Hier fehlt
ein Logo
Hier fehlt
ein Logo

moba*DSL* Documentation

version 0.8.0

2015-11-25

Contents

1 Purpose

TBD (ekke)

Zweck und Nutzen der DSL

2 Overview

TBD (ekke)

Architektur des Gesamtsystems, Zusammenspiel DSL und Generatoren ...

3 Installation

TBD (ekke)

Eclipse, IntelliJ, Xcode?

4 mobaDSL Grammar

Applications are modeled in application model files. These files take the extension `.moba`. In a `.moba` file, one can use the mobaDSL to describe application models.

The main semantic elements of the mobaDSL are the following:

- "Application" – the root element of an application model that contains all the other elements. An application may consist of several application models.
- "Template" declarations – used to import other application models into an application model file. Thus, elements (e.g. datatype declarations) may be shared between models and can be extended in application models they are imported into.
- "Generator" declarations – used to specify which generator should be used to generate the application code from the model file.
- "Datatype" declarations – a way to define datatypes that can be used in the application model (or imported into other application models using template declarations).
- "Constant" declarations – used to define or overwrite/redefine constants.
- "Enum" declarations – a way to define enums with given values (or to extend resp. customize "inherited" enums).
- "Entity" declarations – the model of an object that can be persisted on the device. It can contain further elements such as properties and references.
- "DTO" declarations – the model of a Data Transfer Object (DTO) that can be used as the payload that is sent to or received from RESTful services. It can contain further elements such as properties and references.
- "Queue" specifications – used to define a FIFO queue that can contain Entities, DTOs or other queues.
- "REST service" definitions – used to describe REST interfaces (and, depending on the generator, to generate the appropriate services).
- "Settings" definitions – a mechanism for defining application properties etc. (e.g. user name and password, preferred connection types etc.).
- "Attribute" – a reference to an enum, a constant, a class or a "datatype" (as defined in the datatype declaration); used within Entity, DTO and settings definitions. Offers multiplicity.
- "Reference" – used within Entity or DTO definitions to reference another Entity resp. DTO (or, within a queue definition, to reference Entities, DTOs or queues). Offers multiplicity.
- "Properties" – Properties in the form of `key=value` pairs can be attached to datatypes and attributes.
- "Comments" can be added to all elements.

5 Application model files

Application models are described in `.moba` files. These files describe application model and are the basis for the code generation. Application models may be split over several `.moba` files.

`application` An application declaration is the root element of the moba*DSL* grammar. Everything is contained in an application: Template imports, generator selections, datatypes constants, enums, Entities, DTOs, queues, settings etc. have to be defined inside the application definition.

In short, "everything is an application".

Applications must be given a version number (in the form of *`major.minor.revision`* or *`major.minor.revision-SNAPSHOT`*. Furthermore, a set of properties in the form of `key=value` pairs may be defined for an application in brackets.

Syntax:

```
application name version version [ properties ] {
    template imports
    generator selections
    datatype definitions
    Entities
    DTOs
    queues
    settings
}
```

`use template` Model elements may be shared among several application model files. To do this, the "receiving" application model file must import the "originating" model as a template. The `use template` keyword is used to address the application model where elements should be inherited from.

The application that is used as a template can be specified in the form of *`applicationName`*`:`*`version`*. Content assist for applications defined in the same workspace is available. Alternatively, templates can be referenced by a URL in the form of `index://`*`URL`*`:`*`applicationName`*`:`*`version`*. The download can then be triggered via the "Quickfix" tooltip.

URL?

Elements inherited from another application model may be extended or customized.

Syntax:

```
use template applicationName:version
use template index://URL:applicationName:version
```

generator This keyword allows the selection of a generator that converts the application model to executable code. The generator can be specified using its generator ID and must be given a name.

The generator ID must match an Equinox extension in the extension point `org.mobadsl.grammar.generatorDelegate` and can be given as a fully qualified name after the `extensionPoint` keyword within braces after the generator name. The version of the generator to use is defined after a colon. See chapter 13 on how to use custom generators for translating models to executable code.

Flo zu Hülf!

Within the braces, the `mixin` keyword allows the specification of a reference to another defined generator that is then attached to and called together with the generator (sequentially as defined).

If several generators are defined, the one generator that is actually called can be marked with the `active` keyword. The other generators are used only if they are tied into it using the `mixin` keyword.

Syntax:

```
active generator name {
    extensionPoint generatorId : generatorVersion
    mixin otherGeneratorName
}
```

6 Simple Application Features

6.1 Authorization

The mobaDSL makes it possible to select authorization mechanisms for an entire application. These mechanisms must be implemented in the generators that are used to translate the application model to executable code and are addressed as constants (capital letters, numbers and underscores only; starting with a capital letter).

ekke implementier

Authorization mechanisms may be given propterties in the form of `key=value` pairs in brackets (see chapter 10 for details).

Syntax:

`authorization` ***`NAME_CONSTANT`*** `[` *`properties`* `]`

6.2 Constants

The mobaDSL is designed to deal with constants in a straightforward manner.

Constants of the following types may be defined:

- String (in single or double quotes)
- Numeric (positive integers only)
- Decimal (double precision floating point)

Constants are defined using the `const` keyword followed by the name of the constant (capital letters, numbers and underscores only; starting with a capital letter) and the desired value that shall be assigned to the constant (or by the name of another constant).

Referencing other constants by their name allows concatenation: The + sign is used to connect the parts that are concatenated into a new constant. An arbitrary number of parts can be used in a such a concatenation.

Note: In order to avoid confusion with the boolean values TRUE and FALSE, the enum integer values for zero and one have to be written as `00` and `01`, respectively.

copy-paste-error oder gehört das hierher?

Syntax:

```
const CONST_NAME value|otherConstant + value|otherConstant + ...
```

6.3 Datatypes

The mobaDSL allows the definition of datatypes. These are translated by the generator to the appropriate executable code. The exact behaviour of the generator can be controlled by the datatype definitions.

Datatypes may extend or customize "parent" datatypes, both inherited from a template or defined in the same application model; this is done with the `extends` keyword.

The mobaDSL allows the definition of "ordinary", primitive, array and "date" datatypes. Three "ordinary" datatypes are supported out of the box: strings, integers and decimal numbers. A string datatype is created with the flag `isString`, an integer datatype takes the `isNumeric` flag, and a decimal number (double precision floating point number) is marked with the flag `isDecimal`.

Ordinary datatypes are converted to their usual representation in the executable code (using wrapper types where applicable; e.g. Java `Integers`). Using the `isPrimitive` keyword, a datatype can be marked as primitive, resulting in its translation to a primitive datatype where applicable (e.g. Java `ints`).

The keyword `isArray` triggers the creation of a datatype that holds an array rather than a single instance.

"Date" datatypes are intended for dealing with time-related information: They can contain date, time and timestamp values. This is achieved with one of the following keywords: `isDate`, `isTime` or `isTimestamp`. In any case, a date format may be specified using a format string (or a constant).

Datatypes may be defined as enums using the `enum` keyword. See chapter 6.4 for details.

Furthermore, constraints may be declared for datatypes in parentheses after the `constraints` keyword. See chapter 11 for details about the available constraints.

Properties in the form of `key=value` pairs may be listed in brackets and added to datatypes. For both keys and values, both strings and constants may be used. See chapter 10.

Syntax:

```
datatype name extends parentName
    isArray
    isString|isNumeric|isDecimal isPrimitive
    isDate|isTime|isTimestamp ( formatString|formatConstant )
    enum enumDefinition
    constraints ( constraints )
    [ properties ]
```

6.4 Enums

An enum is a datatype that contains a set of predefined constants. Variables of this datatype must be equal to one of these values.

Enums declared in the mobaDSL are translated to the appropriate form of enums in the executable code. A datatype is made into an enum by using the `enum` keyword in the datatype declaration, followed by the definition of the literals in braces.

Enum literals are defined using the `lit` keyword. They associate a literal (usually in capital letters) with a string in quotes and an integer used to represent the value; for example `IN = ( "inbound", 00 )`. Expressions defining enum literals are separated by commas.

Note: In order to avoid confusion with the boolean values TRUE and FALSE, the enum integer values for zero and one have to be written as `00` and `01`, respectively.

Syntax:

```
datatype name
    enum {
        lit LITERAL = ( "literalString", valueInt ) ,
        lit LITERAL = ( "literalString", valueInt )
    }
```

6.5 Serialization Types

The mobaDSL offers the possibility to select serialization types (e.g. XML or JSON) for the transport of DTOs that are sent/received by REST services (cf. chapter 7.4). These serialization types must be implemented in the generators that are used to translate the application model to executable code and are addressed as constants (capital letters, numbers and underscores only; starting with a capital letter).

ekke implementier

Services may be given propterties in the form of `key=value` pairs in brackets (see chapter 10 for details).

Syntax:

`serialization` ***`NAME_CONSTANT`*** `[` *`properties`* `]`

7 Complex Application Features

7.1 DTOs

DTOs (Data Transfer Objects) are the another central feature of the mobaDSL. DTOs hold attributes and references and wrap them into a single object used for communication purposes, e.g. as the payload of a message. DTOs are not persisted.

For each DTO that is defined in an application, the corresponding executable code with all variable definitions, getter and setter methods, annotations etc. is automatically generated.

DTOs may extend other DTOs. In this case, they are derived from their parent DTO. That means that the properties, attributes and references of the parent DTO are inherited.

Furthermore, properties in the form of `key=value` pairs may be added in brackets (see chapter 10 for details).

Within braces, the features of the DTO (attributes and references) are defined (see chapters 8 and 9).

Syntax:

```
dto name extends parentName [ properties ] {
    dtoFeatures
}
```

7.2 Entities

Entities are (along with DTOs) the most complex elements in the mobaDSL. An Entity is an abstraction above a business object. An Entity is defined by its name and properties, references and operations. Generally, an Entity is an object which can keep a state about variables and references and can persisted.

For each Entity that is defined in an application, the corresponding executable code with all variable definitions, getter and setter methods, annotations etc. is automatically generated.

Entities may extend other Entities. In this case, they are derived from their parent Entity. That means that the properties, attributes and references of the parent Entity are inherited.

Furthermore, settings for caching may be defined for an Entity (see chapter 13.3 for details about caching), and properties in the form of `key=value` pairs may be added in brackets (see chapter 10 for details).

Within braces, the features of the Entity (attributes and references) are defined (see chapters 8 and 9), and the attributes that are indexed can be specified as a comma-separated list after the `index` keyword.

Syntax:

```
entity name extends parentName
            cache ( type=cacheType strategy=cacheStrategy )
            [ properties ] {
    entityFeatures
    index attributes
}
```

7.3 Queues

The mobaDSL uses queues to hold a series of elements and to process them in order. Queues hold references to other Entities or DTOs (or other queues).

For each queue that is defined in an application, the corresponding executable code with all variable definitions, getter and setter methods, annotations etc. is automatically generated.

Queues may extend other queues. In this case, they are derived from their parent queue. That means that the properties and references of the parent queue are inherited.

Furthermore, properties in the form of `key=value` pairs may be added in brackets (see chapter 10 for details).

Within braces, the references that the queue holds are defined (see chapter 9).

Syntax:

```
queue name extends parentName [ properties ] {
    queueReferences
}
```

7.4 REST Services

Applications running on mobile devices often communicate with web servers using RESTful interfaces. The mobaDSL offers a way to describe such services (and, provided appropriate generators, to generate the server-side code for these interfaces).

restCrud undocumented

On a basic level, the mobaDSL grammar allows the definition of REST services. These services can then be aggregated into entire REST workflows.

7.4.1 Custom REST Services

REST services are declared with the `rest` keyword. For services that are expected to deal with large amounts of data, a `bigData` flag may be set. Services may extend other services and can be given propterties in the form of `key=value` pairs in brackets (see chapter 10 for details).

For each REST service, headers, parameters, a method (GET, PUT, POST, DELETE) and DTO descriptions for request, response and error DTOs can be specified. The DTOs used by the service may be marked with an `isArray` flag and can be given a serialization type (using the serialization type's name after the `as` keyword; cf. chapter 6.5 for details about serialization types).

Syntax:

```
rest bigData name extends parentName [ properties ] {
    headers {
        param datatype name = value
    }
    paramters {
        param datatype name = value
    }
    method = GET|PUT|POST|DELETE
    requestDto = dto isArray    as serializationType
    responseDto = dto isArray    as serializationType
    errorDto = dto isArray    as serializationType
}
```

7.4.2 REST Workflows

REST services may be assembled into workflows. A workflow is created with the `restWorkflow` keyword. Workflows may extend other workflows and can be given propterties in the form of `key=value` pairs in brackets (see chapter 10 for details).

Syntax:

```
restWorkflow name extends parentName [ properties ] {
    service serviceName
}
```

7.5 Settings

The mobaDSL provides mechanisms to model application settings (such as user account information etc.). Settings are modeled as a set of attributes.

For each set of settings that is defined in an application, the corresponding executable code with all variable definitions, getter and setter methods, annotations etc. is automatically generated.

Settings may extend other settings. In this case, they are derived from their parent set of settings. That means that the properties and attributes of the parent DTO are inherited.

Furthermore, properties in the form of `key=value` pairs may be added in brackets (see chapter 10 for details).

Within braces, the settings attributes are defined (see chapter 8).

The keyword `active` controls which settings are used if there is more than one set.

Syntax:

```
active settings name extends parentName [ properties ] {
    settingsAttributes
}
```

8 Attributes

Attributes of Entities, DTOs or Settings definitions are references to datatypes or enums. They can be regarded as (and, in the executable code, are translated to) variables and are defined by the keyword `var` followed by a datatype (defined in the datatype section) and a name.

Attributes can be given multiplicity information in square brackets; the supported options are:

- `0..1` (default), shorthand `?`
- `1`
- `0..*`, shorthand `*`
- `1..*`, shorthand `+`

Furthermore, the following keywords are valid:

`var` The basic keyword for an attribute; defines a variable that can be used within the Entity and persisted. Appropriate getter and setter methods are created.

`lazy` triggers lazy loading for an attribute.

`transient` marks the attribute to be transient. Transient attributes are not persisted. Getter and setter methods are created.

`constraints` Constraints of various type may be defined for attributes. Appropriate validation mechanisms are generated in the executable code. Multiple constraints can be defined (comma-separated). See chapter 11 for details.

`domainDescription` defines an attribute that is used as a domain description. A domain description must be of a string datatype.

`domainKey` defines a domain key. Primitive datatypes may be used as domain keys.

`properties` Attributes can optionally be given additional information in the form of `key=value` pairs. This information can then be retrieved from the semantic model. The `key=value` pairs are defined in brackets after the attribute name. Multiple pairs can be defined (comma-separated).

Syntax:

```
var lazy transient domainKey domainDescription
    constraints ( constraints )
    datatype [ multiplicity ] name
    [ properties ]
```

9 References

The mobaDSL tracks relations between objects of the same kind using the concept of references. References are available for Entities, DTOs or Queue definitions. They can be regarded as (and, in the executable code, are translated to) variables of the same type as the containing object and are defined by the keyword `ref` followed by a type (of the same kind) and a name.

References can be given multiplicity information in square brackets; the supported options are:

- `0..1` (default), shorthand `?`
- `1`
- `0..*`, shorthand `*`
- `1..*`, shorthand `+`

Furthermore, the following keywords are valid:

var The basic keyword for an reference; defines a relation to an object of the same kind (Entity, DTO or Queue). Appropriate getter and setter methods are created.

cascade defines a "cascading" relation: If the "parent" object is disposed, the objects that it references with the `cascade` flag set are disposed as well.

lazy triggers lazy loading for a reference.

transient marks the reference to be transient. Transient references are not persisted. Getter and setter methods are created.

properties Attributes can optionally be given additional information in the form of `key=value` pairs. This information can then be retrieved from the semantic model. The `key=value` pairs are defined in brackets after the attribute name. Multiple pairs can be defined (comma-separated).

Syntax:

```
ref cascade lazy transient
    targetType [ multiplicity ] name
    [ properties ]
```

10 Properties

Properties in the form of arbitrary `key=value` pairs may be added to datatypes and attributes.

FRIENDS

11 Constraints

The mobaDSL supports the following types of constraints that can be defined for datatypes and attributes of Entities, DTOs and Settings definitions:

- `isNull` – asserts that a value is null
- `isNotNull` – asserts that a value is not null
- `min = valueDouble` – asserts that a value is greater or equal than a given value
- `max = valueDouble` – asserts that a value is less or equal than a given value
- `digits ( firstInt, secondInt )` – asserts that a floating point number has *firstInt* digits before the decimal point and *secondInt* digits after it
- `minLength = valueInt` – asserts that a string is at least a certain number of bytes long
- `maxLength = valueInt` – asserts that a string is at most a certain number of bytes long
- `regexp = "regexString"` – asserts that a string matches a regular expression

12 Inheritance

Inheritance ergänzen

13 Generators

13.1 Custom Generators

...

ekke

13.2 Friends

ekke

13.3 Caching

TBD ekke

14 Comments

Comments can be added anywhere in a .Entities text file and are copied over into the generated Java code. Comments are enclosed in `/* ... */`.

Comments before the `package` keyword are copied over to *all* generated Java classes – this is the place for copyright notices.