from the MD² language will be represented here.

Actions

Firstly, the actions encapsulate a certain piece of business logic in a static class, which are either static and possibly accessible by action calls or dynamically generated CustomActions. Hereby, static actions normally implement the static performAction method and can be delivered an event, which contains information to execute the action properly. CustomActions on the other side use the performCustomAction method to implement all custom code fragments defined in the MD² language and each fragment will be generated in one line of code. As the whole application is event/action-driven, all closed piece of functionality is encapsulated in a single action, which is called by the AppDelegate.

The AppDelegate static class serves as the only access to the actions and as a facade from the outside. In addition to the redirection of action calls the whole event-action binding dealing with view events (e.g. the touch on a button) or the check of OnConditionEvents after a view has been changed. Therefore, it provides a protocol, which have to be implemented to redirect event to this class further.

Finally, the central AppData class stores all objects relevant to the actions at runtime, e.g. the controllers or content providers.

Content providers

The second concept supported by the language and implemented in iOS are the content providers. Hereby, each specific content provider has to be configured in some parameters only and the main functionality will be provided by the static ContentProvider class. These parameter are the entity name, if it is local or remote, the connection respectively the url, the allowed operations and the where clause filter.
As already said, the ContentProvider serves as a starting point for each model access and redirects according to the parameters the functionality to static Request classes. These encapsulate a single CRUD operation performed on the model, e.g. read an entity from a remote or create a entity object locally.
In addition, filters will be generated for each where clause defined in a content provider. These dynamically read the input fields addressed in the clause and returns the according filter for the remote or local case.

Controllers

Continuing with the **controllers**, these dynamically generated classes play the central role in connected the model with view elements. Despite their complexity, most functionality will be outsourced to the static Controller class and only the appropriate view, which will also be generated, has to be set. Hence, the controller object manages the access on the view.
Here, the static Controller class provides methods to trigger the view loading, load the data from the model into the view, write changes widget values from the view to the model and check workflow conditions, e.g. perform a check for validity or data.

Data mappers

The next element of the controller layer discussed here is the data mapper. For each controller this component maps the identifier of view elements to the according model parts by accessing the content providers. Therefore, it stands in between the controller objects and the content providers. By the according tasks performed in actions, a mapping can be registered and unregistered at the data mapper. Moreover, it provides auxiliary methods to access and manipulate the content provider and, thus, the model. Another important issue is the translation of enum values into enum strings.

Events

In this section the events and event handling will be presented. Events are generally containers that contain information to execute an action and signal the action handling, which action have to be performed. Most events are static classes and only the OnConditionEvents are dynamically generated out of the model defined by the MD^2 language. These define the condition that have to be checked, whereas the conditions are jointly shared by the workflows, which will be introduced later.

Additionally, the event handling will be processed by the EventHandler class. This is a central hub for any event occuring in the view and delegates them as an intermediary to the correct class, e.g. AppDelegate. Similarly, to the AppDelegate it provides a set of protocol, which view components can implement in order to delegate user actions.

Validators

Another important construct is the validator, which is able to check if the content of a certain widget is valid and provides a rich set of conditions that can be validated. Therefore, all validators are static classes and are not generated for each new validator, but simply instantiated, e.g. with a custom message or min/max range. In detail, the following validator are possible with this framework:

- *AlphabeticValidator* - checks if the data is only alphabetic and not numeric
- *FloatValidator* - checks if the data is a floating point number
- *IntegerValidator* - checks if the data is an integer number
- *NotEmptyValidator* - checks if the data is not empty
- *NumberRangeValidator* - checks if the data as a number is between a certain range
- *RegExValidator* - checks if the data fulfills a certain regular expression
- *StringRangeValidator* - checks if the data has a certain range of characters

In order to provide a single point of access all validators of a certain view element will be encapsulated into the WidgetValidation class. Therefore, validator can added and removed as well as all validator checked at the same time to provide an aggregated validation status with the according error message.

Workflows

Finally, the last element of the controller layer is the workflow and plays a central role in connected different views. For the purpose of managing a set of workflows the workflow management handles different workflows, which can be changed at runtime, and handles the general access on workflows.

One step down workflows itself contain several workflow steps and a name needed to change the current workflow. Similarly, to the WorkflowManagement class it also deals with the change of the workflow step either by the workflow name, the associated controller, the following or succeeding workflow step.

On the lowest level, the workflow step is the most complex construct in this section and contains the name, the associated controller respectively view as well as the forward and backward condition. In terms of methods workflow steps provide methods to check the forward and backward condition.

As already said, in general, conditions can be attached to WorkflowSteps and OnConditionEvents and encapsulate the check of a boolean expression defined in the MD^2 language. Therefore, these expression will be translated in the appropriate method invocations and boolean operators of Objectice-C.

Important to note is the fact that workflow actions attached to a certain workflow step are not handled in the workflow section, but makes use of the event-action binding introduced in the Actions section and will be generated as actions.

View

The view component consists of layouts, widgets and specific views. These are the building blocks to implement a rich and highly functional user interface.

Layouts

Thereby, layouts are static and will only be instantiated by certain views to structure the contained components. In this context, three different layouts will be distinguished:

- *GridLayout* - Arranges components in a fixed number of columns and rows
- *FlowLayout* - Arranges the components in a fixed number of columns or rows horizontally or vertically
- *TableLayout* - Arranges the components in a fixed number of columns and rows with bordered cells

The GridLayout is the most flexible one and enables the modeller to set a fixed amount of rows and columns to put elements in. More restrictive the FlowLayout limits the number of columns or rows to 1 depending on the direction, which can be either horizontal or vertical. Finally, the TableLayout is quite similar to the GridLayout, but draws a line around each cell, in order to make a table-like description of data possible.

Widgets

Secondly, the widgets are also static and instantiated during runtime as components of a specific view. Because widgets are quite specific they do not share much functionality, but only the label and the info button that can be put in front of each of them. Hence, the Widget and InfoWidget class deals with these issues and provides the according functionality to other widgets. Especially the identifiers plays an important role as it uniquely identifies a specific widget, in order to access it later on.
In detail, the following widgets are contained:

- *ButtonWidget* - Displays a button to trigger actions
- *CheckboxWidget* - Displays a checkbox to enable/disable features (boolean values)
- *ComboboxWidget* - Displays a combobox to select out of a list of values (enumerations)
- *EntitySelectorWidget* - Displays an entity selector to select a certain instance from a content provider (entity values)
- *ImageWidget* - Displays an image for style or to trigger actions
- *LabelWidget* - Displays a label for simple data visualization
- *TextfieldWidget* - Displays a text field for arbitrary text input and appropriate validation

Therefore, each view element from the language will be represented by a single static class in iOS. In general, the widgets are only dealing with the graphical representation of itself, mainly by setting the so-called frame (a structure consisting of x position, y position, width and height). Furthermore, all event occuring in widgets are forwarded to the EventHandler in the controller layer. This can for example if the button of a combobox has been clicked and the appropriate popover window should appear.

WidgetFactory
Another construct in the context of widgets is the so-called WidgetFactory. This is a statically accessed class, which is used for the creation and initialization of widgets in general. Here, properties like the info button or the picker characteristics (customer vs. date picker) will be configured.

Views
Finally, the dynamically generated view classes represent the most upper level view objects from the MD² language and, hence, serve as container for all sub layouts and widgets. For this purpose, the init an loadView method has to be implemented. Whereas the first one only sets the identifier for the whole view, in order to catch the appropriate string for the tab bar from the string resources, the loadView method initializes the view and all contained sub layouts and widgets. Therefore, dispatch methods will be used for both the container and content elements, e.g. to distinguish the generation of Buttons and Comboboxes or GridLayouts and FlowLayouts.
As with other language constructs, most functionality will be handled by the static superclass View. This handles methods to create different kinds of widgets, add sub views and layouts as well as widgets, the replacement of specific sub views and retrieve a single component or a list of components, e.g. all ComboboxWidgets or the widget with the identifier "street". Additionally, the basic functions like the insets, which is the same for each view will be initialized here. As you can see, most methods except the last ones are used for the view initialization, whereas the last category is used by the controller layer to access widgets at runtime, e.g. to check the validity of a widget.

Preprocessing

For each target platform an own generator class is written that needs to adhere to the interface `IPlatformGenerator`. Each generator class is registered in `MD2RuntimeModule` and gets injected into the the framework automatically. During modelling the Xtext Builder participant then starts the building process every time a model file is saved and generates the platform files.

Prior to generating the source code in an intermediary step the model gets transformed and then passed to the implementing generator classes. Basically this step can be considered an model-to-model transformation with the aim to simplify the generation process.

Autogenerator
The Autogenerator is a handy feature to easily create view elements from model definitions. During preprocessing the references to ContentProviders are resolved and ContentElements are created based on the model attribute types that the ContentProvider declare . The schema for generation is as follows:

- `IntegerType, FloatType, StringType` become TextInput fields

- `DateType, TimeType, DateTimeType` become TextInput fields with corresponding time type attribution (later depicted as Date/Time pickers).
- `BooleanType` becomes an CheckBox field
- `ReferencedType`
    - `Enum` becomes an OptionInput field
    - `Entites` are processed recursively and all elements are wrapped into a FlowLayoutPane

Remarks
With each ContentElement a new `MappingTask` is created accordingly that maps the ContentElement to the attribute provided by the ContentProvider. For this purpose a CustomAction by the name of autoGenerationAction is created that the platform generators need to parse manually. If the view element is unmapped on startup or an user-specified mapping is found, the auto-generated mapping is removed.

In case the model attribute, from which a ContentElement is created, has the optional parameters `name` or `description` set, these values are converted to label and tooltip representations for TextInputs, OptionInputs and CheckBoxes. If no name is given by the modeller the ID of the ContentElement will be assigned as a label name with each uppercase letter preceded by a whitespace (eg. objectAddress -> Object Address).

Cloning and references

The MD² language allows to define certain view elements once and then lets the modeller reuse these elements multiple times. Internally these elements are copied and references pointing to them are resolved during preprocessing. The same behavior applies to view elements that are referenced in a view container (eg. FlowLayoutPane), but are defined outside of this container. A small example might 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 TabbedPane that references two ContainerElements. 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")
		}
	}
}
```

Resolving references

Because view elements can be reused and referenced across the whole model, MD² needed to provide means 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 that means, only `headerPane` can be referenced, but any `headerPane` in a `customerPane` or `generalPane` not, because simply there is no such element during runtime. To solve this issue MD² introduces 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 become real elements and any reference to this element will be resolved.

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

The modeller also might want to reference an auto-generated ContentElement. This is done by referencing the AutoGenerater followed by the model element in square brackets. In the following scenario the TextInput field that is generated for the Customer's firstname shall be referenced:

```
FlowLayoutPane myPane {
	AutoGenerator customerAutoGenerator {
		contentProvider customerContentProvider
		)
	}
}
```

```
contentProvider Customer customerContentProvider {
      providerType default
}

entity Customer {
      firstName: string
}
```

The reference will be: `myPane.customerAutoGenerator[Customer.firstName]`

Remarks
When a view element, which is referenced or reused at a different place, will be 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.

Validators

To ensure proper data integrity and to validate user input each ContentElement 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 (eg. Integer) and its parameters (eg. optional). 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 on startup.

Type specific

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

Type Parameter specific

- Omitting the `optional` keyword will result in a StandardNotNullValidator beeing 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

Replacements

Enums
The MD² 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"
}
```

Workflows

Nested workflows will be flattened.

Before:

```
workflow wf1 {
      step firstStep:
            view mainView.customerView
      step nestedStep:
            subworkflow wf2
      step thirdStep:
            view mainView.resultView
}
workflow wf2 {
      step secondStep:
            view mainView.InputView
}
```

After

```
workflow wf1 {
      step firstStep:
            view mainView.customerView
      step secondStep:
            view mainView.InputView
      step thirdStep:
            view mainView.resultView
}
```

CombinedAction

The sole use of CombinedActions is to trigger execution of other actions. This behavior can also be achieved by using CallTasks in CustomActions. Thus all CombinedActions will be converted to CustomActions internally.

Miscellaneous

Some minor adjustments are made to the model:

- Check for existence of `flowDirection` parameter for all FlowLayoutPanes.
- Duplicate spacer according to the specified number of spacers
- Replace all named colors by their hex color equivalents
- Replace custom validators with standard validator definitions

TestGenerator
Usually model transformations are transparent to the modeller and even for the developer hard to trace. With the TestGenerator, 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.

Challenges

Some major challenges we had to cope were originated by MD²'s unique approach to cross-platform development. An important factor was the relatively high development efforts per platform. Since we had to implement one generator per target platform, each of which generates a complete application project based only on the meta-model, each generator needed to be able to generate a complete code infrastructure on its own. Attempts in de-duplicating seemingly similar implementation logic are possible and seem promising at first sight. While both applications conceptually share the same MVC-style architecture, the implementations differ vastly because of the different SDKs, application models and programming languages. As a consequence of the resulting generator complexity, it became increasingly challenging to semantically validating the platform generators. An MD² developer not only needs to ensure that the platform implementation of a language component has to correspond the language concept correctly. He or she also has to guarantee that the language concept behaves the same among all platforms. Lastly, a main challenge was to harmonize the different process models of the operating systems, since this is an aspect where they differ substantially. As for all parts that differ substantially, one needs to carefully tradeoff between the benefits of supporting a large amount of platform features vs. finding a universal solution that can be efficiently used by the modeler.

To overcome these issues, we focused on reusing concepts from the MD² language in order to minimize difference between the generated code. We encapsulated as much code fragments in non-generated classes for re-use at runtime. Also, we introduced a preprocessing step. It simplifies the model and thereby encapsulates work that otherwise would have to be done by each generator. In order to tackle the issue of semantically validating the platform apps, one could investigate on the usage of automated acceptance testing tools. Such a tool, if it exists or will be developed, would ideally allow to test a sample MD² app against scenarios of user input and output, which are formulated in a behavior-describing style (cf. Selenium).

One of the goals of MD² was to not just to generate the source code files but to generate projects that could be launched directly. For Android this meant to generate the project, classpath, preferences and project settings files required for importing the project in Eclipse and the manifest file for being able to package the generated project as an Android app. All these files are XML files

consisting of mostly static parts. Therefore this task was not challenging for Android. iOS on the other hand needs an Xcode project file. This required us to completely understand the proprietary structure of such files. As example, all source code and resource files of an Xcode project are stored in one folder. But when the project is opened in Xcode a virtual folder structure is present. This virtual folder structure is defined in the project file. A hash value is assigned to each element defined in the project file. These hash values are needed to reference elements within the project file. These hash values do not have a standardized format but a proprietary one. Therefore we needed to get to know all requirements on those hash values and build our own hash value provider. Besides the already mentioned virtual folder structure, each file contained in the project, each file that shall be compiled and the respective build phase and each element that shall be used as resource have to be listed in different blocks in the project file. To generate the project file, besides building our hash value provider and the actual project file, we had to keep track of all files that will be generated, all static files that will be copied, in which virtual folders those files shall be displayed and which file type it has . Therefore we developed the class FileStructure that provides access to all relevant groups of files, being them virtual folder contents, build or resource files.

Another challenge arose due to the complexity and size of the language. Xtext seems not really suited for large languages. When the lines of code of the generated files reach a certain amount, Xtext is not able to successfully generate these files. The error, that occurred then, was hard to narrow down to the fact, that our language just grow too large. When we finally realized this it was the next problem to get to know, how to solve this problem. In the end we found the possibility to set a property in the generation workflow of Xtext, that limits the lines of code in the generated files, and splits these files if the number will be exceeded. A similar challenge was the support to divide the model among several files. If a model is divided among several files, each file will result in a discrete model. These models are linked to each other if there are references (e.g. a controller element refers to a view element). The default hook of Xtext for the Xtend generators supplies just one model but not the set of all models, which was needed by us. Therefore we first had to determine how to get the set of all models and then how to extend the default hook.

Conclusion

The developed language allows the straightforward modeling of powerful apps. First the definition of the model (in the sense of MVC) is reduced to the necessary elements, specifically entities with typed attributes. But these models can easily enhanced with the definition of simple validators, user friendly names and descriptions. Based on a model a whole view can be build with just one element, namely the AutoGenerator. If the layout, that will be generated, does not fit the requirements one can first determine the attributes for which view elements shall be generated, or define the whole view completely by one self. All common basic elements and simple styles are supported by MD². The same is true for the controller. One can quickly model the controller required for a working app, by defining the Main block, the connection to the backend server, if one is required at all, and a contentprovider. If own elements have been defined these can be connected to the model by using mappings. But, besides such simple controllers, more powerful controllers can be easily defined as

well. The controller elements of MD² offer the possibility to define custom validators, declare own events and allow the definition of workflows. Workflows allow to build apps that navigate the user through several steps of input masks, masks to check the input and finally to show him or her the results. It can be defined when the user is allowed to change to the next step and when not. To sum it up, with MD² one can model first prototypes in five minutes and one can easily adapt changes but is not restricted to simple apps. The generators of MD² work automatically without further notice of the modeler and produce apps that can be directly launched. These apps are native apps that provide the user with native look and feel, have no performance restrictions due to wrapping programs and make use of mobile services as the GPS sensors.

The task of the project seminar was to proof the concept of model driven mobile development. Our result is neither "yes, it works", nor "no, it does not", but "it depends". First, one has to consider the benefit cost ratio. If one has to model and maintain many apps than it seems reasonable to use model driven mobile development. But otherwise the effort used to develop the language and the generators will probably outweigh the benefits. To develop a framework like MD² requires one to construct a language supporting all requirements and still being handy, to develop prototypical apps for each of the target platforms and to transform these prototypes to generators, that conform the target language, respect all possibilities provided by the own language and still be general enough to gain advantages. Additionally for each of the tasks different knowledge is required. One has to know how to develop a language (e.g. Xtext), generators (e.g. Xtend) and apps for each of the target platforms (e.g. iOS and Android). If one would develop apps directly one could work faster and with fewer constraints. This leads to the second drawback of model driven mobile development. If one develops native apps directly, one can benefit from the whole powerfulness of the platform and the development language. If one uses model driven mobile development, one has to stick to a superset of features that are supported by all target platforms and are considered by the own language. In the case of MD², the development of apps, that are used to collect data from and present data to users and transfer the further processing of the data to a backend server, is supported. In summary, if one has to develop and maintain many similar apps, model driven mobile development seems appropriate otherwise one has to weigh the huge effort of the development of the framework against the gained savings due to faster development and easier maintenance of apps.

Besides the general limitations of model driven mobile development there are specific features which, due to resource constraints, were not in the scope of this version:

- **Remote validations**. The ability to transmit entities to the back-end in order to perform complex business logic there. The app receives as a response of which entities or entity attributes are invalid, plus corresponding error messages. It is thereby able to integrate this response with ordinary validations and allow for a coherent user experience.
- **Resilience to network failures**. A big challenge in mobile development, in distinction to classical desktop or server environments, is the unavailability of a reliable and fast network connection. App developers have to expect constant failure, high latencies and low bandwidths. While our apps already provide a solid network connection via remote ContentProviders, they could be further improved by additional means for embracing

network failures, such as an automatic fallback mechanism to local storage or the support for caching for minimizing traffic.

- **Allowing further device functionality**. While we respected the idiosyncrasies of mobile application environments in the design of our platform apps, the currently supported device features (local storage, resolving the current location and handling network connection loss) are rather meant to demonstrate that it is basically possible to use platform features. Based upon that, one could integrate more device functionality, e.g. using a camera for taking photos, by adding upon the existing language components.
- **More styling options**. While using the native look and feel of the respective platforms is a main advantage of our solution, there are use cases where altering the app appearance and user interface is needed. In addition to the basic coloring and sizing possibilities already provided, one could implement further styling options.
- **List views**. For displaying and browsing a collection of entities, MD² currently only provides the EntitySelector component. But a common pattern for mobile apps, especially for small screen devices, is the list view, a container to display a collection of similar information pieces in a scrollable list. Since both the Android as well as the iOS SDK feature these container components, integrating them as a new container element in MD² is a moderately challenging task.

Apart from these tangible features, the issue of extendability & flexibility should be addressed on a more conceptual level. In order to use the MD² approach within a real-world, production-ready environment, means of integrating programmatic business logic into our app models are needed. This could be achieved by adding an expression syntax to the declarative MD² language. This expression syntax could range from an own, slick approach that allows simple arithmetic operations, up to a full-fledged solution of using a suitable existing language like Xbase or even Xtend, which already compiles to Java, and cross-compile it to Objective C. Apart from that, another level of flexibility would be given if a modeller could also integrate custom platform code with generated apps, e.g. to use new features that are not supported by MD² yet. The generation gap pattern is a common means for achieving extendability of generated code[3]. For MD², one could define a stable interface between generated code and user written extensions (in Java or Objective C). This, however, imposes the burdens of manually implementing and synchronizing duplicated logic for all platforms.

Therefore we identified and propose this hybrid solution: For executing custom logic, a ubiquitous scripting environment is used. There are languages for which interpreters for all supported platforms are available, namely JavaScript (Rhino on Android, JavaScriptCore on iOS) and Ruby (Rubuto on Android, MacRuby on iOS). Two scenarios are possible: Either interpreting script code directly (this is similar to frameworks like Appcelerator Titanium[4]), or executing it in an embedded browser (this technique is used by the framework PhoneGap[5]). We think this approach is feasible

[3] cf. http://heikobehrens.net/2009/04/23/generation-gap-pattern/

[4] http://www.appcelerator.com/

[5] http://phonegap.com/

due to some reasons: First, the scripting languages in question provide good support for efficiently implementing tasks like data orchestration, calculation and programming application interaction. Second, existing projects (e.g. PhoneGap) can be leveraged and reused to provide access to system functionality. Furthermore, the amount of platform-specific code is reduced and cross-platform reuse is embraced. Lastly, when using an embedded browser, the modeler could also provide custom application parts as a client-side web application which interacts with the generated MD² application. Thus, we think it would be viable to address this approach as another proof-of-concept project.

Appendix

Demo screenshots

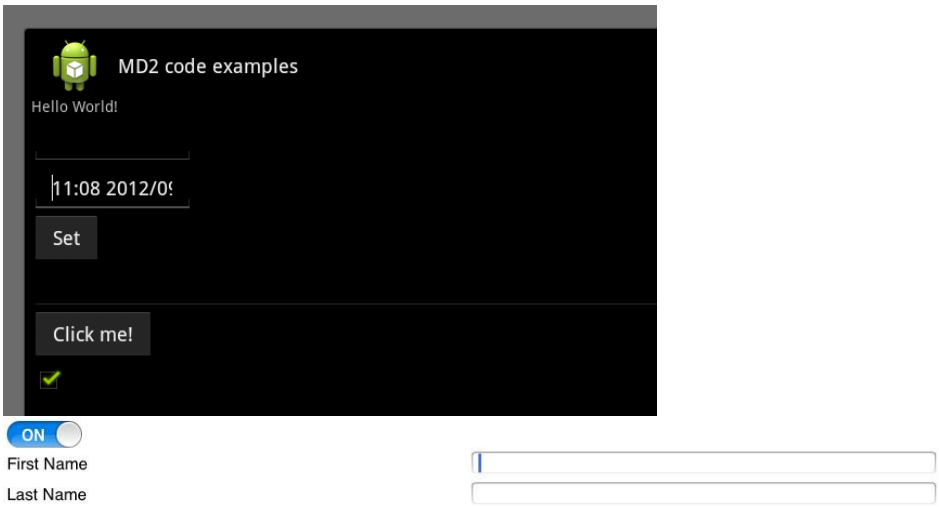


Figure 1: Text input field on Android and iOS

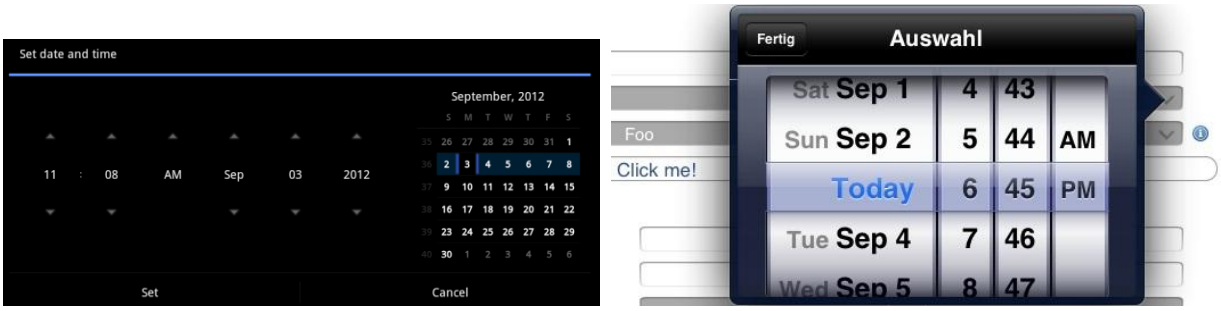


Figure 2: Date and time input field on Android and iOS

Figure 3: Tooltip attached on Widget on Android and iOS

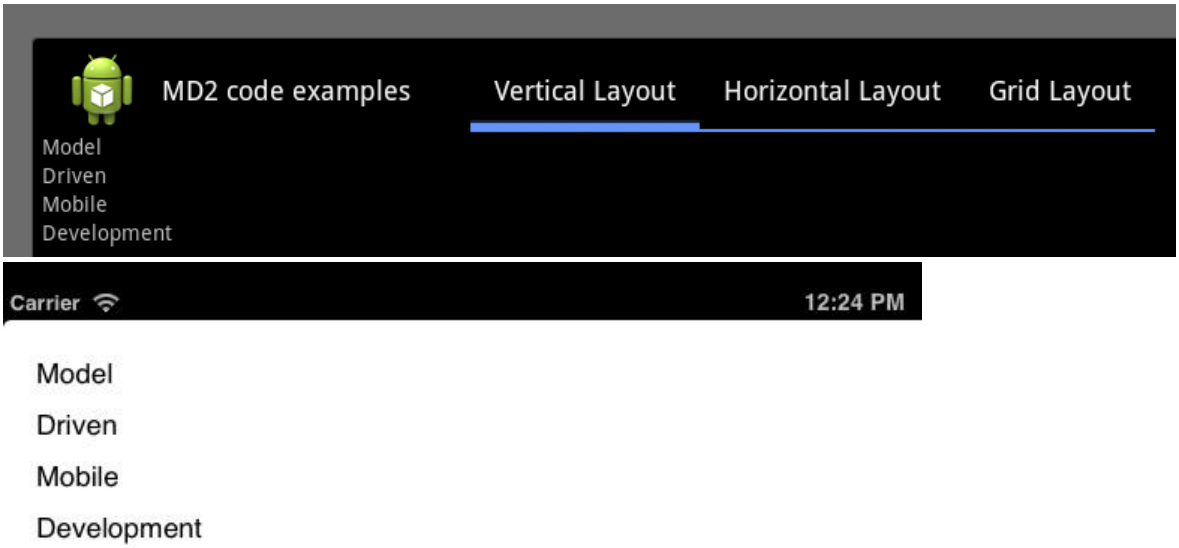


Figure 4: Vertical FlowLayout on Android and iOS

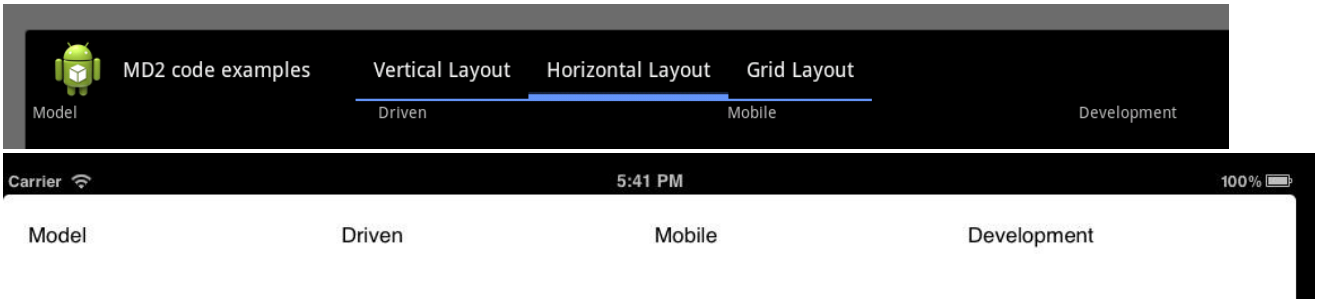


Figure 5: Horizontal FlowLayout on Android and iOS

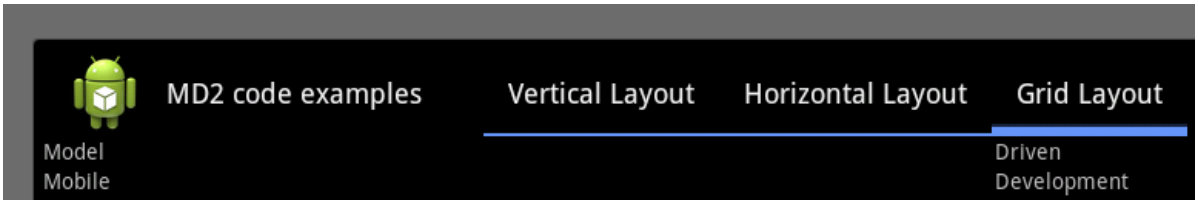


Figure 6: GridLayout on Android and iOS

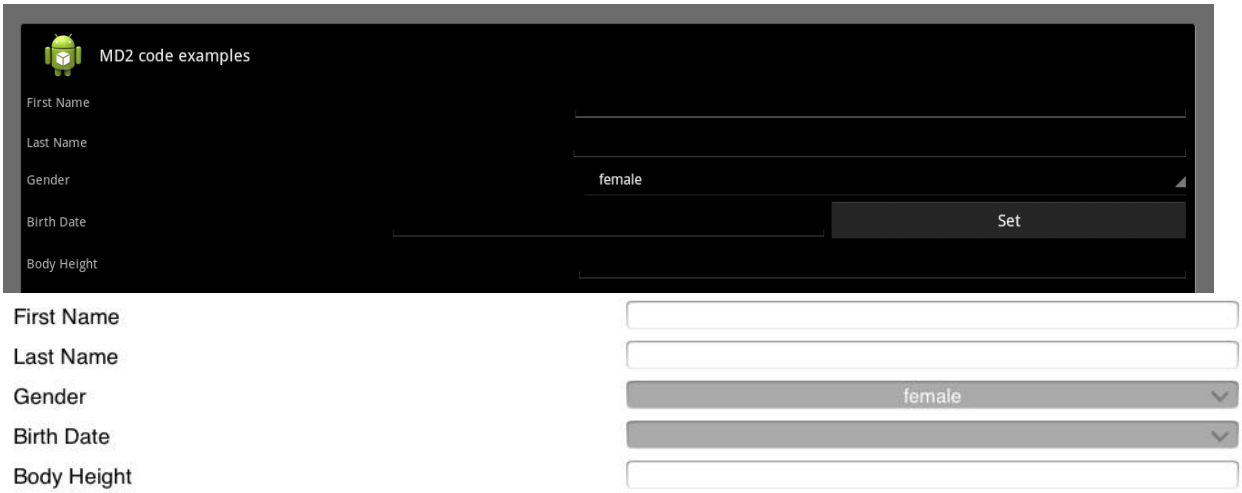


Figure 7: Autogenerated view on Android and iOS

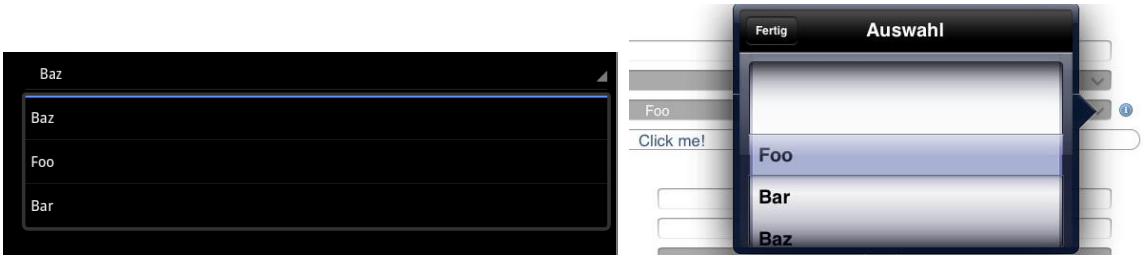


Figure 8: Spinner on Android and Picker on iOS

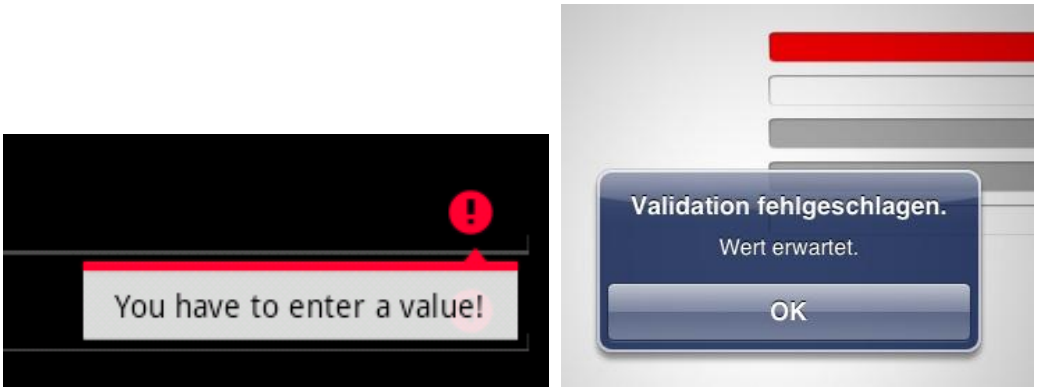


Figure 9: Validated input field on Android and iOS

Language grammar

```
grammar de.wwu.md2.framework.MD2 with org.eclipse.xtext.common.Terminals

generate mD2 "http://www.wwu.de/md2/framework/MD2"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore


////////////////////////////////////
////////////////////////////////////
// Initial
////////////////////////////////////
////////////////////////////////////

/*
 * The MD2Model is the root element of each model.
 * It contains the package definition and model
 * layer in the layer specific type.
 */
MD2Model:
	// Each model layer has to be stored in a
	// corresponding package (see constraints)
	package = PackageDefinition
	modelLayer = MD2ModelLayer?
;

/*
 * The MD2ModelLayer can be either a View,
 * a Controller or a Model
 */
MD2ModelLayer:
	View | Controller | Model
;

/*
 * The PackageDefinition contains the
 * package name as fully qualified name.
 */
PackageDefinition:
	'package' pkgName = QUALIFIED_NAME
;


////////////////////////////////////
////////////////////////////////////
// View layer
////////////////////////////////////
////////////////////////////////////

/*
 * The root View element contains all
 * ViewElements containing to this view model.
 */
View:
```

```
        viewElements += ViewElement+
;

/*
 * A ViewElement can be either
 * a ViewGUIElement or a style.
 */
ViewElement:
       ViewGUIElement | Style
;

/*
 * A ViewGUIElement can be either a
 * ContainerElement or a ContentElement.
 */
ViewGUIElement:
       ContainerElement | ContentElement
;


////////////////////////////////////
// View layer => Content elements
////////////////////////////////////

/*
 * ContentElements are all those elements of a view that don´t contain any
 * nested view elements. They are basically used to present data and
 * information to the user or collect data from the user.
 */
ContentElement:
       TextInput | OptionInput | Label | Image | AutoGeneratedContentElement |
       Spacer | Button | CheckBox | Tooltip | EntitySelector
;

/*
 * A Spacer is used in a GridLayoutPane to mark an empty
 * cell or in a FlowLayoutPane to occupy some space.
 */
Spacer:
       // Hack to force the spacer to have a name attribute in the inferred model
       // the __Dummy keyword will be suppressed in auto completion and via validator
       'Spacer' {Spacer} ('(' 'number'? number = INT? ')')? | '__Dummy' name = ID
;

/*
 * TextInputs are basically used to provide the user the possibility to insert data.
 *
 * Using the 'type' attribute the type of the input field can be be specified
 * currently supported => date, time or time stamp; if no type is set, DEFAULT is
used implicitly
 *
 * TODO add further options to support different input keyboards for e.g. numbers,
emails etc.
 */
TextInput:
```

```
	'TextInput' name = EID ('{' (
		('label' labelText = STRING)? &
		('tooltip' tooltipText = STRING)? &
		('type' type = TextInputType)?
	) '}')?
;

/*
 * This enumeration contains all possible input types of TextInputs.
 */
enum TextInputType:
	DEFAULT = 'default' | DATE = 'date' | TIME = 'time' | DATE_TIME = 'timestamp'
;

/*
 * OptionInputs provide the user the possibility to choose one entry of a list of
strings.
 */
OptionInput:
	'OptionInput' name = EID ('{' (
		('label' labelText = STRING)? &
		('tooltip' tooltipText = STRING)? &
		// Optional: Options may be inferred from the mapped model if its data
type is an enum
		('options' (enumReference = [Enum] | '{' (enumBody = EnumBody)? '}'))?
	) '}')?
;

/*
 * The AutoGeneratedContentElement is bound to a ContentProvider and will
automatically create
 * view elements to display all attributes of the related entity. It is possible to
either
 * exclude attributes specified with exclude or to provide a positive list of
attributes with only.
 * In each case a list of the specified attributes will be stored in
filteredAttributes. Which of
 * the two options has been chosen can be determined via exclude.
 */
AutoGeneratedContentElement:
	'AutoGenerator' name = EID '{' (
		'contentProvider' contentProvider += [ContentProvider | QUALIFIED_NAME]+
			('(' ((exclude ?= 'exclude') | 'only') filteredAttributes +=
EntityPathDefinition+ ')')?
	) '}'
;

/*
 * Buttons provide the user the possibility to call actions,
 * that have been bound on the onTouch event of the Button.
 * This Button specification allows different ways to set the
 * button text (directly or via text attribute).
 */
Button:
	ButtonShorthandDefinition | ButtonExtendedDefinition
```

```
;

/*
 * This is the shorthand definition of a Button, where
 * the text will be set in brackets behind the ID
 */
ButtonShorthandDefinition:
	'Button' name = EID '(' text = STRING ')' ('{'
		('style' style = StyleAssignment)?
	'}')?
;

/*
 * This is the extended definition of a Button, where
 * the text will be set as property
 */
ButtonExtendedDefinition:
	'Button' name = EID '{' (
		'text' text = STRING &
		('style' style = StyleAssignment)?
	) '}'
;

/*
 * CheckBoxes allow the user set boolean values.
 * A predefined status of the CheckBox can be set.
 */
CheckBox:
	'CheckBox' name = EID '{' (
		('label' labelText = STRING)? &
		('tooltip' tooltipText = STRING)? &
		('checked' ((checked ?= 'true') | 'false'))?
	) '}'
;

/*
 * Tooltips allow the modeler to provide the user with additional
 * information. This Tooltip specification allows different ways
 * to set the help text (directly or via text attribute)
 */
Tooltip:
	TooltipShorthandDefinition | TooltipExtendedDefinition
;

/*
 * This is the shorthand definition of a Tooltip, where
 * the image source will be set in brackets behind the ID
 */
TooltipShorthandDefinition:
	'Tooltip' name = EID '(' text = STRING ')'
;

/*
 * This is the extended definition of a Tooltip, where
 * the image source will be set as property
```

```
 */
TooltipExtendedDefinition:
	'Tooltip' name = EID '{'
		'text' text = STRING
	'}'
;

/*
 * Images allow the modeler to specify and display images to the user.
 * This Image specification allows different ways to set the
 * image source (directly or via src attribute)
 */
Image:
	ImageShorthandDefinition | ImageExtendedDefinition
;

/*
 * This is the shorthand definition of a Image, where
 * the image source will be set in brackets behind the ID
 */
ImageShorthandDefinition:
	'Image' name = EID '(' src = STRING ')' ('{' (
		('height' height = INT)? &
		('width' width = INT)?
	) '}')?
;

/*
 * This is the extended definition of a Image, where
 * the image source will be set as property
 */
ImageExtendedDefinition:
	'Image' name = EID '{' (
		'src' src = STRING &
		('height' height = INT)? &
		('width' width = INT)?
	) '}'
;

/*
 * Labels allow the modeler to present text to the user.
 * Normally they are used to denote input elements. This
 * Label specification allows different ways to set the
 * label text (directly or via text attribute)
 */
Label:
	LabelShorthandDefinition | LabelExtendedDefinition
;

/*
 * This is the shorthand definition of a Label, where
 * the text will be set in brackets behind the ID
 */
LabelShorthandDefinition:
	'Label' name = EID '(' text = STRING ')' ('{'
```

```
            ('style' style = StyleAssignment)?
      '}')?
;

/*
 * This is the extended definition of a Label, where
 * the text will be set as property
 */
LabelExtendedDefinition:
      'Label' name = EID '{' (
            'text' text = STRING &
            ('style' style = StyleAssignment)?
      ) '}'
;

/*
 * The EntitySelector allows the user to select an element from a list of
 * elements. The textProposition defines which ContentProvider stores the
 * list and which attribute of the elements shall be displayed to the user
 * to allow him to find the desired element.
 */
EntitySelector:
      'EntitySelector' name = EID '{'
            'textProposition' textProposition = ContentProviderPathDefinition
      '}'
;

////////////////////////////////////
// View layer => Container elements
////////////////////////////////////

/*
 * ContainerElements are used to composite ViewGUIElements.
 */
ContainerElement:
      GridLayoutPane | FlowLayoutPane | AlternativesPane | TabbedAlternativesPane
;

/*
 * A GridLayoutPane allows the arrangement of ViewGUIElements in a grid structure.
The user can specify the
 * number of columns or the number of rows. If one of those is specified the other
one will be calculated by
 * MD2. If both are specified and the resulting number of cells is smaller than the
number of contained
 * elements, the exceeding elements will be ignored.
 */
GridLayoutPane:
      'GridLayoutPane' name = EID '(' params += GridLayoutPaneParam (',' params +=
GridLayoutPaneParam)* ')' '{'
            elements += ViewElementType*
      '}'
;

/*
```

```
 * The GridLayoutPaneParam specifies all possible parameters for a GridLayoutPane.
 * These are besides the number of columns and rows all TabSpecificParams.
 */
GridLayoutPaneParam:
	{GridLayoutPaneColumnsParam} 'columns' value = INT | {GridLayoutPaneRowsParam}
'rows' value = INT | TabSpecificParam
;

/*
 * A FlowLayoutPane allows the arrangement of ViewGUIElements one after another
either horizontally or vertically.
 */
FlowLayoutPane:
	'FlowLayoutPane' name = EID ('(' (params += FlowLayoutPaneParam (',' params +=
FlowLayoutPaneParam)*)? ')')? '{'
		elements += ViewElementType*
	'}'
;

/*
 * The FlowLayoutPaneParam specifies all possible parameters for a FlowLayoutPane.
 * These are besides the flowDirection all TabSpecificParams.
 */
FlowLayoutPaneParam:
	{FlowLayoutPaneFlowDirectionParam} flowDirection = FlowDirection |
TabSpecificParam
;

/*
 * The FlowDirection lists all possible orientations that can be used
 * to define the flow of elements contained in a FlowLayoutPane
 */
enum FlowDirection:
	HORIZONTAL = 'horizontal' | VERTICAL = 'vertical'
;

/*
 * The AlternativesPane allows the definition of several ContainerElements of which
one will be
 * shown to the user. The user will be able to choose which ContainerElement shall be
displayed.
 */
AlternativesPane:
	'AlternativesPane' name = EID  ('(' (params += TabSpecificParam (',' params +=
TabSpecificParam)*)? ')')? '{'
		elements += ContainerElementType*
	'}'
;

/*
 * The TabbedAlternativesPane is a special AlternativesPane that
 * allows the user to switch between tabs by offering him a tab bar.
 */
TabbedAlternativesPane:
	'TabbedPane' name = EID ('(' ')')? '{'
```

```
            elements += ContainerElementType*
      '}'
;

/*
 * TabSpecificParam defines the parameters the user can set to a
 * ContainerElement that is contained in a TabbedAlternativesPane.
 */
TabSpecificParam:
      {TabTitleParam} 'tabTitle' tabTitle = STRING | {TabIconParam} 'tabIcon'
tabIcon = STRING
;

////////////////////////////////////
// View layer => Typedef
////////////////////////////////////

/*
 * The ViewElementType allows to either specify a new ViewGUIElement or to refer to
an existing one
 */
ViewElementType:
      {ViewElementRef} value = [ViewGUIElement | QUALIFIED_NAME] (rename ?= '->'
name = EID)? |
      {ViewElementDef} value = ViewGUIElement
;

/*
 * The ContainerElementType allows to either specify a new ContainerElement or to
refer to an existing one
 */
ContainerElementType:
      {ContainerElementRef} value = [ContainerElement | QUALIFIED_NAME] (rename ?=
'->' name = EID)? ('(' (params += TabSpecificParam (',' params +=
TabSpecificParam)*)? ')')? |
      {ContainerElementDef} value = ContainerElement
;

////////////////////////////////////
// View layer => Style definitions
////////////////////////////////////

/*
 * The StyleAssignment allows the user to
 * either define a new Style or to refer
 * to an existing one.
 */
StyleAssignment:
      {StyleDefinition} definition = StyleBody |
      {StyleReference} reference = [Style]
;

/*
 * A Style allows the user to define a
 * new Style in the StyleBody and allow
```

```
 * the reference to it by defining a name.
 */
Style:
	'style' name = EID
	body = StyleBody
;

/*
 * The StyleBody contains the actual style and can be used to set
 * the appearance of the text of some ViewGUIElements. It is
 * possible to set the fontSize, the color and the textStyle.
 */
StyleBody:
	'{' {StyleBody} (
		('fontSize' fontSize = INT)? &
		('color' color = Color)? &
		('textStyle' ((bold?='bold'? & italic?='italic'?) | 'normal'))?
	) '}'
;

/*
 * A Color can be either specified
 * as HEX_COLOR or NamedColor.
 *
 * Notice:
 * Preprocessing replaces all NamedColors
 * by their hex color equivalents.
 */
Color:
	{HexColorDef} color = HEX_COLOR |
	{NamedColorDef} color = NamedColor
;

/**
 * NamedColor contains the set of the 16 named web colors as specified in HTML 4.01.
 */
enum NamedColor:
	AQUA = 'aqua' | BLACK = 'black' | BLUE = 'blue' | FUCHSIA = 'fuchsia' |
	GRAY = 'gray' | GREEN = 'green' | LIME = 'lime' | MAROON = 'maroon' |
	NAVY = 'navy' | OLIVE = 'olive' | PURPLE ='purple' | RED = 'red' |
	SILVER = 'silver' | TEAL = 'teal' | WHITE = 'white' | YELLOW = 'yellow'
;


////////////////////////////////////
////////////////////////////////////
// Controller layer
////////////////////////////////////
////////////////////////////////////

/*
 * The root Controller element contains all
 * ControllerElements containing to this view Controller
 */
Controller:
```

```
        controllerElements += ControllerElement+
;

/*
 * The ControllerElement is the super type of all possible ControllerElements.
 */
ControllerElement:
      ContentProvider | Validator | Action | Workflow | OnConditionEvent | Main |
RemoteConnection
;

////////////////////////////////////
// Controller layer => Events
////////////////////////////////////

/*
 * An Action provides the user the possibility to
 * declare a set of tasks. An Action can be
 * either a CustomAction or a CombinedAction.
 */
Action:
      'action'
            (CustomAction | CombinedAction)
;

/*
 * A CustomAction contains a list of CustomCodeFragments
 * where each CustomCodeFragment contains one task.
 */
CustomAction:
      'CustomAction' name = EID '{'
            codeFragments += CustomCodeFragment*
      '}'
;

/*
 * CombinedActions allow the composition of Actions.
 */
CombinedAction:
      'CombinedAction' name = EID '{'
            ('actions' actions += [Action | QUALIFIED_NAME]+)?
      '}'
;

/*
 * SimpleActions are simple tasks that can be used by the user to declare operations
that the app shall perform.
 */
SimpleAction:
      {GotoNextWorkflowStepAction} 'NextStepAction' |
      {GotoPreviousWorkflowStepAction} 'PreviousStepAction' |
      {GotoWorkflowStepAction} ('GotoStepAction' '(' wfStep = [WorkflowStep |
QUALIFIED_NAME] (',' silentFails ?= 'silent')? ')') |
      {GotoViewAction} ('GotoViewAction' '(' view = AbstractViewGUIElementRef ')' )
|
```

```
	{DataAction} ('DataAction' '(' operation = AllowedOperation contentProvider =
[ContentProvider | QUALIFIED_NAME] ')') |
	{NewObjectAtContentProviderAction} ('NewObjectAction' '('contentProvider =
[ContentProvider | QUALIFIED_NAME] ')') |
	{AssignObjectAtContentProviderAction} ('AssignObjectAction' '('(bindings +=
NewObjectEntityBinding (',' bindings += NewObjectEntityBinding)*)? ')') |
	{GPSUpdateAction} 'GPSUpdateAction' '(' bindings += GPSActionEntityBinding
(',' bindings += GPSActionEntityBinding)* ')' |
	{SetActiveWorkflowAction} 'SetActiveWorkflowAction' '(' workflow = [Workflow]
')'
;

/*
 * The NewObjectEntityBinding binds a ContentProvider to an object contained in
another ContentProvider.
 *
 * [Move to respective definition] The NewObjectEntityBinding  initializes a new
object for the
 * specified Entity or Attribute in the set ContentProvider.
 */
NewObjectEntityBinding:
	'use' contentProvider = [ContentProvider | QUALIFIED_NAME] 'for' path =
ContentProviderPathDefinition
;

/*
 * The GPSActionEntityBinding defines the combination of GPSFields
 * and STRINGs and to which Attribute of which Entity
 * stored in which ContentProvider it shall be set.
 */
GPSActionEntityBinding:
	entries += GPSActionEntityBindingEntry ('+' entries +=
GPSActionEntityBindingEntry)* 'to' path = ContentProviderPathDefinition
;

/*
 * A GPSActionEntityBindingEntry is
 * either one GPSField or a STRING
 */
GPSActionEntityBindingEntry:
	gpsField = GPSField | string = STRING
;

/*
 * The GPSField lists all possible parts of a GPS position that are supported.
 */
enum GPSField:
	LATITUDE = 'latitude' | LONGITUDE = 'longitude' | ALTITUDE = 'altitude' | CITY
= 'city' | STREET = 'street' |
	NUMBER = 'number' | POSTAL_CODE = 'postalCode' | COUNTRY = 'country' |
PROVINCE = 'province'
;

////////////////////////////////////
// Controller layer => Custom code
```

```
////////////////////////////////////

/*
 * A CustomCodeFragment is a define the possible commands and tasks that can be
declared in a CustomAction.
 *
 * TODO define all keywords that are supported by our language
 * TODO provide possibility to implement arbitrary code (in theory)
 */
CustomCodeFragment:
	{EventBindingTask} ('bind' ('action' | 'actions') actions+=ActionDef+ 'on'
events += EventDef+) |
	{EventUnbindTask} ('unbind' ('action' | 'actions') actions+=ActionDef+ 'from'
events += EventDef+) |
	{ValidatorBindingTask} ('bind' ('validator' | 'validators')
validators+=ValidatorType+ 'on' (referencedFields += AbstractViewGUIElementRef)+) |
	{ValidatorUnbindTask} ('unbind' ('validator' | 'validators')
(validators+=ValidatorType+ | allTypes ?= 'all') 'from' (referencedFields +=
AbstractViewGUIElementRef)+) |
	{CallTask} ('call' action = ActionDef) |
	{MappingTask} ('map' referencedViewField = AbstractViewGUIElementRef 'to'
pathDefinition = ContentProviderPathDefinition) |
	{UnmappingTask} ('unmap' referencedViewField = AbstractViewGUIElementRef
'from' pathDefinition = ContentProviderPathDefinition)
;

/*
 * The AbstractViewGUIElementRef allows to reference any defined ViewGUIElement.
 * First a top level element has to be defined in ref. In tail a recursive navigation
 * to nested elements can be specified. If the last tail points to an
 * AutoGeneratedContentElement path or simpleType can be used to navigate inside the
 * Entity, the ContentProvider is bound on. This will be an reference to the auto
 * generated ViewGUIElement containing the value of the specified Attribute.
 */
AbstractViewGUIElementRef:
	ref = [ecore::EObject | QUALIFIED_NAME] (tail =
NestedAbstractViewGUIElementRef | '[' (path = EntityPathDefinition | simpleType =
SimpleDataTypeWrapper) ']')?
;

/*
 * The NestedAbstractViewGUIElementRef allows to point to an ViewGUIElement
 * nested in another ViewGUIElement. In tail a recursive navigation
 * to nested elements can be specified. If the last tail points to an
 * AutoGeneratedContentElement path or simpleType can be used to navigate inside the
 * Entity, the ContentProvider is bound on. This will be an reference to the auto
 * generated ViewGUIElement containing the value of the specified Attribute.
 *
 * A NestedAbstractViewGUIElementRef will result in an AbstractViewGUIElementRef.
 * The difference and reason, why the NestedAbstractViewGUIElementRef is needed,
 * is that AbstractViewGUIElementRef can just point to top level ViewGUIElements
 * while NestedAbstractViewGUIElementRef can point to nested ViewGUIElements.
 * This is important, if a ViewGUIElement referred in a ContainerElement
 * shall be referenced.
 */
```

```
NestedAbstractViewGUIElementRef returns AbstractViewGUIElementRef:
	'->' ref = [ecore::EObject | QUALIFIED_NAME] (tail =
NestedAbstractViewGUIElementRef | '[' (path = EntityPathDefinition | simpleType =
SimpleDataTypeWrapper) ']')?
;

/*
 * The ActionDef allows the user to
 * either define a new Action or or
 * directly a SimpleAction
 */
ActionDef:
	{ActionReference} actionRef = [Action | QUALIFIED_NAME] |
	{SimpleActionRef} action = SimpleAction
;

/*
 * The EventDef allows the user to refer to one of the three possible events.
 */
EventDef:
	{ViewElementEventRef} (referencedField = AbstractViewGUIElementRef '.' event =
ElementEventType) |
	{GlobalEventRef} event = GlobalEventType |
	{ConditionalEventRef} eventReference = [OnConditionEvent | QUALIFIED_NAME]
;

/*
 * ElementEventType lists all possible
 * events supported by ViewGUIElements
 */
enum ElementEventType:
	ON_TOUCH = "onTouch" |
	ON_LEFT_SWIPE = "onLeftSwipe" |
	ON_RIGHT_SWIPE = "onRightSwipe" |
	ON_WRONG_VALIDATION = "onWrongValidation"
;

/*
 * GlobalEventType lists all possible app wide events.
 */
enum GlobalEventType:
	CONNECTION_LOST = 'onConnectionLost'
;

/*
 * The ValidatorType allows to either specify
 * a new Validator or to use a StandardValidator
 */
ValidatorType:
	{CustomizedValidatorType} validator = [Validator | QUALIFIED_NAME] |
	{StandardValidatorType} validator = StandardValidator
;


////////////////////////////////////
```

```
// Controller layer => Validators
////////////////////////////////////

/*
 * Validator allows to declare one of the supported Validators.
 */
Validator:
	'validator' (
		{RegExValidator} 'RegExValidator' name = EID '{' (params +=
RegExValidatorParam (params += RegExValidatorParam)*) '}' |
		{IsIntValidator} 'IsIntValidator' name = EID '{' (params +=
ValidatorMessageParam)? '}' |
		{IsNumberValidator} 'IsNumberValidator' name = EID '{' (params +=
ValidatorMessageParam)? '}' |
		{IsDateValidator} 'IsDateValidator' name = EID '{' ((params +=
IsDateValidatorParam (params += IsDateValidatorParam)*)?) '}' |
		{NumberRangeValidator} 'NumberRangeValidator' name = EID '{' (params +=
NumberRangeValidatorParam (params += NumberRangeValidatorParam)*) '}' |
		{StringRangeValidator} 'StringRangeValidator' name = EID '{' (params +=
StringRangeValidatorParam (params += StringRangeValidatorParam)*) '}' |
		{NotNullValidator} 'NotNullValidator' name = EID '{' (params +=
ValidatorMessageParam)? '}' |
		RemoteValidator
	)
;

/*
 * 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.
 */
RemoteValidator:
	'RemoteValidator' name = EID '{' (
		(params += ValidatorMessageParam)? &
		('connection' connection = [RemoteConnection])? &
		('model' contentProvider = [ContentProvider] | 'attributes'
(provideAttributes += ContentProviderPathDefinition)+)
	) '}'
;

/*
 * StandardValidator contains all Validators that can be directly
 * assigned to input fields without being defined explicitly before.
 * Each StandardValidator supports a set of parameters to allow
 * to define the behavior of the StandardValidator.
 */
StandardValidator:
	{StandardIsIntValidator} 'IsIntValidator' ('(' (params +=
ValidatorMessageParam)? ')')? |
	{StandardNotNullValidator} 'NotNullValidator' ('(' (params +=
ValidatorMessageParam)? ')')? |
```

```
	{StandardIsNumberValidator} 'IsNumberValidator' ('(' (params +=
ValidatorMessageParam)? ')')? |
	{StandardIsDateValidator} 'IsDateValidator' ('(' (params +=
IsDateValidatorParam (',' params += IsDateValidatorParam)*)? ')')? |
	{StandardRegExValidator} 'RegExValidator' '(' params += RegExValidatorParam
(',' params += RegExValidatorParam)* ')' |
	{StandardNumberRangeValidator} 'NumberRangeValidator' '(' params +=
NumberRangeValidatorParam (',' params += NumberRangeValidatorParam)* ')' |
	{StandardStringRangeValidator} 'StringRangeValidator' '(' params +=
StringRangeValidatorParam (',' params += StringRangeValidatorParam)* ')'
;

/*
 * ValidatorParam is used to define a common super type of all specific
ValidatorParams.
 *
 * The ValidatorParam is not used in the language but will be used by the generators.
Therefore
 * it can be seen as a workaround to manipulate the meta model, that will be built by
Xtext.
 */
ValidatorParam:
	ValidatorMessageParam | IsDateValidatorParam | RegExValidatorParam |
NumberRangeValidatorParam | StringRangeValidatorParam
;

/*
 * ValidatorMessageParam provides the possibility
 * to define a message, that will be shown to the
 * user if the validation fails.
 */
ValidatorMessageParam:
	'message' message = STRING
;

/*
 * The IsDateValidatorParam allows to define a format
 * that the date at hand shall conform to.
 * Additionally it contains the ValidatorMessageParam
 */
IsDateValidatorParam:
	ValidatorMessageParam | {ValidatorFormatParam} 'format' format = STRING
;

/*
 * The RegExValidatorParam allows the definition of a regular
 * expression, that the validator uses to validate the user input.
 * Additionally it contains the ValidatorMessageParam
 */
RegExValidatorParam:
	ValidatorMessageParam | {ValidatorRegExParam} 'regEx' regEx = STRING
;

/*
 * The NumberRangeValidatorParam allows the definition of
```

```
 * a numeric range that shall contain the user input.
 * Additionally it contains the ValidatorMessageParam
 */
NumberRangeValidatorParam:
	ValidatorMessageParam | {ValidatorMaxParam} 'max' max = FLOAT |
{ValidatorMinParam} 'min' min = FLOAT
;

/*
 * The StringRangeValidatorParam allows the definition
 * of a string length range. The length of the STRING
 * input by the user will be checked against this range.
 * Additionally it contains the ValidatorMessageParam
 */
StringRangeValidatorParam:
	ValidatorMessageParam | {ValidatorMaxLengthParam} 'maxLength' maxLength = INT
| {ValidatorMinLengthParam} 'minLength' minLength = INT
;


////////////////////////////////////
// Controller layer => Main
////////////////////////////////////

/*
 * The Main object contains all basic information about an app.
 * Each set of models must contain exactly one Main object.
 */
Main:
	'main' '{'
		(
			('appName' appName = STRING) &
			('appVersion' appVersion = STRING) &
			('defaultConnection' defaultConnection = [RemoteConnection])? &
			('startView' startView = AbstractViewGUIElementRef) &
			('modelVersion' modelVersion = STRING) &
			('onInitialized' onInitializedEvent = [Action]) &
			('defaultWorkflow' defaultWorkflow = [Workflow])?
		)
	'}'
;


////////////////////////////////////
// Controller layer => ContentProvider
////////////////////////////////////

/*
 * The RemoteConnection allows to specify
 * the connection to a backend server.
 */
RemoteConnection:
	'remoteConnection' name = EID '{' (
		'uri' uri = STRING &
		('password' password = STRING)? &
```

```
            ('user' user = STRING)? &
            ('key' key = STRING)?
      ) '}'
;

/*
 * A ContentProvider stores an instance of a ModelElement or a SimpleDataType. It
allows to
 * CREATE_OR_UPDATE (save), READ (load) and DELETE (remove) the stored instance.
Which of
 * those operations is possible is specified in allowedOperations. By default all
operation
 * 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.
 */
ContentProvider:
      'contentProvider' type = DataType name = EID '{' (
            //('cache' ((cache ?= 'true') | 'false'))? &
            'providerType' (default ?= 'default' | local ?= 'local' | connection =
[RemoteConnection]) &
            (filter ?= 'filter' filterType = FilterType ('where' whereClause =
WhereClauseCondition)?)? &
            ('allowedOperations' allowedOperations += AllowedOperation+)? //default
= all CRUD operations
      ) '}'
;

/*
 * The FilterType lists all possible types of filters.
 */
enum FilterType:
      ALL = 'all' | FIRST = 'first'
;

/*
 * The WhereClauseCondition allows to composite criteria that
 * have to be fulfilled by instances to be loaded.
 */
WhereClauseCondition:
      (ops += 'not')? subConditions += WhereClauseConditionalExpression
      (ops += ('and' | 'or') (ops += 'not')? subConditions +=
WhereClauseConditionalExpression)*
;

/*
 * The WhereClauseCondition summarizes all possible criteria that
 * have to be fulfilled by instances to be loaded. It resolves to
 * a WhereClauseCondition and is required to allow recursive definition.
 */
WhereClauseConditionalExpression returns WhereClauseCondition:
      ('(' WhereClauseCondition ')') |
      {BooleanExpression} value = Boolean |
      {AttributeEqualsExpression} eqLeft = EntityPathDefinition op = Operator
eqRight = SimpleExpression
```

```
;

/*
 * The Operator defines all Operators
 * that can be used to compare values.
 */
enum Operator:
      EQUALS = 'equals' | GREATER='greater' | SMALLER = 'smaller' |
      GREATER_OR_EQUAL = '>=' | SMALLER_OR_EQUAL = '<='
      // EQUALS = '==' | GREATER = '>' | SMALLER = '<'
      // By now, do not support these equivalents
      // as they lead to an unsupressable warning
;

/*
 * DataType allows to refer to an already defined
 * ModelElement or to use a SimpleDataType.
 */
DataType:
      (
            {ReferencedModelType} entity = [ModelElement | QUALIFIED_NAME] |
            {SimpleType} type = SimpleDataType
      ) many ?= BRACKETS?
;

/*
 * A SimpleDataTypeWrapper wraps a SimpleDataType.
 * This is used to store the chosen SimpleDataType
 * of the list of possible SimpleDataTypes.
 */
SimpleDataTypeWrapper:
      type = SimpleDataType
;

/*
 * SimpleDataType lists all possible SimpleDataTypes.
 */
enum SimpleDataType:
      INTEGER = 'integer' | FLOAT = 'float' | STRING = 'string' | BOOLEAN =
'boolean' |
      DATE = 'date' | TIME = 'time' | DATE_TIME = 'timestamp'
;

/*
 * AllowedOperation lists all possible AllowedOperations.
 */
enum AllowedOperation:
      CREATE_OR_UPDATE = 'save' | READ = 'load' | DELETE = 'remove'
;


////////////////////////////////////
// Controller layer => Workflows
////////////////////////////////////
```

```
/*
 * A Workflow is used to define several steps in which the
 * application can currently be. It is possible to define
 * several Workflows, Workflows can be nested and there
 * is at most one Workflow active.
 */
Workflow:
	'workflow' name = EID '{'
		workflowSteps += WorkflowStep*
	'}'
;

/*
 * Each WorkflowStep defines one view that is related to it
 * and will be displayed if the WorkflowStep becomes the
 * current WorkflowStep of the active Workflow. Additionally
 * conditions can be defined, that restrict switching to the
 * next or previous WorkflowStep. Also events can be
 * specified that trigger the change to the next or previous
 * WorkflowStep. Instead of the former mentioned settings,
 * a Workflow can be referred to that will become active
 * while this WorkflowStep is the current one.
 */
WorkflowStep:
	'step' name = EID ':' (
		('view' view = AbstractViewGUIElementRef &
		('forwardCondition' '{'
			forwardCondition = Condition
		'}')? &
		('forwardMessage' forwardMessage = STRING)? &
		('backwardCondition' '{'
			backwardCondition = Condition
		'}')? &
		('backwardMessage' backwardMessage = STRING)? &
		('forwardOnEvent' forwardEvents += EventDef+)? &
		('backwardOnEvent' backwardEvents += EventDef+)?) |
		'subworkflow' subworkflow = [Workflow]
	)
;

/*
 * The OnConditionEvent provides the user the possibility
 * to define own events. The OnConditionEvent specifies a
 * Condition. If this Condition will be fulfilled, the
 * OnConditionEvent is fired.
 */
OnConditionEvent:
	'event' 'OnConditionEvent' name = EID '{'
		condition = Condition
	'}'
;

/*
 * A Condition allows to compose ConditionalExpressions with the operators 'not',
'and' and 'or'.
```

```
 */
Condition:
	(ops+='not')? subConditions += ConditionalExpression (ops += ('and' | 'or')
(ops+='not')? subConditions += ConditionalExpression)*
;

/*
 * A ConditionalExpression defines an expression that evaluates to either true or
false.
 * It is possible to set a BooleanExpression, an EqualsExpression, that compares to
values,
 * or a GuiElementStateExpression that proofs the state of a ViewGUIElement.
 */
ConditionalExpression:
	('(' Condition ')') |
	{BooleanExpression} value = Boolean |
	{EqualsExpression} eqLeft = SimpleExpression (op = 'equals' (not ?= 'not')?
eqRight = SimpleExpression) |
	{GuiElementStateExpression} op = 'is' (not ?= 'not')? isState =
ViewElementState reference = AbstractViewGUIElementRef
;

/*
 * Boolean lists all
 * possible Boolean values.
 */
enum Boolean:
	TRUE='true' | FALSE='false'
;

/*
 * ViewElementState lists all
 * possible ViewElementStates.
 */
enum ViewElementState:
	VALID='valid' | EMPTY='empty' | CHECKED='checked' | FILLED='filled'
;

/*
 * A SimpleExpression contains either a
 * simple data type value of STRING, INT
 * or FLOAT or a Reference to the value
 * of a ViewGUIElement.
 */
SimpleExpression:
	{StringVal} value = STRING |
	{IntVal} value = INT |
	{FloatVal} value = FLOAT |
	{Reference} reference = AbstractViewGUIElementRef
;


////////////////////////////////////
////////////////////////////////////
// Model layer
```

```
////////////////////////////////////
////////////////////////////////////

/*
 * The root Model element contains all
 * ModelElements containing to this model model
 */
Model:
        modelElements += ModelElement+
;

/*
 * The ModelElement is
 * the super type of all
 * possible ModelElements.
 */
ModelElement:
        Entity | Enum
;

/*
 * An Enum allows the user to define
 * a list of STRING values, stored
 * in the enumBody.
 */
Enum:
       'enum' name = EID '{'
              (enumBody = EnumBody)?
       '}'
;

/*
 * An EnumBody allows the user to define
 * a list of STRING values.
 */
EnumBody:
       elements += STRING (',' elements += STRING)*
;

/*
 * An Entity allows the user to define a type,
 * that will be used as data transfer object.
 * The user can provide a list of Attributes.
 */
Entity:
       'entity' name = EID '{'
              attributes += Attribute*
       '}'
;

/*
 * Each Attribute has a name and an AttributeType.
 * Additionally the user can provide further
 * information to an Attribute, namely an
 * extendedName and a description. These further
```

```
 * information will be used by the
 * AutoGeneratedContentElement to generate a
 * label and a tooltip.
 */
Attribute:
	name = EID ':' type = AttributeType ('{'
		('name' extendedName = STRING)?
		('description' description = STRING)?
	'}')?
;

/*
 * AttributeType is the super type of all possible AttributeTypes.
 * These can be besides SimpleDataTypes a references to an already
 * defined ModelElement or an implicit Enum declaration.
 */
AttributeType:
	{ReferencedType} entity = [ModelElement | QUALIFIED_NAME]  many ?= BRACKETS?
('(' (params += ReferencedTypeParam (',' params += ReferencedTypeParam)*)? ')')? |
	{IntegerType} 'integer'
	many ?= BRACKETS? ('(' (params += IntegerTypeParam (',' params +=
IntegerTypeParam)*)? ')')? |
	{FloatType} 'float'
	many ?= BRACKETS? ('(' (params += FloatTypeParam (',' params +=
FloatTypeParam)*)? ')')? |
	{StringType} 'string'
	many ?= BRACKETS? ('(' (params += StringTypeParam (',' params +=
StringTypeParam)*)? ')')? |
	{BooleanType} 'boolean'
	many ?= BRACKETS? ('(' (params += BooleanTypeParam (',' params +=
BooleanTypeParam)*)? ')')? |
	{DateType} 'date'
	many ?= BRACKETS? ('(' (params += DateTypeParam (',' params +=
DateTypeParam)*)? ')')? |
	{TimeType} 'time'
	many ?= BRACKETS? ('(' (params += TimeTypeParam (',' params +=
TimeTypeParam)*)? ')')? |
	{DateTimeType} 'timestamp'
	many ?= BRACKETS? ('(' (params += DateTimeTypeParam (',' params +=
DateTimeTypeParam)*)? ')')? |

	// EnumType => Transformed to explicit Enum (=> ReferencedType) after
preprocessing
	{EnumType} '{' (enumBody = EnumBody)? '}'                                  many
?= BRACKETS? ('(' (params += EnumTypeParam (',' params += EnumTypeParam)*)? ')')?
;

/*
 * The parser rule BRACKETS defines
 * how square brackets look like.
 * BRACKETS can be annotated to an
 * AttributeType to declare the
 * Attribute to have a one to many
 * relation.
 */
```

```
BRACKETS:
	'[' ']'
;

/*
 * PathDefinition is used to define a common super type of all specific
PathDefinitions.
 *
 * The PathDefinition is not used in the language but will be used by the generators.
Therefore
 * it can be seen as a workaround to manipulate the meta model, that will be built by
Xtext.
 */
PathDefinition:
	ContentProviderPathDefinition | EntityPathDefinition
;

/*
 * The EntityPathDefinition defines
 * a path to an attribute of an entity.
 */
EntityPathDefinition:
	entityRef = [Entity] tail = PathTail
;

/*
 * The ContentProviderPathDefinition
 * defines a path to an attribute of
 * an entity stored in a ContentProvider.
 */
ContentProviderPathDefinition:
	contentProviderRef = [ContentProvider] (tail = PathTail)?
;

/*
 * The PathTail defines a recursive
 * list of nested Attribute references.
 */
PathTail:
	'.' attributeRef = [Attribute] (tail = PathTail)?
;

////////////////////////////////////
// Attribute parameter definitions
////////////////////////////////////

// TODO Support default values?

/*
 * AttributeTypeParam is used to define a common super type of all specific
AttributeTypeParams.
 *
 * The AttributeTypeParam is not used in the language but will be used by the
generators. Therefore
```

```
 * it can be seen as a workaround to manipulate the meta model, that will be built by
Xtext.
 */
AttributeTypeParam:
	ReferencedTypeParam | IntegerTypeParam | FloatTypeParam | StringTypeParam |
BooleanTypeParam | DateTypeParam | TimeTypeParam | DateTimeTypeParam | EnumTypeParam
;

/*
 * ReferencedTypeParam is a super type of all
 * possible parameters of referenced Attributes.
 * The value being optional is the only parameter.
 * This parameter will be used to generate a validator.
 */
ReferencedTypeParam:
	{AttrIsOptional} optional ?= 'optional'
;

/*
 * IntegerTypeParam is a super type of all
 * possible parameters of integer Attributes.
 * Besides the value being optional, it is
 * possible to mark it as the identifier of
 * the entity and to provide a minimum and
 * a maximum value. These parameters will
 * be used to generate validators.
 */
IntegerTypeParam:
	{AttrIsOptional} optional ?= 'optional' |
	{AttrIdentifier} identifier ?= 'identifier' |
	{AttrIntMax} 'max' max = INT |
	{AttrIntMin} 'min' min = INT
;

/*
 * FloatTypeParam is a super type of all possible
 * parameters of float Attributes. Besides
 * the value being optional, it is possible
 * to provide a minimum and a maximum value.
 * These parameters will be used to generate validators.
 */
FloatTypeParam:
	{AttrIsOptional} optional ?= 'optional' |
	{AttrFloatMax} 'max' max = FLOAT |
	{AttrFloatMin} 'min' min = FLOAT
;

/*
 * StringTypeParam is a super type of all possible
 * parameters of string Attributes. Besides the
 * value being optional, it is possible to mark
 * it as the identifier of the entity and to
 * provide a minimum and a maximum string length.
 * These parameters will be used to generate validators.
 */
```

```
StringTypeParam:
      {AttrIsOptional} optional ?= 'optional' |
      {AttrIdentifier} identifier ?= 'identifier' |
      {AttrStringMax} 'maxLength' max = INT |
      {AttrStringMin} 'minLength' min = INT
;

/*
 * BooleanTypeParam is a super type of all
 * possible parameters of boolean Attributes.
 * The value being optional is the only
 * parameter. This parameter will be used
 * to generate a validator.
 */
BooleanTypeParam:
      {AttrIsOptional} optional ?= 'optional'
;

/*
 * DateTypeParam is a super type of all
 * possible parameters of date Attributes.
 * Besides the value being optional, it
 * is possible to provide a minimum and a
 * maximum value. These parameters will
 * be used to generate validators.
 */
DateTypeParam:
      {AttrIsOptional} optional ?= 'optional' |
      {AttrDateMax} 'max' max = DATE |
      {AttrDateMin} 'min' min = DATE
;

/*
 * TimeTypeParam is a super type of all
 * possible parameters of time Attributes.
 * Besides the value being optional, it
 * is possible to provide a minimum and a
 * maximum value. These parameters will
 * be used to generate validators.
 */
TimeTypeParam:
      {AttrIsOptional} optional ?= 'optional' |
      {AttrTimeMax} 'max' max = TIME |
      {AttrTimeMin} 'min' min = TIME
;

/*
 * DateTimeTypeParam is a super type of
 * all possible parameters of datetime
 * Attributes. Besides the value being
 * optional, it is possible to provide
 * a minimum and a maximum value. These
 * parameters will be used to generate
 * validators.
 */
```

```
DateTimeTypeParam:
	{AttrIsOptional} optional ?= 'optional' |
	{AttrDateTimeMax} 'max' max = DATE_TIME |
	{AttrDateTimeMin} 'min' min = DATE_TIME
;

/*
 * ReferencedTypeParam is a super type of
 * all possible parameters of referenced
 * Attributes. The value being optional
 * is the only parameter. This parameter
 * will be used to generate a validator.
 */
EnumTypeParam:
	{AttrIsOptional} optional ?= 'optional'
;


////////////////////////////////////
////////////////////////////////////
// Terminal and data type rules
////////////////////////////////////
////////////////////////////////////

/**
 * Definition of a date. A string that conforms of the following format is expected:
 * YYYY-MM-DD
 */
DATE returns ecore::EDate:
	STRING
;

/**
 * Definition of the time. A string that conforms of the following format is
expected:
 * hh:mm:ss[.nnnnnnn][(+|-)hh[:mm]] or hh:mm:ss[.nnnnnnn][Z]
 */
TIME returns ecore::EDate:
	STRING
;

/**
 * Definition of date and time. A string that conforms ISO 8601 is expected.
 * YYYY-MM-DDThh:mm:ss[.nnnnnnn][(+|-)hh[:mm]] or
 * YYYY-MM-DDThh:mm:ss[.nnnnnnn][Z]
 */
DATE_TIME returns ecore::EDate:
	STRING
;

/**
 * Float #.#
 */
FLOAT returns ecore::EDouble:
	INT '.' INT
```

```
;

/*
 * Extended, keyword insensitive ID. Using the EID allows to use a known keyword as
an ID.
 */
EID:
	'type' | 'default' | 'label' | 'tooltip' | 'options' | 'contentProvider' |
'exclude' |
	'tabTitle' | 'tabIcon' | 'textProposition' |
	'displayAll' | 'fontSize' | 'color' | 'textStyle' | 'bold' | 'italic' |
'normal' |
	'aqua' | 'black' | 'blue' | 'gray' | 'green' | 'lime' | 'maroon' | 'navy' |
'olive' | 'purple' |
	'red' | 'silver' | 'white' | 'yellow' | 'action' | 'actions' |
	'silent' | 'event' | 'valid' | 'empty' | 'filled' | 'enum' | 'entity' | 'name'
|
	'description' | 'optional' | 'latitude' | 'longitude' | 'altitude' | 'city' |
'street' | 'number' |
	'postalCode' | 'country' | 'province' |ID

	//	'only' | 'style' | 'text' | 'checked' | 'height' | 'width' |
	//	'src' | 'columns' | 'rows' | 'horizontal' | 'vertical' |
	//	'onRightSwipe' | 'onWrongValidation' | 'onConnectionLost' | 'validator'
|
	//	'connection' | 'model' | 'attributes' | 'message' | 'format' | 'regEx' |
'max' | 'min' |
	//	'maxLength' | 'minLength' | 'main' | 'appName' | 'appVersion' |
'defaultConnection' | 'startView' | 'modelVersion' |
	//	'uri' | 'password' | 'user' | 'key' |
	//	'cache' | 'providerType' | 'local' | 'filter' | 'where' |
'allowedOperations' | 'first' |
	//	'save' | 'load' | 'remove' | 'workflow' | 'step' |
	//	'forwardMessage' | 'backwardMessage' | 'validator' | 'validators' |
	//	'unbind' | 'map' | 'unmap' | 'all' | 'call' | 'onTouch' | 'onLeftSwipe'
| 'use' | 'for' | 'to' | 'bind' | 'on' | 'from'
	//	'forwardCondition' | 'forwardOnEvent' | 'backwardCondition' |
'backwardOnEvent' |'view' | 'onInitialized' | 'defaultWorkflow' |
	//	'integer' | 'float' | 'string' | 'boolean' | 'date' | 'time' |
'timestamp' | 'remoteConnection' |  'contentProvider' |
;

/**
 * Qualified names of the form
 * <tt>ID (.ID)*</tt>
 */
QUALIFIED_NAME:
	EID ('.' EID)*
;

/**
 * Hexadecimal color definitions of the form #ffffff or #ffffffff.
 * In the 8-digit definition the leading two digits specify the (optional) alpha
channel.
 */
```

```
terminal HEX_COLOR:
       '#'
       (('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f'))?
       ('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f')
       ('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f')
       ('0'..'9'|'A'..'F'|'a'..'f') ('0'..'9'|'A'..'F'|'a'..'f')
;
```

Backend connection specification

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 oder 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 names, 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) }

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|fale),
  "error": [
     {
      "message": "Allgemeine Fehlermeldung",
      "attributes": ["attribut1", "attribut2"]
     },
     {
      "message": "can’t 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`

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"
      }
      ]
}
```