# DYNAMO

*Java Web Application Accelerator Framework*

**Bas Rutten**

*20 May 2016~~16 May 2016~~*

# Dynamo – Web Application Accelerator Framework

Open Circle Solutions BV.

Bas Rutten

Published: 2016-03-18

## TABLE OF CONTENTS

# 1 Introduction

The Dynamo Web Application Accelerator Framework is a software development framework developed by Open Circle Solutions that aims to increase productivity by using design principles such as convention over configuration, model-driven development and DRY (don't repeat yourself).

At the core of Dynamo is the concept of the *Entity Model*. The Entity Model describes the attributes and behaviour of an entity (or domain object) in your application. This Entity Model can then be used as the basis for creating forms, tables, search screens etc.

The *Entity Model* of an entity is automatically constructed based on the properties of the attributes of the entity (using sensible defaults as described by the convention over configuration principle) and can further be modified by using annotations and message bundle overwrites. The main goal is to reduce the amount of (boilerplate) code required to perform common actions like creating search screens and edit forms.

Complementing the *Entity Model* is a set of user interface components (widgets) that can be used to quickly construct screens for common use cases, and a number of base classes for the Data Access and Service layers.

The Dynamo framework is built around a number of proven and highly productive set of technologies:

- JPA2 for ORM
- QueryDSL for type-safe query generation
- Spring as the application framework
- Vaadin as the user interface framework
- Apache Camel for integration (optional)

# 2 THE ENTITY MODEL

## 2.1 Basics

In order to create the entity model, you need access to an *EntityModelFactory*. The *EntityModelFactory* is a Spring singleton and can be acquired by injection (@Inject or @Autowired).

You can then acquire the *EntityModel* for a certain entity by calling the *getModel(Class<?> clazz)* method. This will retrieve the entity model for the specified class, lazily constructing it if it is not there yet. Note that the entity model is effectively immutable and application-scoped (or more precisely, it has the Spring singleton scope, i.e. there is one instance per Spring application context).

This also means that the same entity model is (in principle) used by all screens within an application. Since this would be too restrictive in practice, it is possible to construct separate instances for separate screens or use cases, by calling the **getModel(String reference, Class<?> clazz)** method. This will construct the entity model based on the provided class, but it will allow you to override certain attributes using message bundle overrides (more on this later). The *reference* string is the unique identifier you assign to the model (if you just specify a class, then the simple name of the class is used as the reference).

The classes for which you create an Entity Model must inherit from the *nl.ocs.domain.AbstractEntity* class. See chapter 6 for details.

## 2.2 Entity level settings

The Entity Model supports a number of attributes that define how the entity itself is represented. These include:

- **displayName**: the name of the entity (e.g. "Car")
- **displayNamePlural**: the name of the entity, in plural form  (e.g. "Cars")
- **description**: textual description of the entity
- **displayProperty**: the name of the property to use when displaying the entity inside e.g. a combo box.
- **sortOrder**: how the entities are sorted by default when displayed in a table or list. The *sortOrder* consists of a comma separated list of attribute names and sort directions, e.g. "name asc, age desc". This should be familiar for anybody who has used SQL before.

## 2.3 Attributes

Every Entity Model consists of a number of *Attribute Models*. By default, an Attribute Model is created for every valid property of the entity. E.g. if you have an entity Person with properties "name" and "age", then the attribute model for the Person entity will contain two attribute models, one for "name" and one for "age".

The following rules apply for when constructing the attribute models:

- An attribute model will be created for every (public), non-static no-parameter getter-method that follows JavaBean naming conventions (e.g., "getAge()"; for Boolean properties, the getter may also start with "is").
- The entity class does not necessarily have to contain an actual field corresponding to the property. This allows you to create attribute models for read-only or composite properties (e.g. "getNameAndAge()" that concatenates the name and age as  a string).
- Certain attributes are ignored. Currently this includes only "version" (used for JPA optimistic locking) and "class" (every object has a "getClass()" method).
- Attributes can be simple (String, Integer, Long, enumerated types etc.) or complex (a reference to another entity, a collection of simple types, or a collection of other entities). Entity Model generation is nested, which means that if a property of an entity is again an entity, then an entity model for the nested property will also be generated. This entity model is separate from the non-nested entity model that would be constructed directly for the entity.
- Getters that are annotated with *@AssertTrue* or *@AssertFalse* are skipped (these are methods that are used for Bean Validations, not properties for the meta model).

An Attribute Model has a *name* attribute that is equal to the name of the property. This *name* can be used to retrieve the attribute model from the entity model:

```
AttributeModel getAttributeModel(String attributeName);
```

For a nested attribute model, the name of the model consists of the concatenation of the names of the unnested models. E.g. if you have a Person entity that has an attribute of type Address, then the "housenumber" Attribute Model of the address has the path "address.housenumber".

*Note: this should all makes perfect sense as it corresponds to the paths that are used in e.g. JPQL queries and by the Vaadin data binding.*

## 2.4 Defaults and attribute overrides

The Entity Model generation is based on sensible defaults and metadata. E.g. the value of the *type* attribute of an Attribute Model is directly taken from the type of the property, and certain other aspects e.g. whether the attribute is visible in a table or can be used in a search form are derived from this type (e.g. by default a complex attribute will not be visible in a table).

In addition to this, the Entity Model generation process will also take certain JSR-303 annotation (e.g. @Required) into account. A detailed explanation for each attribute will be given below.

If the default values are not sufficient, you can override them by using annotation:

- On the entity level, you can use the @Model annotation.
- On the attribute level, you can use the @Attribute annotation.

The *@Model* annotation can be placed only on the class level, e.g.:

```
@Model(displayProperty = "description")
public class Meeting extends AbstractEntity<Integer> {
```

The *@Attribute* annotation can be placed either directly on the property, or on its getter method. Annotations placed on the getter method precede those placed on the property, in order to easily allow you to override behaviour in subclasses. Within a single entity, you can use both access types interchangeably.

## 2.5 Message bundle overrides

The annotation override mechanism is quite powerful, but it has some downsides. E.g. it hard-codes certain string values (display name, description) into your application and it does not directly allow for internationalization. It also only allows you to override the behaviour of the "default" Entity Model that is based directly on the class, and not the behaviour of any derived Entity Models.

If you need to override the behaviour of a derived Entity Model, you can use the message bundle mechanism to achieve this. Message bundle overrides must be placed in the *entitymodel.properties* file (create a locale-specific version of this file if you need to; the normal Java message bundle mechanic is fully supported).

Message bundle entries must have the following structure:

**[Reference].[Attribute Model Name].[Attribute]=[Value]**

Where:

- **[Reference]** is the reference to the ~~attribute~~ entity model. This is the simple class name of the entity for a standard entity model, and the user-provided reference for a non-standard model.
- **[Attribute Model Name]** is the (possibly nested) name of the attribute model. This is empty in case you are directly overriding a setting of the Entity.
- **[Attribute]** is the identifier of the attribute that you want to modify. For a full list, see the *EntityModel* class which contains constants that denote the possible values (or refer to the sections below)
- **[Value]** is the desired value

The [Attribute Model Name] part is optional and must be omitted when you want to directly set an attribute of the Entity Model itself.

Some examples:

**Person.displayName=Persoon**

Sets the display name of the Person entity to "Persoon"

**CustomPerson.displayName=Bijzonder Persoon**

Sets the display name used by the "CustomPerson" entity model to "Bijzonder Persoon"

**Person.name.visible=true**

Sets the visibility of the "name" attribute model to true

**Person.address.street.readOnly=true**

Sets the "read only" attribute of the "address.street" attribute model (a nested model) to false.

Please observe the following:

- For Boolean properties, use the (lower case) values "true" and "false".
- For numbers, simply type the String representation of the numeric value. Use the period "." as the decimal separator.
- For enumeration values, use the upper-case String representation of the enumeration value.
- For dates, use the formats that are specified by the system properties:
    - "default.date.format" (dd-MM-yyyy) for dates
    - "default.time.format" (HH:mm:ss) for times
    - "default.datetime.format" (dd-MM-yyyy HH:mm:ss)

# 3 DETAILED SETTING EXPLANATIONS

## 3.1 DisplayName

In message bundle: **displayName=[desired value]**

The **displayName** setting determines how the property will be named on-screen. By default, it is derived from the property name, replacing camel casing by spaces, e.g. "mininumAge" will be translated to "Minimum Age".

## 3.2 Description

In message bundle: **description=[desired value]**

The **description** setting determines the value of the tooltip that the user will see when hovering over the input field for the property.

If not set, it will default to the **displayName.**

## 3.3 Prompt

In message bundle: **prompt=[desired value]**

The **prompt** setting determines the value of the prompt that shows up inside the editable field for the property.

If not set, it defaults to the **displayName.**

## 3.4 Type

The *type* setting represents the Java type of the property. It cannot (for obvious reasons) be overwritten.

## 3.5 Attribute Type

The "attribute type" setting is a classification of the type of the property. It is determined automatically during the Entity Model generation process and can have the following values:

- BASIC: represents a simple property like a String, a number, a date etc.
- DETAIL: a property that appears as a @OneToMany or @ManyToMany relation in the entity class, e.g. the *orderLines* attribute inside an *Order* attribute will be considered a DETAIL attribute. *You can also think of a Detail attribute as a collection.*

- MASTER: a property that appears as a @OneToOne or @ManyToOne in the entity class.
- LOB: a property that is annotated with @Lob and represents a large binary object (like a file or an image)
- EMBEDDED: used during the entity model construction to handle embedded properties (using the @Embedded annotation). This will be covered in the advanced section.
- ELEMENT_COLLECTION: a property that is annotated with the @ElementCollection annotation, i.e. a collection of Strings or Integers.

The attribute type, in combination with the (Java) type are largely responsible for how a certain property will be displayed on-screen in an edit form:

- For a BASIC property, a simple edit component will be displayed, based on the type of the property:
  - For String and numeric fields, a simple Text Field will be rendered. For a String property, you can use the *textFieldMode* setting in order to render a text area instead.
  - For a Boolean, a check box will be rendered
  - For a (java.util.)Date, a date picker will be rendered (for special cases, see the section on the DateType setting)
  - For an enumeration a combo box will be created. You can use the message bundle to specify translations for the enum values (more on this below)
- For a LOB property, an upload field will be created. This field can be further modified in several ways.
- For a MASTER property, by default a combo box will be created that contains all the possible values (as retrieved from the database). You can potentially replace this by a lookup field or *ListSelect* by setting the *selectMode* attribute.
- For a DETAIL property, by default a *ListSelect* will be rendered. A ListSelect is a select component that spans multiple lines and allows you to select multiple items.
- For an ELEMENT_COLLECTION property, a simple table is rendered that allows you to modify the collection of strings or numbers.

For a search form the rendering is a bit different:

- For a BASIC property:
  - For a property of type string, a simple text field is created. This text field can be used to perform a search. You can use additional properties to switch the case sensitivity and whether to allow prefix or substring matches.
  - For a numeric property or a Date property, two search fields are generated. These allow the user to perform an interval search (return all values that are higher than the value in the first field and lower than the value in the second field). This interval search is inclusive (i.e. the boundaries match).

    - For a Boolean property, a checkbox is displayed, that contains three values: true, false, and "no value".
    - For an enumeration, a combo box containing all values of the enumeration is displayed.
- For a MASTER property, by default a combo box containing all values of the master is displayed. You can use the *selectMode* setting to replace this by a lookup field or a ListSelect.
- Searching on DETAIL properties is not supported. LOB properties cannot be used in a search form.

Note that any text fields will automatically have the appropriate converters and validators associated to them.

The attribute type setting also determines whether the property will be visible by default:

- In a table, by default only BASIC attributes will be visible. Use the *showInTable* attribute in order to show a complex attribute in a table
    - For a MASTER property, the value of its "displayProperty" property will be displayed.
    - For a DETAIL property (remember, it represents a collection!), the values of the "displayProperty" properties of all individual entities in the collection will be displayed, separated by commas.
- In an edit screen, any MASTER or DETAIL properties will by default not be displayed. You can change this by setting the *complexEditable* property to true. Note that in many cases you will also have to implement some custom logic for dealing with these properties.

With regard to the display of enumeration values: the display value of an enumeration will be taken from the message bundle. The following format must be used:

**[Simple Class Name].[Enumeration Value]=[desired value]**

E.g.:

```
PriorityType.LOW=Low
PriorityType.MEDIUM=Medium
PriorityType.HIGH=High
PriorityType.ULTRA=Ultra
```

## 3.6 Visible

In message bundle: **visible = true | false (or SHOW | HIDE)**

The **visible** setting determines whether a property will be displayed *in an* edit *form*. It is not to be confused with the **showInTable** attribute that governs whether a property shows up in a table.

By default, all properties will have **visible** set to true, except for the "id" property which is reserved for a technical primary key and will by default be hidden from the user.

Note that if you set the *visibility* setting to false using the annotation override, the *showInTable* setting will also be set to false. If you *do* want the property to show up in a table view, explicitly set the *showInTable setting* to true.

Note: instead of "true" you can also use the value "SHOW" and instead of "false" you can also sue the value "HIDE".

## 3.7 ShowInTable

In message bundle: **showInTable = true | false (or SHOW | HIDE)**

The **showInTable** setting determines whether a property will be displayed in a table.

By default, all BASIC properties will have **showInTable** set to true, except for the "id" property which is used for a technical primary key and will by default be hidden from the user.

For all other properties you have to manually set the attribute to "true" (or "SHOW").

## 3.8 ReadOnly

In message bundle: **readOnly = true | false**

The **readOnly** setting determines whether a property is read only. Read-only properties will show up in an edit form (or in an editable table) but cannot be modified. This attribute has no effect on the rendering of the property in a read-only table.

Note that an attribute which has the value or the *readOnly* attribute set to "false" will *not* show up in an edit form when creating a new entity.

## 3.9 DefaultValue

In message bundle: **defaultValue = [desired value]**

The **defautlValue** setting can be used to set the default value that appears in an edit form when creating a new entity.

You always specify this as a String; if the value is to be converted to a decimal number, use the period (".") as the decimal separator. For enumeration values, use the upper-case String representation of the desired value.

For date values, use the String representations according to the system properties *default.date.format (dd-MM-yyyy), default.time.format (HH:mm:ss) and default.datetime.format (dd-MM-yyyy HH:mm:ss)*.

## 3.10 ComplexEditable

In message bundle: **complexEditable = true | false**

This setting can be used to determine whether a complex property (i.e. of type MASTER or DETAIL) shows up in an edit form. Note that in some cases, some additional code will be needed to properly deal with these properties.

This setting defaults to *false*.

For an attribute of type DETAIL, by default a ListSelect component will be rendered. This is a component that allows the user to select multiple items from a list. This generally works fine for many-to-many relations but will not work if the~~y~~ir details will have to be manually created. In this case, you need to write some custom code. This will be covered later in the manual.

For an attribute of type MASTER, by default a combo box will be rendered. You can modify this by changing the value of the *SelectMode* attribute. It is also possible to provide a custom implementation (this will be covered later in the manual).

## 3.11 DisplayFormat

In message bundle: **displayFormat = [desired value]**

The **displayFormat** setting indicates how certain values will be formatted. It is currently supported for values of type java.util.Date only.

The value of the *displayFormat* attribute must be a valid Java data/time formatting instructions, e.g. “dd-MM-yyyy”, but you can also use different separators like “dd/MM-yyyy” or use formats like “yyyy-MM-dd”.

## 3.12 TrueRepresentation and FalseRepresentation

In message bundle: **trueRepresentation = [desired value]**

In message bundle: **falseRepresentation = [desired value]**

The **trueRepresentation** and **falseRepresentation** settings can be used to modify how a Boolean value is displayed in read-only mode. By default, such a value will simply by displayed as “true” or “false”, but these settings can be overruled by setting respectively the **trueRepresentation** and **falseRepresentation** values.

This setting does nothing in edit mode, since in that case a checkbox will be rendered.

## 3.13 Image

In message bundle: **image = true | false**

This setting can be used on a LOB property to specify whether it represents an image. By default this setting has the value *false*. If set to true, the application will try to render a preview image of the contents of the property.

## 3.14 AllowedExtensions

In message bundle: **allowedExtensions = [desired value]**

This setting can be used to set the extensions of the files that are accepted by the file upload component that is generated for a LOB property. By default its value is empty, which means there are no restrictions on the file type.

The value can be set to a comma-separated list of supported extensions, e.g. “bmp,jpg,png”. Note that you must not include the “.” character.

## 3.15 MainAttribute

In message bundle: **main = true | false**

The Boolean setting **mainAttribute** can be used to specify that a certain property is the *main* property of an entity. The main property is the property that will be used to construct the title of the entity in an edit form.

By default, the first encountered property of type String will be marked as the main attribute.

In addition, the *searchable* attribute of the main property will be automatically set to *true*.

## 3.16 Sortable

In message bundle: **sortable = true | false**

The **sortable** setting can be used to specify whether a table can be sorted on a particular property. By default it is set to *true* for all properties.

## 3.17 Percentage

In message bundle: **percentage = true | false**

The **percentage** setting is used to indicate whether a numeric value represents a percentage. By default this attribute has the value *false*. If set to true, then the value of the property will be displayed with a “%” sign following it, both in read-only and edit mode.

The percentage sign is purely cosmetic, the actual value of the property is not converted or changed in any way.

## 3.18 Precision

In message bundle: **precision = [desired numeric value]**

The **precision** setting determines the number of digits will be shown behind the decimal separator when displaying non-integer numbers. By default it is set to 2.

## 3.19 Required

In message bundle: **N/A**

The **required** setting determines if a property is a required property. Its value is taken by default from the presence or absence of the JSR303 "@NotNull" annotation.  Currently, this setting cannot be modified using annotation or message bundle overrides.

## 3.20 Searchable

In message bundle: **searchable = true | false**

The **searchable** setting determines whether a property will show up in a search form on a search screen. By default it is set to *false*.

## 3.21 SearchCaseSensitive

In message bundle: **searchCaseSensitive = true | false**

The **searchCaseSensitive** setting determines whether search operations on the property are case sensitive. The default is false.

## 3.22 SearchPrefixOnly

In message bundle: **searchPrefixOnly = true | false**

The **searchPrefixOnly** setting determines whether search operations on the property check only for a prefix match. If this is set to true, then searching for e.g. "a" will only match "almond" ("a" appears at start) but not "walnut" ("a" appears in the middle). If set to false, then "a" will match both "almond" and "walnut".

By default this attribute is set to *false*.

## 3.23 DateType

In message bundle: **dateType = TIMESTAMP | DATE | TIME**

The **dateType** setting can be used to determine how a property of type java.util.Date will be edited:

The allowed values are:

- TIMESTAMP: In this case the application renders a date picker that includes a time selection component.
- DATE: in this case the application renders a date picker without a time selection component.
- TIME: in this case a custom time selection component is rendered.

By default, the value of the **dateType** setting is derived from the Java Persistence **@Temporal** annotation. If this annotation is not present, then the value DATE is used.

*Note: we currently only support java.util.Date, not java.sql.Date or the Java 8 Date types.*

## 3.24 SelectMode

In message bundle: **selectMode = COMBO | LOOKUP | LIST**

The **selectMode** setting is used to specify how a field for editing an attribute of attribute type MASTER will be rendered. By default, a combo box will be rendered for such an attribute, but if the setting is set to the value LOOKUP then a lookup field (which allows the user to bring up a search dialog) will be rendered. If you set the value to LIST then a multi-line ListSelect component will be rendered.

## 3.25 TextFieldMode

In message bundle: **textFieldMode = TEXTAREA | TEXTFIELD**

The **textFieldMode** setting can be used to specify whether to render either a text field or a text area for editing an attribute of type String. The default is *TEXTFIELD*.

## 3.26 Email

In message bundle: **N/A**

The **email** setting can be used to specify that a field must contain a valid email address. It is set to *true* if the property is annotated with the (custom) @Email annotation.

## 3.27 MaxLength

In message bundle: **maxLength = [desired integer value]**

The **maxLength** setting can be used to specify the maximum allowed length of a field. Its value is derived from the *max* value of the @Size annotation.

Note that there is a special case: in case you have a property that is annotated with @ElementCollection, then the @Size annotation will govern the size of the collection, not the maximum length of the individual items in the collection. In this case, use the **maxLength** setting to set the maximum length of the individual items.

## 3.28 MinLength

In message bundle: **minLength = [desired integer value]**

The **minLength** setting can be used to specify the minimum allowed length of a field. Its value is derived from the *min* value of the @Size annotation.

Note that there is a special case: in case you have a property that is annotated with @ElementCollection, then the @Size annotation will govern the size of the collection, not the minimum length of the individual items in the collection. In this case, use the **minLength** setting to set the minimum length of the individual items.

## 3.29 URL

In message bundle: **url = true | false**

The **url** setting can be used to specify that a certain String property must be rendered as a clickable URL.

The default value is *false*. If set to *true* then a validator will be added to the field (when in edit mode) that checks if the entered value is a valid URL. Also, in read-only mode the application will render a clickable URL containing the value of the attribute.

# 4 ATTRIBUTE ORDERING AND GROUPING

## 4.1 Attribute ordering

In message bundle: **attributeOrder=[comma separated list of attribute names]**

By default, the properties of an entity will be displayed in the order in which they appear in the class file. This can be overruled by using an **@AttributeOrder** annotation or setting the **attributeOrder** via the message bundle.

The **@AttributeOrder** annotation takes a single parameter, named *attributeNames* which contains an array of field names – the order in which the attributes appear in the array is the order in which they will appear in the application.

You can achieve the same effect by including a message like **Person.attributeOrder=name,age,gender** in the message bundle (use commas to separate the values). The message in the bundle will overwrite the ordering set by @AttributeOrder.

The ordering does not have to be complete; if you leave out any attributes, then those will be placed (in the normal order) *after* any attributes that are explicitly mentioned in the annotation or the message bundle.

## 4.2 Attribute grouping

In addition to attribute ordering, the attributes can also be grouped together. In order to achieve this, you can include an **@AttributeGroups** annotation on your entity class, which can in turn include any number of **@AttributeGroup** annotations.

Each **@AttributeGroup** annotation contains the name of the group and an array that contains the names of the properties that must be included in the group. As an example, consider:

```
@AttributeGroups(attributeGroups = {
    @AttributeGroup(displayName = "Main", attributeNames = { "code", "name",
"parentLocation",
        "channel", "region", "freeTag", "weeklyTurnover", "participation",
        "expectedQuantity", "periodBegin", "periodEnd",
"periodOpenIndication",
        "numberOfDisplays", "display", "deliveryTimeInWeeks", "packageType"
}),
    @AttributeGroup(displayName = "Address", attributeNames = { "address",
"zipCode", "city",
        "country", "telephone", "fax", "email", "manager" }) })
public class Store extends ProgrammeLocation {
```

The above defines two attribute groups, “Main” and “Address”, each containing a number of attributes.

When you want to achieve the same using a message bundle, then this will look as follows:

**Store.attributeGroup.1.displayName=Main**
**Store.attributeGroup.1.attributeNames=<comma separated list of attribute names>**
**Store.attributeGroup.2.displayName=Address**
**Store.attributeGroup.2. attributeNames =<comma separated list of attribute names>**

I.e. you include two messages for every attribute group: one containing the display name and one containing the attribute names as a list of comma-separates attribute names. The messages are numbered starting at "1".

Note that the attribute grouping is only used to determine which properties to group together, not to determine the order in which the attributes appear: the order still follows from the attribute ordering described in section 4.1.

# 5 ADVANCED TOPICS

## 5.1 Embedded attributes

The Entity Model framework supports dealing with properties that are embedded in an entity, via the **@Embedded** annotation.

As an example, consider a Person entity that has as a property named "address" which is an embedded object of type Address that has the fields "street", "city", and "country".

The Entity Model framework translates the properties of the embedded object (street, city, country) to properties of the embedding object. The name of these attribute will reflect the nesting, i.e. the attributes will have "address.street", "address.city" and "address.country" as their names.

To the user of the framework, this is handled transparently, since the embedded properties can be used in exactly the same way as the normal attributes.

## 5.2 Nested entity models

The Entity Model framework supports dealing with nested entities. When the Entity Model framework generates an entity model for an entity, it automatically creates nested entity models for all complex properties it encounters. This is currently supported up to three levels deep.

Note that the entity model that is created for the nested entities is a separate model from the top-level model for the entity. So if you directly create a model for the "Address" entity, this is a different model than the nested model for "Person.address".

Note that some settings behave differently for nested entity models. E.g. for any properties of nested entities, the *searchable* and *showInTable* settings will be set to false.

You can override setting on nested attribute models in the same way as you can override attributes of non-nested entities, i.e. by including a message in the message bundle that contains the full path to the property (e.g. **Movie.director.name.displayName**="Director Name").

## 5.3 Element collections

The Entity Model framework also supports dealing with "element collection" properties, i.e. properties that are collections of simple types (currently, only String and Integer are supported) and that are annotated with the **@ElementCollection** annotation.

For these properties, the application will automatically generate a simple table component that allows you to add items to, remove items from, and modify items in the collection. You can use the minLength and maxLength settings to modify the minimum allowed length and maximum allowed length of the individual items (in case of a collection of Strings).

# 6 Data access, Service layers and general concepts

## 6.1 Data access layer and entities

Dynamo has certain requirements with regard to the Data Access layer and Entity classes that are used in applications developed with the framework.

All Entity classes (classes that map to a table in the database) must inherit from the *AbstractEntity* class. This means that they inherit a *version* field (used for optimistic locking) and an *id* field that denotes the technical primary key. The type of this id field is configurable via the generic type parameter of the *AbstractEntity* class.

An example entity will look like this:

```
@Entity
@Table(name = "meetings")
@AttributeOrder(attributeNames = { "description", "desiredLocation",
"meetingDate", "startTime",
    "endTime", "attendees", "whiteboard", "videoConferencing",
"phoneConferencing", "priority" })
@Model(displayProperty = "description")
public class Meeting extends AbstractEntity<Integer> {
```

For every Entity class you must create a Data Access Object (DAO) interface and the accompanying implementation. The DAO must inherit from BaseDAO:

```
public interface MeetingDao extends BaseDao<Integer, Meeting> {
}
```

And the implementation must inherit from BaseDaoImpl:

```
@Repository("meetingDao")
public class MeetingDaoImpl extends BaseDaoImpl<Integer, Meeting> implements
MeetingDao {

  private static QMeeting qMeeting = QMeeting.meeting;

  @Override
  public Class<Meeting> getEntityClass() {
    return Meeting.class;
  }

  @Override
  protected EntityPathBase<Meeting> getDslRoot() {
    return qMeeting;
  }
```

The minimal implementation must implement just two methods: getEntityClass() which returns the type of the entity that is managed by the DAO, and getDslRoot() which returns the QueryDSL root.

QueryDSL is a framework that is used by the Dynamo Framework in order to create type-safe queries. Basically, what QueryDSL does is create a QueryDSL class for every entity class in your application. When developing in Eclipse, the IDE will automatically generate the appropriate classes. You can also run a command line Maven build in order to generate them.

Finally, note that the DAO implementation class is annotated with @Repository, which will register it as a Spring bean.

## 6.2 Service

In addition to developing a DAO for entity, you must also create a service object. This service object in its basic form will serve as a delegate to the DAO, but it is also the place where you can place business logic.

The declaration of a service interface is very easy; the service must extends *BaseService*.

```
public interface MeetingService extends BaseService<Integer, Meeting> {
```

The implementation is equally simple:

```
@Service("meetingService")
@Transactional
public class MeetingServiceImpl extends BaseServiceImpl<Integer, Meeting>
implements MeetingService {


  @Inject
  private MeetingDao meetingDao;

  @Override
  protected BaseDao<Integer, Meeting> getDao() {
     return meetingDao;
  }
```

In its most basic form, you only have to extend the BaseServiceImpl class and inject the appropriate DAO. This DAO must also be returned by the *getDao()* method.

Finally, note that the service must be annotated with @Service, registering it as a Spring bean.

## 6.3 Common service methods

The BaseService (and BaseDao) class offer a number of very commonly used methods that should take care of the most basic data retrieval and storage needs:

- **long count()** -> return the number of entities in the database
- **long count(Filter filter, boolean distinct)** -> returns the number of entities that match a certain filter
- **T createNewEntity()** -> creates a new entity
- **void delete(List<T> list**) -> Deletes a list of entities
- **void delete(T entity)** -> Deletes a single entity
- **List<T> fetch(Filter filter, int pageNumber, int pageSize, FetchJoinInformation[] joins, SortOrder… orders)** -> fetches a page of data
- **T fetchById(ID id, FetchJoinInformation… joins)** -> fetches an entity and its relations baded on a primary key
- **List<T> fetchByIds(List<ID> ids, FetchjoinInformation[] joins, SortOrder… orders)** -> Fetches a page of data based on the IDs
- **T fetchByUniquePropertyId(String property, Object value, Boolean caseSensitive, FetchJoinInformation[] joins)** -> Fetches an entity based on a unique property value.
- **List<T> find(Filter filter, int pageNumber, int pageSize, SortOrder… orders)** -> retrieves a page of data based on the provided filter.
- **List<T> find(Filter filter, SortOrder… orders)** -> Retrieves a page data of data based on the provided filter.
- **List<T> findAll(Order… orders)** -> Retrieves all entities of a certain type. Use with caution
- **T findById(ID id)** -> Find an entity based on its primary key
- **T findByUniquePropertyId(String property, Object value, Boolean caseSensitive)** -> Retrieves an entity based on a unique property value.
- **List<ID> findIds(Filter filter, SortOrder… orders)** -> Returns a list of IDs that match the provided filter, sorted according to the provided sort orders.
- **Class<?> getEntityClass()** -> Returns the class of the entity that is managed by this service.
- **T save(T entity)** -> Saves an entity
- **List<T> save(List<T> entities)** -> Saves a list of entities

## 6.4 Fetching and paging

The Dynamo framework is built around the concept op fetching data (using fetch join queries) whenever possible. The philosophy behind this is that it is usually much faster to fetch all required data using a single relatively big query, rather than performing numerous smaller queries to achieve the same result.

For this reason, we recommend to keep the use of eager fetching to an absolute minimum and use lazy fetching combined with fetch joins whenever possible.

The framework supports a number of methods that make it possible to fetch data based on a primary key or collection of keys, and also allow you to specify with relations to fetch as part of the query.

Note e.g. the following method defined in *BaseService*:

```
public T fetchById(ID id, FetchJoinInformation... joins);
```

As you can see, this method accepts a vararg parameter that specifies which relations to fetch. If left empty, the application will use the default setup, which you can specify by overriding the *getFetchJoins* method in the entity's DAO, e.g.:

```
  @Override
  protected FetchJoinInformation[] getFetchJoins() {
    return new FetchJoinInformation[] { new
FetchJoinInformation("attendees"),
          new FetchJoinInformation("desiredLocation"),
          new FetchJoinInformation("assignedLocation"), new
FetchJoinInformation("room"),
          new FetchJoinInformation("organisation") };
  }
```

This means that whenever you perform a fetch query using a standard service method, and you do not explicitly specify which relations to fetch, all relations returned by this method will be fetched.

Take great care here not to include any substantially large relations, since this can lead to poor performance.

When creating user interface components (see later), you will generally be able to specify which relations to fetch for the data in that user interface component. Once again, when you do not specify anything, the defaults defined in the DAO will be used.

Note that if you create a component that contains a tabular display of data, you can specify the way in which the tabular display will be filled. There are two options here:

- ID_BASED (FETCHING) – As described above. The application will execute a query that will retrieve the primary keys of the entities to be displayed, followed by a query that fetches a number of these entities (and their relations) based on these primary keys and information about which relations to fetch.
- PAGING – The application will first execute a query to determine the amount of entities, and will then use a paging query (using *firstResults* and *maxResults* in order to retrieve a subset of the desired entities). This approach *also* supports the fetching of associated relations, but take care that you must only fetch many-to-one or one-to-one relations in this fashion. This is because if you fetch one-to-many or many-to-many relations in this way, the result set will contain multiple rows per entity, which clashes with the *firstResults* and *maxResults* settings and will cause Hibernate to fetch a large data set and do the sorting in memory. This is often horribly inefficient.

Note that in both cases, the table is filled in a lazy fashion – only a small subset of the available data will be retrieved. Which approach is best depends on the situation – if you have a large data set and no relations to fetch then paging is preferred. If you have a lot of relations to fetch (or if you must fetch any one-to-many or many-to-many relations), use the id-based approach.

## 6.5 Validation

The validation facilities in the Dynamo Framework are based on JSR 303 (Bean Validation) standard: in order to express validation rules, simply use the standard annotations (@NotNull, @Size, @Min etc.) on the properties of your entity.

You can also use @AssertTrue and @AssertFalse to express more complex (inter-field) validation rules, or write your own validations by implementing the *ConstraintValidator* interface.

Custom validation messages can be included in the *ValidationMessages.properties* message bundle.

When you want to save an entity, it is automatically validated against these validation rules, and an *OCSValidationException* will be thrown if any validations fail.

Also note that when you create a standard edit form, the appropriate validators are automatically assigned to each form field based on the JSR 303 validation rules. So, if you enter a value of 1000 in a field that has "999" as the maximum value, a validation error message is automatically displayed.

### 6.5.1 Checking for identical entities

There is one additional feature with regard to validation that deserves special mention. In case you have an entity that contains a logical primary key (either a single field or a combination of fields) the framework provides a fairly easy way to check for possible duplicates. To do so, you only have to override the *findIdenticalEntity* method from the *BaseServiceImpl* in your service implementation class.

This method takes an entity as its only parameter and inside the method body, you can perform a query to check if there already is an entity that has the same values for the unique field or combination of fields.

Consider the following example that checks if there already is a product that shares a programme and a code with the product being validated:

```
  @Override
  protected Product findIdenticalEntity(Product product) {
    return productDao.findByProgrammeAndCode(product.getProgramme(),
product.getCode());
  }
```

Note that you do not have to check if the entity being returned is the same one as the entity being validated, the framework will take care of this for you.

## 6.6 Filtering

Both the user interface and the service layer make use of a Filter mechanism in order to limit the result sets returned by certain queries (e.g. the *count* and *find* methods of the BaseService).

This mechanism is based on the mechanism provided by Vaadin: for every Vaadin container, you can add one or more Filters (instances of

**com.vaadin.data.container.Filter**) which can be used to restrict the data set that is contained in it.

Vaadin offers a number of Filters, e.g. for comparison (Equal, Greater, Less, etc.), String comparison (Like, SimpleStringFilter), negation (Not) and aggregation (And, Or)

The Dynamo framework leverages this functionality and expands on it, by adding several filters (Contains, In, Modulo) that are not available in Vaadin.

Most user interface components in Dynamo that can be used to display a collection of data have the option to set a (list of) Vaadin filters that are always applied to the data (regardless of any further user input).

In addition, some components offer the developer the option to specify a *fieldFilter* map. This map contains key/value pairs that map an attribute name (from the Entity Model framework) to a Vaadin filter. This mechanism can be used to restrict the values that appear in e.g. a combo box or a lookup field in an edit form.

Note that in the UI layer the framework uses the Vaadin filters (*com.vaadin.data.container*), and in the service layer and DAO layer, the OCS implementation (*nl.ocs.filter*) is used. The framework converts the filters when needed.

As stated before, the Vaadin framework lacks some filter that are offered by the OCS framework. If you want to use on of the filters that is offered by the OCS framework but not by Vaadin, you can add a dummy filter in Vaadin and replace this (typically in a method in the DAO implementation class) by the real OCS filter using the *FilterUtil.replaceFilter* method.

# 7 Project Setup

## 7.1 Project structure

By default, projects created using the Dynamo Framework consist of a root project (with a root pom) that contains three sub-projects: *domain, core* and *ui:*

- The *domain* subproject contains the domain classes.
- The *core* subproject contains the service and business logic classes.
- The *ui* project contains the user interface.

Each subproject follows the default structure of a Maven project and thus has four source folders:

The following directory structure shows how the projects are organized:

- domain
    - src/main/java
    - src/main/resources
        - META-INF
            - **entitymodel.properties** the message bundle used to configure the entity model.
        - **application.properties** contains the application properties
        - **messages.properties** the message bundle used for internationalization.
        - **ValidationMessages.properties** the message bundle used for configuring Bean Validation error messages
    - src/test/java
    - src/test/resources
        - META-INF
            - **testApplicationContext.xml** the Spring configuration file that is used when running integration tests.
- core
    - src/main/java
    - src/main/resources
        - META-INF
            - Spring
                - **applicationContext-common.xml** contains the Spring configuration file for common components (that should be available both during tests and during runtime)
                - **camelContext.xml** contains the definition of the Camel routes.
    - src/test/java
    - src/test/resources
- ui
    - src/main/java
    - src/main/resources

        - **menu.properties** the message bundle used for configuring the menu (more on this later)
    - src/main/webapp
        - WEB-INF
            - **applicationContext.xml** the main Spring configuration file
            - **web.xml** the web deployment descriptor
    - src/test/java
    - src/test/resources

If you look closely at the above, you will notice that there are two kinds of configuration files needed to make the application function properly: message bundles and Spring configuration files. We will cover these two types of files in more detail in the following sections

## 7.2 Message bundles

A Dynamo application uses a number of message bundles (see the tree in the previous section for information on where these bundles are located). All of these message bundles are made available to the Spring Framework and you can retrieve a message from them using the *MessageFactory* which is a Spring-managed singleton bean that you can inject into your services. Note that many standard components already have a reference to this *MessageFactory*.

The message bundles used in the application all serve different purposes:

- *messages.properties* is the message bundle that must be used for all messages that actually appear on your screens. E.g. if you want a button to show the text "Click me" then you could include a message like

  **mybutton.caption=Click me**

  In the message bundle and then use the **messageService.getMessage("mybtton.caption")** method in order to retrieve the message. Note that you can use the standard features of a message bundle to provide internationalization, e.g. you can create a message bundle "messages_fr.properties" and fill that with the French translations of your messages. This bundle will then be picked up if the locale of the application is set to French.

- *ValidationMessages.properties* is the message bundle used configuring Bean Validation error messages. You can refer to messages from this bundle in the following way. Say that in your bundle you have the following message:

  ```
  product.aliases.different=Duplicate product aliases found. Please
  make sure they are all different
  ```

  Then in the code you can refer to this message by placing the message name within curly brackets:

  ```
  @AssertTrue(message = "{product.aliases.different}")
  ```

```
public boolean isAliasesDifferent() {
```

- *entitymodel.properties* is the message bundle that is used to override the default behaviour of the Entity Model factory. See section 2.5 for instructions on how to use this override mechanism.
- *menu.properties* is the message bundle that is used to configure the structure of your menu. It contains both the textual descriptions of menu items, and the structure of the menu. More on this can be found in chapter 8.

The *MessageService* provides a number of methods for retrieving messages. Some of these are used internally be the framework and should not be used directly. The following methods are intended for developers:

- **getMessage(String key)** retrieves a message based on its key, using the application locale. If no message is found then a warning message will be returned.
- **getMessage(String key, Object… args)** retrieves a message based on its key, using the application locale, and using the specified parameters. If the message contains placeholders ({0}, {1}, {2} etc) these will be replaced by the provided parameters. T
- **getMessage(String key, Locale locale, Object… args)** retrieves a message based on its key, and using the specified parameters, for the specified locale. If the message contains placeholders ({0}, {1}, {2} etc) these will be replaced by the provided parameters.

If a message with a certain key cannot be found, then a default message that returns that no message with the given key can be found, will be returned. If you do not want this behaviour, you can use the **getMessageNoDefault** version of the method instead.

## 7.3 Spring configuration

The application uses several Spring configuration files. These are defined as follows:

- *applicationContext.xml* -> This is the main configuration file that it used to bootstrap the Spring framework when actually running the application. It is located in the WEB-INF directory. It contains the beans that are needed for communication with an actual Postgresql database.
- *applicationContext-common.xml* -> This file contains the configuration for the common beans that are both used by the main application and while running integration tests. This includes things like the configuration for the entity manager factory, the message service, the property placeholder mechanism etc. This file is including (using Spring's *import* mechanism) into the *applicationContext.xml* file.
- *testApplicationContext.xml (domain)* -> this file, which is located in the **src/test/resources/META-INF** directory of the *domain* subproject, contains the beans that are used for running integration tests from the *domain* subproject. It

is a standalone file that does not include any other files and sets up an in-memory database.

- *testApplicationContext.xml (core)* -> this file, which is located in the **src/test/resources/META-INF** directory of the *core* subproject, contains the beans that are used for running integration tests from the *core* subproject.

Take special note that the *applicationContext-common.xml* and *applicationContex.xml* file makes use of the Spring support for *profiles* which allows you to conditionally load certain beans depending on the environment on which you are deploying.

The way this works is by including a "beans" element at the bottom of your configuration file and declaring a profile for which these beans are used:

```
  <beans profile="default">

    <!-- Embedded database to support unit tests -->
    <jdbc:embedded-database id="datasource" type="HSQL"></jdbc:embedded-
database>
    <util:map id="jpaPropertyMap">
      <entry key="hibernate.hbm2ddl.auto" value="create-drop" />
      <entry key="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect" />
    </util:map>
```

E.g. the above declares an embedded data source that is only available for the "default" profile, which is the profile that will be used when you do not specify anything further. In our case, this is the profile that is used for the integration tests.

In the web.xml file, you can specify a context parameter that will be used to set the profile that is used when actually running the live application:

```
  <context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>live</param-value>
  </context-param>
```

This means that the "live" profile will be used when the application is actually deployed, and the beans defined for the *default* profile will not be available.

## 7.4 System properties

Dynamo supports a couple of ways of dealing with (system) properties.

The easiest way of declaring a property is by including it in the *application.properties* file which is located in the **src/main/resources** directory of the *domain* subproject, and is automatically included in the deployable.

This file is a standard Java properties file and thus contains a number of key/value pairs:

```
default.date.format=dd-MM-yyyy
default.time.format=HH:mm:ss
default.datetime.format=dd-MM-yyyy HH:mm:ss
default.locale.active=true
```

We use the standard Spring **PropertyPlaceHolderConfigurer** in order to make this property file available to the rest of the application:

```
  <bean id="propertyPlaceholder"

  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigu
rer">
    <property name="systemPropertiesModeName"
value="SYSTEM_PROPERTIES_MODE_OVERRIDE" />
    <property name="location" value="classpath:application.properties" />
    <property name="ignoreResourceNotFound" value="false" />
  </bean>
```

You can then inject a system property into your code in the following way:

```
@Value("${import.correction.threads}")
private String numberOfThreads;
```

Or using the alternative notation:

```
@Value("${systemProperties['import.correction.threads']}")
private String numberOfThreads;
```

Note that the properties defined in *application.properties* can be overridden in several ways:

- By declaring an actual system property (using the *–D* command line argument, or by defining a system property in your application server configuration, e.g. in the *standalone.xml* file in JBoss EAP).
- By overriding an application property during the Maven build. Note that you should not do this for properties that are environment-dependent since it would make your build environment-specific which is bad form.

If you want to override a property using the Maven build process, do the following:

- Include the property in the *application.properties* file in the following way:

application.version=${project.version}

- If needed, include the property in the *properties* section in the root pom.xml file of the project. (Note that in this example we use a built-in Maven property so it is not needed to explicitly define it).
- During the Maven build, the placeholder in the *application.properties* file will be replaced by the property value from the pom.xml file.

# 8 Views, Menus, and authentication

A Vaadin application allows you to define one or more *Views* that roughly correspond to pages in a more traditional application. Dynamo provides support for using Spring bean injection into these views and also allows for declarative security and the automatic adaptation of the menu bar based on the user's rights.

```
@SpringView(name = Views.PROGRAMME_SETUP_VIEW)
@UIScope
@Authorized(roles = { Roles.VIEW_PROGRAM })
public class ProgrammeSetupView extends BaseProgrammeView {

  @Autowired
  private ProductService productService;
```

The above shows an example of a view definition. The view is annotated with **@SpringView** in order to allow the injection of Spring beans into the view. Note that the **@SpringView** annotation takes a single *name* parameter which must contains a unique name of the view.

The view is also annotated with @UIScope to ensure it has a UI-wide scope (a UI-scope in Vaadin corresponds to a single browser tab).

Once a view is annotated with @SpringView, you can use the @Autowired annotation to inject Spring-managed beans into the view. *Note that @Inject unfortunately does not seem to work here.*

The **@Authorized** annotation is used to restrict access to the view to users in certain roles (in this case the "VIEW_PROGRAM" role). If a user tries to access a view for which he/she is not authorized, he will be immediately be taken to an "Access denied" page.

Based on the above, the Accelerator Framework also supports the declarative construction of a menu bar. For this, the *MenuService* is used.

The *MenuService* is a Spring-managed bean which contains a *constructMenu* method that you can use in order to construct the menu bar. The menuService is configured using the *menu.properties* message bundle:

```
sas.menu.1.displayName=Set-Up
sas.menu.1.1.displayName=Set-Up
sas.menu.1.1.destination=ProgramSetupView
sas.menu.1.1.tabIndex=0
sas.menu.1.2.displayName=Sales Forecast
sas.menu.1.2.destination=ProgramSetupView
sas.menu.1.2.tabIndex=1

sas.menu.2.displayName=Warehouse Deliveries
sas.menu.2.1.displayName=Overview
sas.menu.2.1.destination=WarehouseDeliveryView
```

When calling the *constructMenu* service, you must pass along a prefix that will be used to retrieve the messages from the bundle. In the above case, the prefix is "sas.menu" which means that the menu will be constructed based on the messages starting with this prefix.

The *MenuService* will iterate over the messages, starting at "sas.menu.1", followed by "sas.menu.2", "sas.menu.3" etc., and try to recursively build a menu for each message. Child items can be specified by including additional numbers, e.g. "sas.menu.1.1" is the first child item of the first menu item. Note that the numbering must always start at 1 and must be continuous (you are not allowed to skip any numbers). Menu items can be nested arbitrarily deep, e.g. "sas.menu.1.1.1.1" would be valid).

For every menu item, you can specify the following:

- displayName: this is the name of the menu item (the text that will appear in the menu). It is required
- destination: this is the name of the view to which the application will navigate after the user select the corresponding menu item. It is an optional value. If you do not specify a destination, then the menu item will not be clickable, and only serve as a container for child menu items.
- tabIndex: this is the tab index (starting at zero) of the tab to display. It can be used in case the view to which the user navigates contains multiple tab sheet, in order to directly display the correct tab.

The menu service will use the above information to build a menu as follows:

- It will find the message `sas.menu.1.displayName` and use it to create a top-level menu item named "Set-Up".
- It will see that this menu item has children (sas.menu.1.1) and (sas.menu.1.2) and add two menu items named "Set-Up" and "Sales forecast" to the top-level menu item. Both items are clickable and will lead to the **ProgramSetupView**, but with a different tab selected.
- It will then return to the top level, and find the `sas.menu.2.displayName` message to construct a top-level menu item named "Warehouse Deliveries", which has a child item "Overview" that leads to the view named "WarehouseDeliveryView".

Any menu items will automatically be hidden/shown based on the **@Authorized** annotations placed on the views. So, if the framework detects that a menu item leading to the "ProgramSetupView" must be included, but the user does not have the "VIEW_PROGRAM" role, then the menu item will be hidden automatically. Any parent items that do not have any visible children will also be automatically hidden.

Note that you can include multiple separate menu structures in one *menu.properties* file simply by using different prefixes.

# 9 Composite UI components

Everything you have read up until now is an introduction that is needed to understand how the Dynamo Framework can easily be used to create fairly complex screens and user interface components. The next sections will cover some of the most common use cases.

## 9.1 Search screen (with edit options)

The framework offers the *SimpleSearchLayout* component for creating search screens. A simple application of this *SimpleSearchLayout* looks as follows:

```
    SimpleSearchLayout<Integer, Meeting> meetingSearchLayout = new 
SimpleSearchLayout<>(
        meetingService, getModelFactory().getModel(Meeting.class), 
QueryType.ID_BASED,
        new FormOptions(), new SortOrder("description", 
SortDirection.ASCENDING));
```

As you can see, you create a new instance of the SimpleSearchLayout and pass both the class of the primary key of the entity and the class of the entity itself as type parameters. You then provide the following:

- The Service that is used to retrieve data from the data layer
- The Entity Model that is used to construct the layout
- The type of the query: this can either be ID_BASED or PAGING as described in section 6.
- A *FormOptions* object. This will be explained below.
- The (default) sort order – this is optional
- The fetch joins to use – this is a vararg parameter and thus optional. It can be used to specify which relations to fetch when executing the search query. If left empty, then the default joins described earlier will be used.

And that is basically it – this is all the code you need in order to create a working search screen.

Note that the entity model you have specified before will be used to create the screen. I.e. all properties that are marked as *searchable* will be present in the search form, and all properties that have *showInTable* set to true will show up in the results table.

Of special note is the *selectMode* attribute that can be used to switch between displaying a combo box, a list box, and a lookup field for looking up a property of type MASTER.

By default, this search screen will also come with an "Add" button that will open a screen that when clicked brings up a screen that allows you to add a new entity, and a "Remove" button that can be used to remove an entry.

You can use the "FormOptions" to specify the behaviour of the screen. The FormOptions class is basically a *parameter object* that is used in order to prevent excessively long lists op parameters. It has a number of *showXXX* and *hideXXX* methods that can be used to toggle the availability of certain options.

E.g., by default any detail screen will open in edit mode, but if you don't want this behaviour you can set the "openInViewMode" parameter to true. After this is set, the detail screen will be read-only after being opened, and an edit button will be provided to switch the screen to edit mode.

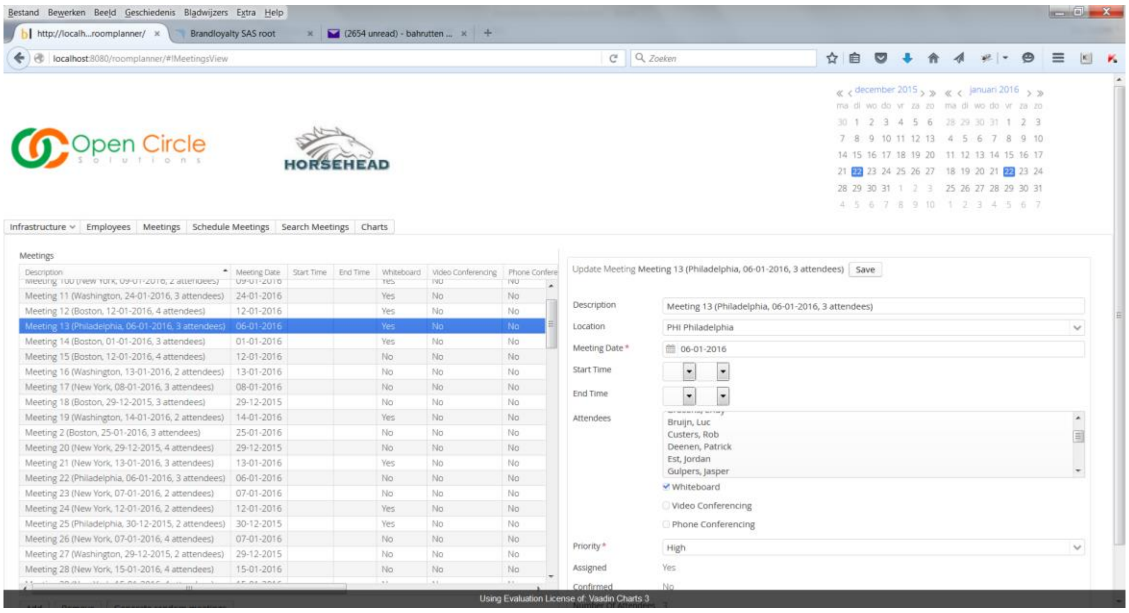
Note however that for more complex use cases, there are a lot of settings to play around with:

- **additionalFilters** can be used to set a list of filters that will be applied to every search query (even when the user leaves all search fields blank). Use this if you don't want to display all records but a certain subset.
- **fieldFilters** can be used to apply additional filters to search fields that allow the user to pick from a set (notably, for attributes of type MASTER or DETAIL). E.g. suppose that you have a search field for attribute "manager"that displays a list of people (instances of the Person class). If you want to display only managers whose name starts with a "b" then you can add a field filter that maps the "manager" attribute to the filter **Like("firstName", "b%")**;
- **detailJoins** is used to pass along the relations that must be joined when retrieving a single object when loading the details screen. Usually you can leave this empty and just rely on the defaults.
- **pageLength** can be used to set the number of rows that are displayed in the table.
- The **addSortOrder** method can be used to add additional sort orders (the first sort order can be passed to the constructor).
- **nrOfColumns** can be used to specify the desired number of columns in the search form. This default to "1".

## 9.2 Split layout

A split layout is a layout that contains both a results table and an edit form, displayed either next to each other or below each other (with the results table on top). You can use the *screenMode* parameter to set the desired orientation.

An example of a split layout in horizontal mode can be seen here:



Meetings

| Description | Meeting Date | Start Time | End Time | Whiteboard | Video Conferencing | Phone Confere |
|---|---|---|---|---|---|---|
| Meeting 11 (Washington, 24-01-2016, 3 attendees) | 24-01-2016 | | | Yes | No | No |
| Meeting 12 (Boston, 12-01-2016, 4 attendees) | 12-01-2016 | | | Yes | No | No |
| Meeting 13 (Philadelphia, 06-01-2016, 3 attendees) | 06-01-2016 | | | Yes | No | No |
| Meeting 14 (Boston, 01-01-2016, 3 attendees) | 01-01-2016 | | | Yes | No | No |
| Meeting 15 (Boston, 12-01-2016, 4 attendees) | 12-01-2016 | | | No | No | No |
| Meeting 16 (Washington, 13-01-2016, 2 attendees) | 13-01-2016 | | | No | No | No |
| Meeting 17 (New York, 08-01-2016, 3 attendees) | 08-01-2016 | | | No | No | No |
| Meeting 18 (Boston, 29-12-2015, 3 attendees) | 29-12-2015 | | | No | No | No |
| Meeting 19 (Washington, 14-01-2016, 2 attendees) | 14-01-2016 | | | Yes | No | No |
| Meeting 2 (Boston, 25-01-2016, 3 attendees) | 25-01-2016 | | | No | No | No |
| Meeting 20 (New York, 29-12-2015, 4 attendees) | 29-12-2015 | | | No | No | No |
| Meeting 21 (New York, 13-01-2016, 3 attendees) | 13-01-2016 | | | Yes | No | No |
| Meeting 22 (Philadelphia, 06-01-2016, 3 attendees) | 06-01-2016 | | | No | No | No |
| Meeting 23 (New York, 07-01-2016, 2 attendees) | 07-01-2016 | | | No | No | No |
| Meeting 24 (New York, 12-01-2016, 2 attendees) | 12-01-2016 | | | Yes | No | No |
| Meeting 25 (Philadelphia, 30-12-2015, 2 attendees) | 30-12-2015 | | | Yes | No | No |
| Meeting 26 (New York, 07-01-2016, 4 attendees) | 07-01-2016 | | | No | No | No |
| Meeting 27 (Washington, 29-12-2015, 2 attendees) | 29-12-2015 | | | No | No | No |
| Meeting 28 (New York, 15-01-2016, 4 attendees) | 15-01-2016 | | | No | No | No |

Update Meeting Meeting 13 (Philadelphia, 06-01-2016, 3 attendees)

| | |
|---|---|
| Description | Meeting 13 (Philadelphia, 06-01-2016, 3 attendees) |
| Location | PHI Philadelphia |
| Meeting Date * | 06-01-2016 |
| Start Time | |
| End Time | |
| Attendees | Bruijn, Luc; Custers, Rob; Deenen, Patrick; Est, Jordan; Gulpers, Jasper |
| | ✓ Whiteboard |
| | ☐ Video Conferencing |
| | ☐ Phone Conferencing |
| Priority * | High |
| Assigned | Yes |
| Confirmed | No |

As with the SimpleSearchLayout there are only a couple of lines of code needed in order to set up a (basic) SplitLayout.

Note that there are a couple of variants of the SplitLayout:

- ServiceBasedSplitLayout – which retrieves the data by making a service call.
  - ServiceBasedDetailLayout – a service based split layout that keeps track of a parent entity.
- FixedSplitLayout – a split layout to which you can pass a (fixed) collection of data to display (it doesn't use the service to retrieve the data to display)
  - FixedDetailLayout – a fixed split layout that keeps track of a parent object.

The *Detail* variants that keep track of a parent entity are useful when you are displaying a collection of that data that depends on a parent object being selected first – imagine for example that you have an application that first lets you pick an organisation and then allows you to browse a list employees of that organisation – in this case you can use a **ServiceBasedDetailLayout** that keeps track of the organisation as the parent object.

A simple example of a ServiceBasedSplitLayout looks as follows:

```
    ServiceBasedSplitLayout<Integer, Employee> employeeLayout = new
ServiceBasedSplitLayout<Integer, Employee>(
        employeeService, getModelFactory().getModel(Employee.class), new
FormOptions(),
        null, null) {

      @Override
      protected Filter createFilter() {
        return null;
      }
    };
```

You can specify the following parameters when creating it:

- The Service that is used to retrieve data from the data layer
- The Entity Model that is used to construct the layout
- A *FormOptions* object.
- A sort order (optional)
- A list of relations to fetch when executing the search query

Note that you can also implement the abstract *createFilter* method. This method must return the filter that will be used when executing the search that is used to fill the results table.

Like the *SimpleSearchLayout*, the *SplitLayout* supports the use of *fieldFilters*. You can also change attributes like *pageLength*.

The *SplitLayout* components also support the use of the FormOptions parameter object Of special interest here is the *screenMode* setting, which allows you to special whether to display the form in vertical mode (table above the edit form) or horiziontal mode (table to the left of the edit form). The default is *horizontal*.

## 9.3 SimpleEditLayout

The *SimpleEditLayout* component can be used to display or edit a single entity. It is useful if for some reason you already have an entity that you want to display, without having to first select it from a table or search screen.

Creating a *SimpleEditLayout* is very similar to the creating the other layout components and looks as follows:

```
    SimpleEditLayout<Integer, Organisation> layout = new 
SimpleEditLayout<Integer, Organisation>(
        organisation, organisationService, 
entityModelFactory.getModel(Organisation.class),
        new FormOptions());
```

You can specify the following parameters when creating it:

- The entity of which you want to display the details
- The Service that is used to retrieve data from the data layer
- The Entity Model that is used to construct the layout
- A *FormOptions* object.
- A sort order (optional)

## 9.4 TabularEditLayout

The *TabularEditLayout* component can be used to display a collection of data in a table. The constructor for the TabularEditLayout constructor looks as follows:

```
  public TabularEditLayout(BaseService<ID, T> service, EntityModel<T>
entityModel,
        FormOptions formOptions, SortOrder sortOrder, FetchJoinInformation...
joins) {
    super(service, entityModel, formOptions, sortOrder, joins);
  }
```

This results in a table in which items can directly be edited, added, and deleted. Optionally, you can use the *FormOptions* parameter object to make sure that the screen is initially displayed in read only mode, with an *Edit* button that can be used to toggle the edit mode.

Analogous to the *SplitLayout* there is also a *TabularEditDetailLayout* that can keep track of a parent entity.

## 9.5 HorizontalDisplayLayout

The *HorizontalDisplayLayout* is a very simple component that simply displays the attributes of an entity using a horizontal layout. Only properties of type BASIC are displayed.

## 9.6 LazyTabLayout

This is a component that is meant as an alternative to the standard Vaadin TabSheet, which immediately loads all tab sheets when first created. This component improves upon this by only loading a tab when it is actually opened. Until then, it simply constructs an empty dummy tab.

To use this component, subclass it an implement the *getTabCaptions* and *initTab* methods. The first method must return an array of Strings that contain the captions to use for the various tabs – these will also serve as keys to determine which tabs have already been loaded, so make sure they are all different.

The *initTab* method takes a single argument, namely the index of the tab that must be created. It must instantiate and return the component that is to be used as the content for that tab.

## 9.7 ProgressForm and UploadForm

*ProgressForm* is a form that can be used for executing (synchronous) batch processes. It comes with a built-in progress bar that can be used to give the user feedback on the status of the batch process.

Note that you can set the mode of the form: it can either be SIMPLE (no progress bar is used) or PROGESSBAR (a progess bar is used to keep track of the progression)

It its simplest form, the *ProgessForm* is simply a class that offers some based method to construct a form:

- The *doBuildLayout* method must be used to build the form. It can contain UI components that can be used to modify the parameters of the batch process, or one or more buttons to actually start the process. Note that by default the *ProgressForm* does *not* contain a button to start the job.
- The *estimateSize* method can be used estimate the size of the data that is being processed. You can use any means necessary in order to arrive at an estimate. This method does not do anything in case the mode of the progress form is set to SIMPLE.
- The *startWork* method must be called in order to actually start the batch process. Typically, this is the method that you will call from e.g. a Button's ClickListener.
- The *process* method is the method that contains the actual logic that must be called in order to carry out the batch process. Typically, you will delegate to a service method or something similar.
- If you need to carry out any validations before starting the process, overwrite the *isFormValid* method.

Under the covers, the progress form will use the prepare method to spawn two threads: one thread in which the actual processing is carried out and one thread that will periodically poll for the progression of the job and update the progress bar. Once the process starts, the component will automatically switch to the process bar view, and once it is done it will switch back to the form view. You can use the *afterWorkComplete* method to carry out an action after the job completes.

By default, the ProgressForm uses an AtomicInteger to keep track of the progress – if you want to use this (admittedly quite limited) mechanic, you can pass it along to the code that performs your batch processing and increment it whenever a step of your processing is done. The polling thread will use this to determine the completion rate of the job. If you want to update the progress in another way, you can overwrite the *estimateCurrentProgress* method in your subclass.

*UploadForm* is a subclass of *ProgressForm* that can be used to easily create a file upload form – it already contains a file upload component that will kick off the batch process once the upload button is clicked.

## 9.8 DetailsEditTable

The Dynamo framework contains built-in support for editing many-to-many relations (to already existing entities) in edit forms (a ListSelect component will be rendered by default) but the situation in which you have an object that contains a one-to-many relation in which the detail elements are an integral part of the entity being edited. A classic example of this would be the creation of a new order and the order lines that make up the order: the order lines do not exist on their own but are a vital part of the order.

In order to be able to create both an entity and its related sub-entities, you can use a *DetailsEditTable*. This is a fairly complex component which (at least for the moment) you have to initialize explicitly. You can do so by overriding the *constructCustomComponent* (see the section on callbacks) and have this return an appropriate instance of the *DetailsEditTable*.

This looks as follows:

```
List<ProductAlias> pas = new ArrayList<>(getSelectedItem().getAliases());
    Collections.sort(pas, new RowNumberComparator());

    FormOptions fo = new FormOptions();
    fo.setShowRemoveButton(true);
    fo.setHideAddButton(false);


return new DetailsEditTable<Integer, ProductAlias>(pas,
getEntityModelFactory().getModel(
        ProductAlias.class), viewMode, fo) {

      @Override
      protected ProductAlias createEntity() {
        // init logic
      }

      @Override
      public void fillDetails() {
        List<ProductAlias> pas = new
ArrayList<>(ProductDetailLayout.this.getSelectedItem()
            .getAliases());
        Collections.sort(pas, new RowNumberComparator());
        setItems(pas);
      }

      @Override
      protected void removeEntity(ProductAlias toRemove) {
          // remove logic
      }
    };
```

1. The DetailsEditTable, like many composite components, takes two type parameters: the type if the primary key of the entity and the type of the entity itself.
2. The constructor takes a collection of items to display (for a new entity this will obviously be empty; for an existing entity, you can take it from the entity that is being edited. Note that the DetailsEditTable only works on in-memory data, it cannot be used to query a repository. The idea is that you have already fetched the main entity you are editing, and all its related detail entities anyway.
3. The constructor also takes a FormOptions parameter object which you can set to e.g. specify if a Remove button must be shown. By default, an Add button is shown but the Remove button is hidden.

4. Finally, the constructor takes a Boolean *viewMode* parameter. You can simply pass along the parameters from the *constructCustomField* method you are overriding.
5. The *createEntity* method can be used to initialize a new entity. It is called after the user clicks the *Add* button.
6. Likewise, the *removeEntity* method can be used to remove the detail object from the parent entity.
7. The fillDetails method will be called whenever the user selects a different entity in the layout component that this DetailsEditTable is part of. It must be used to re-initialize the DetailsEditTable table using the detail entities of the newly selected entity. You must the *setItems* method in order to fill the table with the correct entities.

The *DetailsEditTable* is one of our earlier experiments and we are aware of the fact that it currently doesn't work optimally. This will be improved upon in a future version.

## 9.9 Callbacks

Most of the composite UI components support callback methods that allows you to influence the way they are working.

### 9.9.1 constructCustomField

One of the most common callback methods is the *constructCustomField* method which allows you to override the normal field generation process and create a field of your own. It is of course recommended that you use the Entity Model whenever possible, but the option to override it is there in case you need it. The override method looks as follows:

```
@Override
protected Field<?> constructCustomField(EntityModel<Meeting> entityModel,
           AttributeModel attributeModel, boolean viewMode, boolean
searchMode) {
  // your implementation here
}
```

The parameters are as follows:

- **entityModel** – the entity model on which the component is based
- **attributeModel** – the attribute model for the property for which you want to generate a custom field
- **viewMode** – whether the screen is in view mode
- **searchMode** –whether the screen is in search mode (i.e. the custom field you are generating will appear in the search form.

What you should typically do is inspect the attribute model and check if it is the model for the field whose generation you want to override. If this is the case, construct the custom field and have the method return it. Otherwise, simply return null, in which case the framework will default to the normal field generation process.

### 9.9.2 postProcessButtonBar

The *postProcessButtonBar* method can be used to make modifications to the button bar that contains the buttons like the Save button, the Add button etc.

You can overwrite this method and add code that adds one or more new buttons to the screen. The buttons are always added to the right of the existing buttons (in the order in which they are added).

If the button should become enabled/disabled based on whether or not a row is selected in the table, then you can call the *registerDetailButton* method (passing the button as the argument) inside the *postProcessButtonBar* method.

### 9.9.3 postProcessEditFields

If you want to add custom behaviour to the edit form, you can do so in the *postProcessEditFields* method. This method that e.g. be used to add extra behaviour to some fields (i.e. respond to the change of field A by changing the value in field B, or by disabling field C).

Note: you should *not* use this method in order to modify the characteristics of the fields that you can easily manipulate using the entity model. E.g. if you always want a field to be invisible or read-only, use the entity model for this. Use the *postProcessEditFields* method only for complex inter-field behaviour.

### 9.9.4 postProcessLayout

The postProcessLayout method can be used to add additional functionality to any layout. This method is called after the full layout has been constructed.

### 9.9.5 mustEnableButton

This callback can be used to specify extra conditions under which a certain button must be enabled or disabled. It is called after an item is selected in the table.

### 9.9.6 isEditAllowed

Override this method if you need to specify additional conditions under which editing is allowed. If this method returns false, then the screen will be displayed in

### 9.9.7 afterDetailSelected

This method is called after the user has selected a detail entity in the table in a collection-based layout (SimpleSearchLayout or any SplitLayout). The currently selected item and the edit form are passed as parameters. You can use this method to e.g. enable or disable certain extra fields.

Note: for a *SimpleSearchLayout* this is only called after navigating to the detail screen. For the *SplitLayout* this is called directly after selecting a row in the table.

### 9.9.8 afterModeChanged

This method is called after the mode of the screen changes from read only to editable or vice versa.

### 9.9.9 createEntity

This method is called after pressing any "Add" button and servers to set up a fresh instance of an entity. By default is will just create an entity by calling the required no-argument constructor. Note that this entity will be pre-filled with any default attribute values that are defined in the corresponding Entity Model.

If, after this, you need to perform any additional initialization, you can use the *createEntity* method to perform this initialization.