

WESTFÄLISCHE  
WILHELMS-UNIVERSITÄT  
MÜNSTER



Project Seminar  
Model-driven Mobile Development

---

# MD<sup>2</sup> – Handbook

---

*Group Members:*

Jan Dageförde

Julia Dittmer

Andreas Fuchs

Carolin Gülpen

Holger Koelmann

Malte Möser

Tobias Reischmann

*Supervisors:*

Prof. Dr. Herbert Kuchen

Jan Ernsting

Group for Practical Computer Science

University of Münster

19th March 2015



# Contents

<b>Introduction</b>	<b>1</b>
<b>Modeler's Handbook</b>	<b>3</b>
1 Installation . . . . .	3
1.1 Setting up your MD <sup>2</sup> Model Development Environment . . . . .	3
1.2 Setting up your map.apps Development Environment in NetBeans . . . . .	3
1.3 Setting up GlassFish to Run Generated Backends . . . . .	4
2 Getting Started . . . . .	5
2.1 Creating a Project . . . . .	5
2.2 Developing a Single App . . . . .	5
2.3 Deploying a Single App . . . . .	21
3 Development and Deployment of Multiple Apps . . . . .	22
4 Additional Features . . . . .	23
4.1 Calling RESTful Web Services from an App . . . . .	23
4.2 Control of Workflow by calling a RESTful Web Service . . . . .	24
<b>Developer's Handbook</b>	<b>27</b>
1 Installation . . . . .	27
2 DSL Semantics . . . . .	28
3 map.apps Implementation . . . . .	29
3.1 Static map.apps Implementation . . . . .	29
3.2 Generated map.apps implementation . . . . .	37
4 Backend . . . . .	37
4.1 Beans . . . . .	37
4.2 Datatypes . . . . .	37
4.3 Entities . . . . .	38
4.4 File Download Servlet . . . . .	38
4.5 Web Services . . . . .	38
5 map.apps Generator . . . . .	40

<b>A</b>	<b>Sample Workflow</b>	<b>I</b>
<b>B</b>	<b>Known Issues and Suggestions for Future Development</b>	<b>III</b>
<b>C</b>	<b>Further Improvements to your Development Environment</b>	<b>V</b>
1	Reference a Generated App in the Development Project . . . . .	V
2	Jetty: Allow Serving of Symbolically Linked Files . . . . .	V
<b>D</b>	<b>Backend Connection Specification</b>	<b>VII</b>
1	Resource paths . . . . .	VII
2	JSON format conventions . . . . .	VIII
3	Examples . . . . .	IX

# Introduction

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

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

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

Searchtags  
for TO-  
DOs:  
@TOBI,  
@JAN,  
@CARO,  
@MALTE,  
@JULE,  
@ANDI,  
@HOL-  
GER;  
  
FOR  
EVERY-  
ONE:  
@ALL



# Modeler's Handbook

@ALL :  
review pls

This chapter represents the handbook for people who want to use the MD<sup>2</sup> framework for the development of applications. It consists of

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

In general, the MD<sup>2</sup> framework provides the modeller with an understanding of how to generate a multitude of Apps and a corresponding Backend out of a single model source. Using eclipse as an IDE, convenient features, like the auto formatting of the model code by pressing STRG+SHIFT+F, can be used.

## 1 Installation

### 1.1 Setting up your MD<sup>2</sup> Model Development Environment

The following software is required to enable modelling of MD<sup>2</sup> models:

- A current Eclipse IDE with support for Java EE development (e.g. Luna)
- Using the Xtext Update Site, install a current version of the Xtext redistributable
- From the archive that you obtained, install the MD<sup>2</sup> features

You are ready to go.

### 1.2 Setting up your map.apps Development Environment in NetBeans

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

- map.apps 3.1.0

- NetBeans EE (e.g. Version 8.0)
- Apache Tomcat 7.0 with running map.apps runtime<sup>1</sup>

### 1. Set up your map.apps development environment in NetBeans

- Extract the `sampleProjRemote` project from the map.apps distribution and load it in NetBeans.
- In its `pom.xml`, set the `mapapps.remote.base` directive to the URL where the map.apps runtime is installed.
- Right-click the project, select “Custom” and select “Goals...” to create a new Maven goal.
- Open the `nbactions.xml`.

Describe configuration

Describe set property for Jetty port 9090

### 2. Deploy the generic MD<sup>2</sup> runtime bundles by copying the MD<sup>2</sup> runtime bundles into the directory `src/main/js/bundles/` in the project:

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

## 1.3 Setting up GlassFish to Run Generated Backends

To enable deployment of generated backends, follow the subsequent steps:

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

<sup>1</sup>For details on their installation, please refer to the map.apps documentation.



---

“Found GlassFish Server version 4.0.0”.

Otherwise, follow the assistant’s hints.

6. Click “Finish”.

## 2 Getting Started

### 2.1 Creating a Project

1. Initialize a new project by navigating to `File > New > Project...`. There, choose `Other > MD2 Project` and click `Next`.
2. Choose a project name (note that it may not contain whitespace or other non-alphanumeric characters). If necessary, choose a location different from the proposed default.
3. After clicking on `Finish`, a default project structure is generated which you can extend as you need.

The created project contains the following folders:

- `src` contains the MD<sup>2</sup> models of your application and is initialized with a simple default. model.
- `resources` contains resources that the generator will copy into the generated applications.
- `src-gen` contains all artifacts that the generator derives from your model. This folder is organized by platform. For example, there is a folder for the backend and for map.apps.

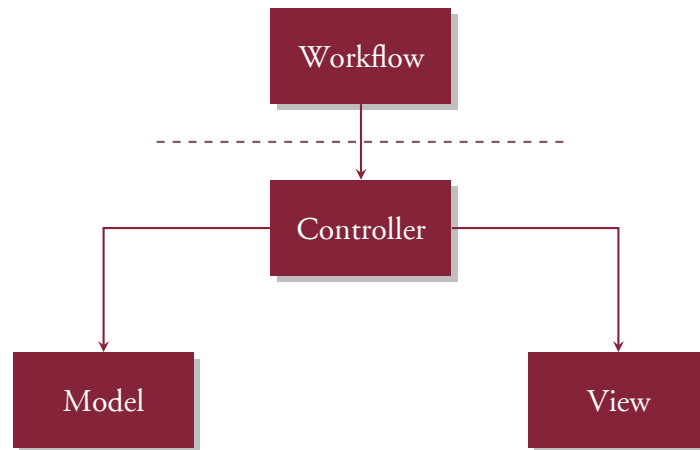
### 2.2 Developing a Single App

This section describes how an application can be developed based on the current (March 2015) state of the MD<sup>2</sup> DSL. To this end, it is structured according to the different layers that constitute a MD<sup>2</sup> model, which are shown in Figure 1. Aside from the conventional MVC layers – model, view and controller – the architecture includes an additional layer which enables the modelling as well as the generation of workflows within and across apps.

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

```
package PACKAGE_NAME
```

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

Figure 1: Architecture of MD<sup>2</sup> Models

### 2.2.1 Workflow

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

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

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

In addition to the events that can be fired, the workflow element also specifies which workflow element is to be started in response to a fired event using the keyword `start`. Moreover, when an event is fired, not only workflow elements can be started but the workflow can be terminated by using the `end workflow` keyword. A workflow element in the workflow layer typically looks as shown in Listing 1.

Listing 1: Workflow Elements in the Workflow Layer

```

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

```

---

```
        end workflow
    }
```

After defining the sequence of workflow elements, the workflow also requires the specification of an app that executes the workflow elements. As shown in Listing 2, an app consists of its ID, a list of workflow elements that are used in the app and a name that is to be used as the app title. In the scenario where only a single app is modeled, all workflow elements can be listed in the app. However, it is also possible to have unused workflow elements.

Listing 2: App Definition in MD<sup>2</sup>

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

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

Finally, the complete workflow specification for one app will be structured as shown in Listing 3. Note that MD<sup>2</sup> does not differentiate between different workflows. However, it is possible to implicitly create multiple workflows by using two or more startable workflow elements that start independent, disjunct sequences of workflow elements.

Listing 3: Workflow Definition in MD<sup>2</sup>

```
package <ProjectName>.workflows

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

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

### 2.2.2 Model

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

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

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

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

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

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

- Entity
- Enum

Supported simple data types are:

- `integer` – integer
- `float` – float of the form `##`
- `boolean` – boolean
- `string` – a string that is embraced by single quotes (`'`) or double quotes (`"`)
- `date` – a date is a string that conforms the following format: `YYYY-MM-DD`
- `time` – a time is a string that conforms the following format: `hh:mm:ss[(+|-)hh[:mm]]`
- `datetime` – a date time is a string that conforms the following format: `YYYY-MM-DDThh:mm:ss[(+|-)hh[:mm]]`
- `file` – a file to be uploaded and stored in an entity field

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

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

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

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

Exemplary entity that represents a person:

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

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

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

## 2.2.3 View

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

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

```
GridLayoutPane NAME (<parameters>) {
```

```

    <Container | Content | [Container] | [Content]>
    <Container | Content | [Container] | [Content]><...>
}

```

For each grid layout at least the number of rows or the number of columns has to be specified. If only one of these parameters is given, the other is calculated by MD<sup>2</sup> 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 `TabbedPanels` 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 `TabbedPanels` for more details.

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

```

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

```

The following comma-separated parameters are supported:

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

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

```

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

```

---

**2.2.3.2 Content Elements** *Input elements* can be used to manipulate model data via mappings (see Section 2.2.4). At the moment BooleanInputs, TextInputs, IntegerInputs, NumberInputs, TimeInputs, DateInputs, DateTimeInputs and **OptionInputs are supported**. All input elements support the optional attributes **label**, **tooltip**, **type**, **disabled**, **default** and **width**. **type** allows to specify the type of the field. **disabled** was intended to provide a boolean for whether an input field is enabled or disabled, but is not regarded by the generator in the current implementation. Default values can be set using **default** and the width using **width**.

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

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

*OptionInputs* are used to represent enumeration fields in the model. In addition to the aforementioned attributes, OptionInputs support the optional **options** attribute. This can be used to populate the input with the string values of the specified **Enum**. If options is not given, the displayed options depend on the Enum type of the attribute that has been mapped on the input field (see Section 2.2.4).

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

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

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

as well as this shorthand definition

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

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

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

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

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

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

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

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

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

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

For the *image* exists the following default definition

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

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

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



---

}.

Images support the following attributes:

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

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

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

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

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

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

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

**Spacer** (INT)

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

```
AutoGenerator NAME {
```

```

    contentProvider [ContentProvider] (exclude|only [Attribute])
}

```

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

```

EntitySelector NAME {
    textProposition [ContentProvider.Attribute]
}

```

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

```

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

```

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

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

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

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

Element -> NAME

---

## 2.2.4 Controller

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

```
main {  
    appVersion STRING  
    modelVersion STRING  
    workflowManager [RemoteConnection]  
    defaultConnection [RemoteConnection]  
    fileUploadConnection [RemoteConnection]  
}
```

The attributes are explained as follows:

- `appVersion` – a string representation of the current app version, e.g. “RC1”
- `modelVersion` – a string representation of the current model version that has to be in accordance with the model version of the backend
- `defaultConnection` (optional) – a default remote connection can be specified here, so that it is not necessary to specify the same connection in each content provider
- `workflowManager` – the remote connection to the workflow manager has to be specified here
- `fileUploadConnection` (optional) – a remote connection to be specified if a file is expected to be uploaded or an uploaded file to be displayed. It is obligatory if at least one `UploadedImageOutput` or `FileUpload` is specified in the view.

review  
points for  
correctness  
@MALTE

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

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

Workflow elements work with `ProcessChains` that define a sequence of possible steps inside the workflow element and are further described in [Section 2.2.4.3](#). Since several process chains can be defined in a single workflow element, the `defaultProcessChain` keyword is used to set a default process chain. This process chain will then be used as starting point for the workflow element and the first view to be shown will also be derived from the default process chain.

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

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

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

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

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

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

- `bind` <Action1> ... <ActionN> `on` <Event1> ... <EventN>
- `bind` <Validator1> ... <ValidatorN> `on` <ViewElement1> ... <ViewelementN>
- `unbind` <Action1> ... <ActionN> `from` <Event1> ... <EventN>
- `unbind` <Validator1> ... <ValidatorN> `from` <ViewElement1> ... <ViewelementN>
- `call` <Action>
- `map` <ViewElement> `to` <ContentProviderField>
- `unmap` <ViewElement> `from` <ContentProviderField>
- `set` <ContentProvider> = <Expression>
- `set` <ViewElement> = <Expression>
- `if` (<Condition>){ <CustomCodeFragment> }
- `elseif` (<Condition>){ <CustomCodeFragment> }
- `else` {<CustomCodeFragment>}

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

Actions are bound to events. There are several types of actions and events available. `CustomActions` and `CombinedActions` are referenced externally whereas `SimpleActions` are declared directly. For events, there

---

are local event types that listen to the state of a certain view element as well as global event types. The most powerful event type is the `OnConditionEvent`.

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

- `ProcessChainProceedAction`: `ProcessChainProceed`  
proceed to the next `ProcessChain` step
- `ProcessChainReverseAction`: `ProcessChainReverse`  
go back to the last `ProcessChain` step
- `ProcessChainGotoAction`: `ProcessChainGoto` `<processChainStep>`  
change to the given `ProcessChain` step
- `SetProcessChainAction`: `SetProcessChain` [`ProcessChain`]  
changes the current `ProcessChain`
- `GotoViewAction`: `GotoView` (`<ViewElement>`)  
change to the given view element
- `DisableAction`: `Disable` (`<ViewElement>`)  
disables a view element
- `EnableAction`: `Enable` (`<ViewElement>`)  
enables a view element
- `DisplayMessageAction`: `DisplayMessage` (`<SimpleExpression>`)  
displays a message
- `ContentProviderOperationAction`: `ContentProviderOperation` (`<AllowedOperation>` `<ContentProvider>`)  
perform a CRUD action (save, load, remove) on the given `ContentProvider`
- `ContentProviderResetAction`: `ContentProviderReset` (`<ContentProvider>`)  
resets the given `ContentProvider`
- `FireEventAction`: `FireEvent` (`<WorkflowEvent>`)  
fires a workflow event to the backend. In response a new workflow element will be started or the workflow terminated.
- `WebServiceCallAction`: `WebServiceCall` (`<WebServiceCall>`)  
sends a request to call an external web service to the backend (for details cf. section 4.1).
- In addition, the `LocationAction` allows to extract an address from content provider fields and to generate a punctual location (i.e. longitude and latitude) for this address. For this purpose, all input and output fields have to be defined. The calculation result for longitude and latitude will be written in their respective output fields. The `LocationAction` is structured as follows:

```
Location (  
    inputs (  
        cityInput <ContentProviderPath>
```

```

streetInput <ContentProviderPath>
streetNumberInput <ContentProviderPath>
postalInput <ContentProviderPath>
countryInput <ContentProviderPath> )
outputs (
latitudeOutput <ContentProviderPath>
longitudeOutput <ContentProviderPath>).

```

There are different event types available:

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

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

```

event NAME {
    <Condition>
}

```

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

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

```

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

```

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

```

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

```

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

---

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

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

*CallTasks* call a different **Action**.

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

*CombinedActions* allow the composition of **Actions**.

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

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

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

```
validator RegexValidator NAME (message STRING regex STRING)
```

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

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

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

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

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

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

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

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

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

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

The *NotNullValidator* makes the input field required.

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

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

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



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

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

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

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

```
step NAME:
    view <[ContainerElement] | [ContentElement]>
    proceed {
        on <Event>
        given <Condition>
        do <Action>
    }
    reverse on <Event>
    goto <ProcessChainStep> (returnTo <ProcessChainStep>) on <Event>
    return on <Event>
    return and proceed on <Event>
    message STRING
```

ProcessChains can be refined using SubProcessChains.

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



---

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

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

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

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

The *remote connection* allows to specify a URI for the backend communication as well as a path to specify the storage location of files to be uploaded. The backend must comply with the MD<sup>2</sup> web service interface as specified in Appendix D.

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

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

## 2.3 Deploying a Single App

### 2.3.1 Backend

The following steps will start the GlassFish server that is used for deploying the backend. Therefore, please grant all privileges that your firewall asks for, if any, so that the server can be accessed.

1. Within the GlassFish installation directory, navigate to `glassfish/bin/`.
2. Run the `asadmin` utility (Windows: Double-click on `asadmin.bat`, Linux/OS X: Open a terminal in that directory and run `./asadmin`).
3. In the GlassFish administration utility, type `start-database` to start the Derby database for the backend.

4. Start Eclipse and import the generated project `<PROJECT_NAME>.backend` by choosing “General” → “Existing Projects into Workspace” in the import wizard.
5. In the Project Explorer tab, right-click the project in Eclipse, choose “Properties”, and navigate to “Targeted Runtime”.
6. Deselect all runtimes and click “Apply”.
7. Select the item “GlassFish 4.0” and click “Apply”.
8. Correct JRE-related build path problems, if any, by resorting to the default JRE.
9. Confirm by clicking “OK”.
10. In the Servers tab, right-click the “GlassFish 4.0” entry, and choose “Add/Remove”.
11. Add the backend project to the server.
12. Start the server.

### 2.3.2 map.apps

You have two alternatives for deploying a map.apps app into your NetBeans development environment:

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

An additional benefit of the last alternative is that newly generated code, that results from changes to your model, will automatically be pushed to the map.apps development environment. If the server is running there, the code will also be deployed to the server automatically.

## 3 Development and Deployment of Multiple Apps

The MD<sup>2</sup> framework also allows to model and generate workflows that involve multiple apps. For this purpose, several apps rather than just one can be specified in the workflow layer. These apps can share the same workflow elements or use different ones as shown in Listing 4. Apart from that, the workflow will look as usual, with the sequence of workflow elements being determined via events. After generation, each app is provided with a list of open issues in the start screen. In this list, all events are presented that were fired from another app and are supposed to start a workflow element which belongs to the current app in action. A user can simply click a listed event to continue the appropriate workflow element.

Listing 4: Workflow Definition for Multiple Apps

```
package <ProjectName>.workflows
```

---

```

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

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

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

```

The deployment of multiple apps is similar to that of a single app. In this case, however, not just one but all created apps have to be deployed, e.g. by setting the corresponding symbolic links as described in Section 2.3 for each app.

## 4 Additional Features

### 4.1 Calling RESTful Web Services from an App

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

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

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

Listing 5: Calling a Web Service From Within a Workflow

```
// Specification of the web service call
```

```

externalWebService <externalWebServiceCallOne> {
    url URL
    method (GET | POST | PUT | DELETE)
    queryparams(
        STRING : (INT | STRING | FLOAT | BOOLEAN | <ContentProviderPath>)
        STRING : (INT | STRING | FLOAT | BOOLEAN | <ContentProviderPath>)
        <...>
    )
    bodyparams (
        STRING : (INT | STRING | FLOAT | BOOLEAN | <ContentProviderPath>)
        <...>
    )
}

// Specify action to call the web service
action CustomAction <CustomActionOne> {
    call WebServiceCall <externalWebServiceCallOne>
}

```

## 4.2 Control of Workflow by calling a RESTful Web Service

The MD<sup>2</sup> language offers a possibility to define a RESTful web service according to listing 6, which will start a certain workflow element. For this purpose for each workflow element marked as *invokable* a web service is created with one endpoint for each invoke definition. When a invoke definition is created within a workflow element of the controller model, the respective workflow element in the workflow model has to be marked as invokable as well. Here a event description can be added, which will be shown in the list of open issues as last event fired.

Listing 6: Offer a Web Service to Start a Workflow

```

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

```

The minimum structure of a invoke definition is simply the keyword *invokable*. The standard path where an endpoint is injected is “/”. If another path should be used this can be defined after the *at*. This is mandatory if multiple endpoint should be created. The standard http method used for the RESTful web service endpoint is *POST*. However, after *using PUT* can be defined instead. In the body of a invoke definition it can be defined if entities and their attributes should be set somehow during the web service

---

call. For this purpose three different possibilities exist. In the following they are described in the order of their appearance within listing 6.

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

For each entity somehow referenced within the definition an instance of this entity will be created and persisted. The type of persistence depends on whether the remote connection of the content provider equals the one of the `workFlowManager`. If they are equal the web service call and the persistence is done on the same server, thus the internal enterprise java beans can be used. Otherwise the other external backend server needs to be called, which is not yet implemented in the current version of the MD<sup>2</sup>. It is not enough, that the urls of the remote connections are equal. The objects need to be identical. If the body is missing no entities or attributes are set.



# Developer's Handbook

The goal of this chapter is to provide the reader with the detailed information necessary for the further development of the MD<sup>2</sup> framework. It includes the required information about the

- installation procedure,
- DSL Semantics,
- structure of the map.apps and backend implementation,
- and gives insights about the structure and interaction of the generated and static code.

In addition, it is also noteworthy that while further developing the framework, the development process should consider that changes within the DSL might also have a direct effect on validators and the formatter. This should be kept in mind during the planning and implementation of future changes. The whole development project together with additional tool and example Apps is available on the projects GitHub page in different repositories.

## 1 Installation

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

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

@ALL  
what  
about the  
prepro-  
cessor [it  
is after all  
the basis  
for all the  
generat-  
ors?]

@ALL  
review pls

## 2 DSL Semantics

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

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

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

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

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

However, if a workflow element fires an event which starts a new workflow element within the same app, this should be handled by the app-specific EventHandler, i.e. no backend communication is required. This is important if future developments are supposed to allow for temporary off-line usage of apps. Thus, the backend handles the start of new workflows *across* apps (currently implemented as EventHandlerWS) and the app-specific event handler handles the start of new workflow elements *within* apps.

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

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



---

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

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

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

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

## 3 map.apps Implementation

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

The generated code for map.apps can be subdivided into three parts: static map.apps code, dynamically generated map.apps code and a backend. The static map.apps code contains the part of the code which does not depend on the models created in the MD<sup>2</sup> DSL. Since it is static, it does not need to be generated, but is required for the overall functionality of the generated apps. The dynamically generated part is completely dependent on the model. The backend is implemented in Java and contains static as well as dynamic code. However, it is completely generated. The backend provides a server which offers functionality such as data storage and communication across apps.

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

### 3.1 Static map.apps Implementation

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

### 3.1.1 Form controls

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

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

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

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

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

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

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

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

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

Special dataform elements enable the use of uploaded files. The `UploadedImageOutput` displays image which have been uploaded by an app's user using a `FileUpload` input element. Given that both elements are mapped to an entity's attribute of type `file`, these elements retrieve an image from, or store an image

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

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

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

### 3.1.2 List of open issues

The `md2_list_of_open_issues` comprises all code necessary for displaying the list of open issues within the app. This list shows all workflow instances, whose state is at a workflow element, which belongs to the

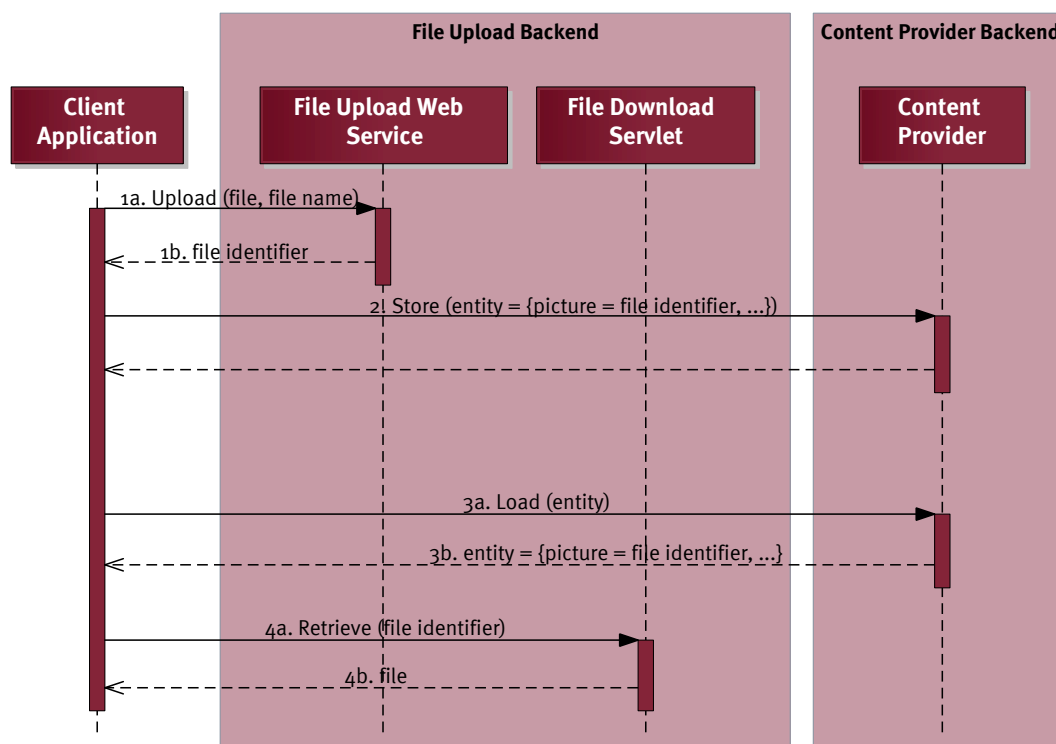



Figure 1: Procedure of uploading and retrieving user-uploaded images

current app. Currently supported data listed in this widget are the guid of the workflow instance, the workflow element name and the last fired event. The list is included as `dijit_widget` and is listed as a `Tool` within the `app.json` under the bundle specifications of `toolset`. In the `ListOfOpenIssuesController` a `DataView` is created, which uses the workflow store as a `DataViewModel`. The workflow store is described in section 3.1.7. In addition to just displaying the workflow instances it is possible to start the workflow element through clicking on the respective entry. Therefore the `ListOfOpenIssuesController` handles the event `onClicked` and calls the function `startWorkflow` of the respective `MD2MainWidget`. Therefore the workflow instance ID is retrieved in combination with the content provider IDs of its current state. With these the content provider referenced in the `workflowStateHandler` are reset to their respective values.

### 3.1.3 Local store

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

### 3.1.4 Location service

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

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

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

### 3.1.5 Runtime

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

**3.1.5.1 MD2MainWidget** The `MD2MainWidget` is responsible for the opening, closing, etc. of views. Each workflow element (see 3) has its own instance of a MD<sup>2</sup> main widget. This is specified in the re-

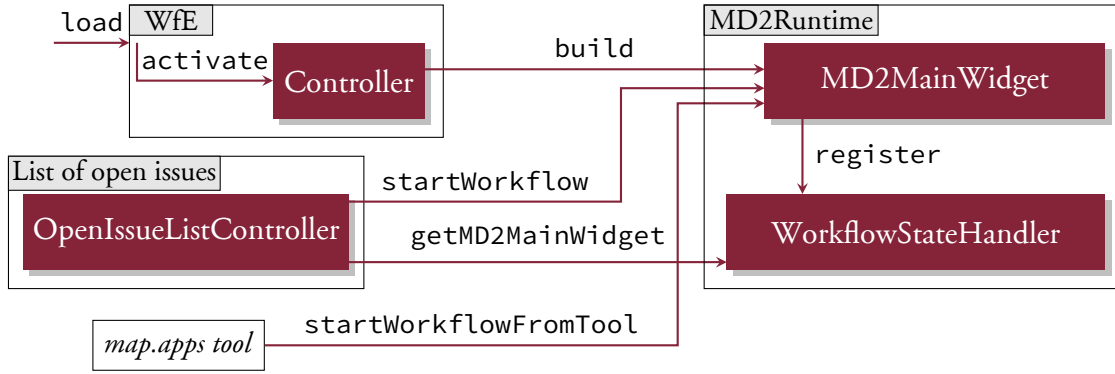


Figure 2: Initialization of the MD<sup>2</sup> main widget in order to start a workflow

spective controller of the workflow element bundle inside the app. That is, the `manifest.json` of the workflow element bundle references an `_md2AppWidget` for its controller. Once the controller is activated (i.e. the `activate` function is called), the respective MD<sup>2</sup> main widget instance is built. This MD<sup>2</sup> main widget is implemented in the file `MD2MainWidget`, which serves as the basic starting point to start a workflow. Thus, it provides methods to start a workflow element. There are different ways a workflow can be started:

- One way is to start it directly from the `map.apps` tool bar. Each workflow element marked as `startable` in the model for this app, gets its own tool. Through clicking on it the method `startWorkflowFromTool` is started, which first resets all current workflow elements and the workflow handlers and then calls the `startWorkflow` function.
- Another way is to start a workflow from the list of open issues (see section 3.1.2). Therefore the `startWorkflow` is directly started from the `OpenIssuesListController`.

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

**3.1.5.2 Actions** Each action that is defined in the MD<sup>2</sup> DSL must also exist in the MD<sup>2</sup> runtime bundle in `map.apps` to be used. Individual actions are stored in the subfolder `simpleactions`. Moreover, an `ActionFactory` must provide a simple method that returns an instance of the respective action (e.g., `getLocationAction` for the `LocationAction`). All actions provide a method `execute` that implements the action. An individual constructor allows to initialize the action, e.g., setting the city's name for a

LocationAction. The ActionFactory is instantiated in the method build of the MD2MainWidget (see figure 2). Thus, every workflow element can access and use this factory.

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

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

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

**3.1.5.6 Entities** This folder contains all internal entities known to the MD<sup>2</sup> map.apps framework. This is the Location entity and the abstract class \_Entity, which is inherited by all generated entities.

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

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

---

**3.1.5.9 Resources** This folder contains images and style files.

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

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

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

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

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

### 3.1.6 Store

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

### 3.1.7 Workflow Store

The workflow store within `md2_workflow_store` differs from the other stores provided by the static bundles. While the local (section 3.1.3) the remote store (section 3.1.6) are used within a content provider the workflow store has its own purposes. This store is used to save and query the status of all workflow instances. On the one side it is injected in the `WorkflowStateTransaction` of the `md2_runtime` bundle (see ?? 3.1.5.13). There it is responsible of informing the backend of an change within the workflow instance. In the model this action can be represented by the `FireEventAction`. Beside the information which event has been fired and which was the current workflow element and instance id, the backend get a list of all ids registered in the content providers. This is necessary to be able to restore the state of the content providers in another app.

On the other hand the workflow store is referenced in the list of open issues. The workflow store implements, same as the other, the functions of the `dojo/Store`. Through this it is possible to hand the workflow store over to a `DataView`, which then displays the information retrieved from the store. While the query function enables to query the whole list of the list of open issues

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

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



## 3.2 Generated map.apps implementation

### 3.2.1 Content Providers

### 3.2.2 Models

### 3.2.3 Workflow Elements

### 3.2.4 Workflow

### 3.2.5 App.json

## 4 Backend

The MD<sup>2</sup> backend is implemented in Java. It is generated by the MD<sup>2</sup> framework from the MD<sup>2</sup> model. Some code in the backend is static, though.

### 4.1 Beans

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

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

### 4.2 Datatypes

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

Listing 7: An integer wrapper for the internal identification number

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

```

@XmlElement
protected int __internalId;

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

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

```

### 4.3 Entities

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

### 4.4 File Download Servlet

### 4.5 Web Services

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

Additionally, some static web service are generated that are used for specific features. Those web services are explained in the following.

#### 4.5.1 Calls to External Webseices

As a simple way to interact with external services, a web service `CallExternalWebServiceWS` in the backend allows to call another web service, that might be on a different system or server. The web service

Please give  
me a de-  
scription...  
@JAN

---

in the backend provides a method that takes a JSON-encoded object as an input. This object must contain the URL, the REST method type and the set of parameters of that method. For example, the following listing depicts how such a JSON object is constructed in map.apps.

Listing 8: JSON-encoded object containing information to call an external web service

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

### 4.5.2 Event Handler

For the communication across apps, the backend offers an event handler web service. This web service handles all workflow events that are fired in one app and start a workflow element in another app, i.e. it is necessary for all switches between apps. Required parameters for this web service are

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

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

### 4.5.3 Workflow State

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

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

#### 4.5.4 File Upload

@JAN  
Please give  
me a de-  
scription...

#### 4.5.5 Version Negotiation

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

## 5 **map.apps** Generator

empty  
@ALL  
[why  
is this  
chapter  
after the  
map.apps  
Imple-  
menta-  
tion?]

# A Sample Workflow

empty?  
@JAN,  
@TOBI,  
@JULE,  
@MALTE



## B Known Issues and Suggestions for Future Development

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

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

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

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

Other possible features are

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

convert a click on a map into location data,

convert coordinates into an address,

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



# C Further Improvements to your Development Environment

## 1 Reference a Generated App in the Development Project

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

1. Open a terminal and navigate to the map.apps NetBeans project directory (e.g. the extracted sample ProjRemote from step 1a in Section 1.2).
2. Navigate to the apps/ directory using `cd src/main/js/apps`.
3. Create a symbolic link using an appropriate command (where `<PROJECT_NAME>` is the name of your MD<sup>2</sup> Project in Eclipse, `<ECLIPSE_PROJECT_LOCATION>` is its location, and `<APP_NAME>` is the name of the generated app(s)):
  - a) Windows:

```
mklink /j <APP_NAME> <ECLIPSE_PROJECT_LOCATION>\<PROJECT_NAME>\src-gen\  
<PROJECT_NAME>.mapapps\<APP_NAME>
```
  - b) Linux / OS X:

```
ln -s <ECLIPSE_PROJECT_LOCATION>/<PROJECT_NAME>/src-gen/<PROJECT_NAME>.  
mapapps/<APP_NAME> <APP_NAME>
```
4. Repeat step 3 for every generated app that you would like to have refreshed automatically.

## 2 Jetty: Allow Serving of Symbolically Linked Files

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



@ALL  
review pls

<Status> - <Response body>

## VII

## Filter parameter

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

## Resource for model version checks

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

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

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

## 2 JSON format conventions

List<Entity>:

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

Having <Entity> = Entity without root node

Entity:

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

---

Validation Result:

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

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

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

Int -> Number

Float -> Number

<Everything else> -> String

Date -> String im Format yyyy-mm-ddThh:mm:ss+hh:mm

### 3 Examples

GET /customer/first returning one customer

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

```
}
```

GET /customer returning multiple customers

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