

ModuleStudio User's Guide v0.5.2

Axel Guckelsberger and many others

September 2, 2011

Contents

I. First steps	6
1. Introduction	7
1.1. About ModuleStudio	7
1.2. Benefits	7
1.3. About this manual	8
1.4. Model layers	8
1.4.1. Application layer	8
1.4.2. Data layer	8
1.4.3. Controller layer	9
1.4.4. View layer	9
1.4.5. Workflow layer	9
1.5. Additional notes	9
2. Component overview	10
2.1. Introduction	10
2.2. Modeling Language	10
2.2.1. Meta model	10
2.2.2. Constraints	10
2.3. Modeling Editors	10
2.3.1. Graphical	10
2.3.2. Textual	11
2.3.3. Structural	11
2.3.4. Hybrid	11
2.4. Generators	11
2.4.1. zClassic	11
2.4.2. Reporting	11
2.4.3. zOO	11
2.5. Additional notes	11
3. Getting started	12
3.1. Installation	12
3.2. First tour	12
3.3. Development process	12
3.4. Additional notes	12

4. User interface	13
4.1. Introduction	13
4.2. Basic usage	13
4.3. Single editors	13
4.3.1. Main editor	13
4.3.2. Model editor	13
4.3.3. Controller editor	13
4.3.4. View editor	13
4.3.5. Workflow editor	13
4.4. Useful hints	15
4.4.1. Keyboard shortcuts	15
4.5. Additional notes	16
II. Enhanced topics	17
5. Validation	18
5.1. Introduction	18
5.2. Basic usage	18
5.2.1. Triggering validation	18
5.2.2. Validation consequences	19
5.3. Application layer	19
5.3.1. Global rules	19
5.3.2. Application	21
5.3.3. Model container	22
5.3.4. Controller container	22
5.3.5. View container	23
5.4. Model layer	23
5.4.1. Model container	23
5.4.2. Entity	24
5.4.3. Entity field	28
5.4.4. Relationship	37
5.4.5. Entity index	43
5.4.6. Variables	44
5.4.7. Other elements	44
5.5. Controller layer	45
5.5.1. Controller container	45
5.5.2. Controller	46
5.5.3. Action	47
5.5.4. Action handler	47
5.5.5. Action event	47
5.5.6. Transition	48
5.6. View layer	48
5.6.1. View container	48

5.6.2.	View	48
5.6.3.	Root panel	49
5.6.4.	Other UI elements	49
5.6.5.	Layout order	49
5.7.	Workflow layer	49
5.8.	Additional notes	50
6.	Application generator	51
6.1.	Introduction	51
6.2.	Basic idea	51
6.3.	How it works	52
6.3.1.	Input dialogs	52
6.4.	The workflow	52
6.5.	Generator reference	53
6.5.1.	Application layer	53
6.5.2.	Model layer	58
6.5.3.	Controller layer	79
6.5.4.	View layer	82
6.5.5.	Workflow layer	86
6.6.	Additional notes	86
7.	Customisation and maintenance	87
7.1.	Introduction	87
7.2.	Long-term maintenance	87
7.2.1.	Keep consistent	87
7.2.2.	Document your changes	87
7.2.3.	Use overriding	87
7.2.4.	Code additions	87
7.2.5.	Use versioning	88
7.3.	Additional notes	88
8.	Other cartridges	89
8.1.	Introduction	89
8.2.	Reporting	89
8.2.1.	Documentation	89
8.2.2.	Model information	89
8.2.3.	Function points	89
8.3.	zOO	89
8.4.	Additional notes	90
9.	AddOns	91
9.1.	Introduction	91
9.1.1.	Language packs	91
9.1.2.	Figure galleries	91

9.1.3. Template sets	91
9.1.4. Generator cartridges	91
9.1.5. Other extensions	91
9.2. Extension Points	91
9.3. Additional notes	91

III. Appendix **92**

10. Glossary **93**

Part I.

First steps

1. Introduction

1.1. About ModuleStudio

ModuleStudio is a development environment with which one can quickly, simply and efficiently describe and generate web applications. Software developers can create complex Zikula extensions in a few steps and meet individual project requirements with them.

ModuleStudio allows you to precisely describe applications for the Zikula Application Framework on an abstract level. The boring writing of schematic code is history, the development is focused on an application's functional differences compared to other Zikula extensions.

ModuleStudio takes Zikula development to a new level as it displaces the typing of source code by a *convenient modeling* process. All applications generated by ModuleStudio are automatically API-compliant and match the actual conventions relating security and usability. When new Zikula core versions occur all changes are updated so all modules will be ready for a new release after simply pressing a button. ModuleStudio therefore reduces architectural motivated adjustments and increases maintainability as well as reusability of Zikula's common artefacts.

See also: [What is ModuleStudio](#)

1.2. Benefits

- *Development time/costs*: avoid wasting weeks for schematical and architectural motivated code parts!
- *Maintainability*: your software is a model - easily changeable and cheaply maintainable! No more efforts for getting your modules up to date for new versions!
- *Code quality*: take profit from best practices and established patterns!
- *Architectural compliance*: take most usage from powerful core frameworks and interfaces! No more unsecure and legacy extensions!
- *Reusability*: share and modify your models! Do not make the same work twice!
- *Understandibility*: avoid having to learn programming rules and framework details! Develop with general terms independant from technical stuff!

More information can be found in these articles:

- [Advantages of ModuleStudio](#)

- How MDSO reduces costs for long-term maintenance of comprehensive software system families
- From scaffolding and UML to MDSO and DSL

1.3. About this manual

This user manual is going to provide all required information to work with ModuleStudio. Furthermore it serves as a reference for all details of the generator. This document should be available in several formats:

- help within ModuleStudio
- online help on our website
- pdf print version

In future versions we are going to add interaction capabilities to incorporate active help into ModuleStudio itself while you are working. For now the focus was to get the manual actually written ;-)

1.4. Model layers

A ModuleStudio model is divided into several submodels in order to separate concerns. This section shows the different layers and their role for the overall application. It is just a brief overview which does not explain single fields or settings in detail. This will be done in later chapters instead.

1.4.1. Application layer

The application layer is the overall container where all sublayers are linked together. In addition there are a bunch of application-wide properties with which general aspects are defined, like for example the name and the author of an application.

1.4.2. Data layer

Each data layer defines a logical group of data structures for the application. This involves the following key concepts:

- Entities
- Entity fields
- Join and inheritance relationships
- Behavioural and functional extensions
- Variables

1.4.3. Controller layer

Each controller layer defines a logical group of use cases for the application. This involves the following key concepts:

- Controllers
- Controller actions
- Action handlers
- Action events
- Transitions

1.4.4. View layer

Each controller layer defines a logical group of view templates for the application. This involves the following key concepts:

- Views
- Root panels, composite panels, sub panels
- Forms and tables
- Activators
- Layout orders

1.4.5. Workflow layer

The workflow layer is not implemented yet. This section is just a dummy for future.

1.5. Additional notes

None yet.

2. Component overview

2.1. Introduction

This page gives you an overview of the main parts of ModuleStudio and how they work together.

2.2. Modeling Language

The inner core of MOST is a domain-specific language (DSL) for Zikula extensions. This language allows a formalised description of applications which is a fundamental requirement to process models automatically with the help of transformations.

For convenience ModuleStudio uses general terms for modeling MVC applications. As you will see later an application model has different submodels for describing corresponding architectural layers (model, controller, view).

2.2.1. Meta model

The meta model defines the essential concepts of the ModuleStudio language, that is which model elements may exist and how they are allowed to work with each other. This allows reusing the basic domain concepts at several places, like validation, editors, generators and so on.

2.2.2. Constraints

In addition to the meta model there are many validation rules (§5) to enrich the modeling language with more precise knowledge. These constraints ensure that the generator can only be started for valid models.

2.3. Modeling Editors

The user interface consists of different types of editors which may include event different kinds of how information is described.

2.3.1. Graphical

Graphical notations are convenient for modeling edges between different nodes. They are not that well suited for creating huge lists of similar elements for instance.

ModuleStudio offers graphical editors for creating and changing models for describing different applications. See the user interface chapter (§4) for more information.

2.3.2. Textual

A textual syntax is very nice for rapid creation of structures. It becomes less handy for relationships.

At the moment there is no textual editor included in ModuleStudio. This is going to change soon though.

2.3.3. Structural

Structural views, for example trees, are another possible viewpoint for describing a model.

At the moment there is no structural editor included in ModuleStudio. This may change in future though.

2.3.4. Hybrid

In future of ModuleStudio is heading towards some kind of hybrid modeling where you can combine textual and graphical editors for describing applications.

2.4. Generators

The generators add technical details depending on the target system. Their task is creating source code or other artifacts from application models.

2.4.1. zClassic

The primary generator cartridge is *zclassic* which creates a Zikula extension. You can read more about this in the generator chapter (§6). Also important is another chapter about customisation and maintenance (§7) of generated applications.

2.4.2. Reporting

The *reporting* cartridge creates some documents from a given model. Please continue here for details (§8.2).

2.4.3. zOO

The *zOO* cartridge aims on creating code for a future equally-named framework. Please continue here for details (§8.3).

2.5. Additional notes

None yet.

3. Getting started

This chapter explains the first steps required for starting creating applications with ModuleStudio.

For now we refer only to existing tutorials as they describe things still quite well.

3.1. Installation

Please see this tutorial.

3.2. First tour

Please see this tutorial.

3.3. Development process

Developing model-driven applications works well in combination with an iterative-incremental development process. In this approach you start with a small model which will then be enhanced in several steps whereby some short tests verify that the direction is correct.

Each cycle consists of the following steps:

1. Create or change model
2. Regenerate
3. Merge changes
4. Test intermediate results

3.4. Additional notes

None yet.

4. User interface

4.1. Introduction

This section shows how to use ModuleStudio. Starting with a general demonstration of the user interface it goes step by step through all single editors required for creating a complete model.

At the moment this page consists only of a few links to corresponding tutorials. It will be enhanced at a later stage after the actual modeling language has matured enough to spend work on completing the user interface.

4.2. Basic usage

Please see this tutorial.

4.3. Single editors

4.3.1. Main editor

Please see this tutorial.

4.3.2. Model editor

Please see this tutorial.

4.3.3. Controller editor

Please see this tutorial.

4.3.4. View editor

The view editor has not been implemented yet. This section is therefore just a dummy for future.

4.3.5. Workflow editor

The workflow editor has not been implemented yet. This section is therefore just a dummy for future.

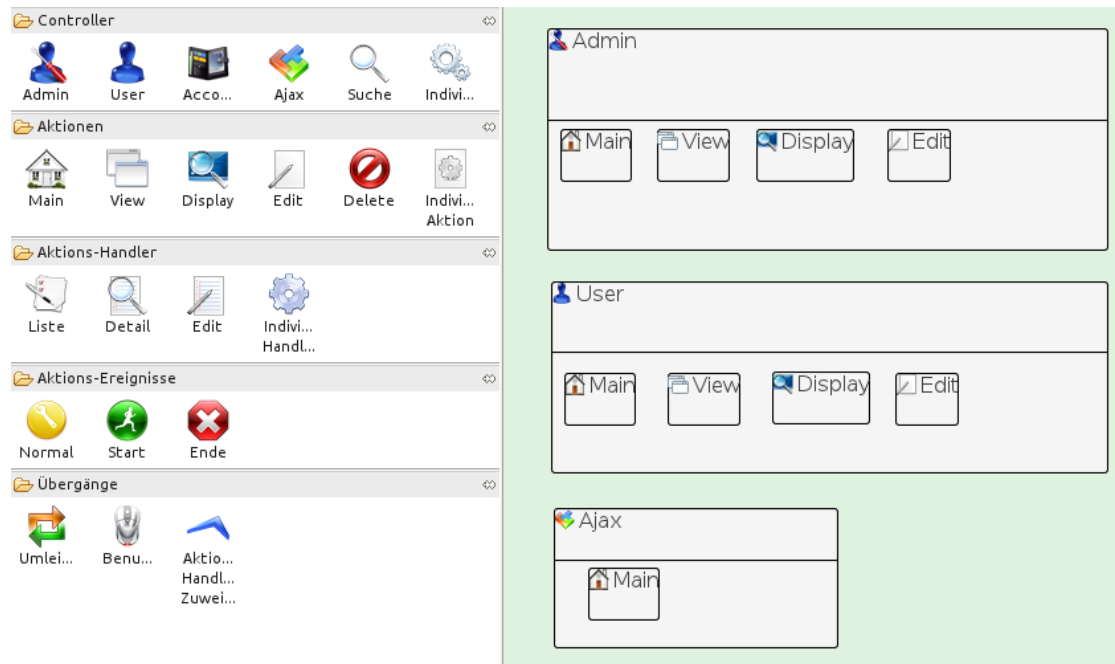


Figure 4.3.: Controller model

4.4. Useful hints

Here are some tutorial showing special abilities for certain use cases:

- Customise palette
- Multiple container elements
- Moving fields with drag n drop
- Creating multiple elements quickly
- Working with multiple windows

4.4.1. Keyboard shortcuts

There are some very handy shortcuts hidden in ModuleStudio. For example it can be worth to experiment with the Ctrl (Alt on Mac) and/or Shift keys when moving or resizing an object.

Here is a list of all editor shortcuts.

If one has selected a palette tool and creates an object in the diagram it is usually required to select the tool again in order to create another object. If one presses the Ctrl (Alt on Mac) key one can create multiple elements of the same type in one step.

4.5. Additional notes

None yet.

Part II.

Enhanced topics

5. Validation

This chapter explains the intention behind validation in ModuleStudio. The biggest part is a reference section listing all validation rules in detail and explains the motivation behind them. You can use this reference to search for certain error messages if you want to know more about the background. If you are unsure about the terminology used in a description please refer to the generator chapter (§6) in this documentation.

5.1. Introduction

Validation is an essential part of a modeling language. By checking the current model against several *constraints* it ensures that certain scenarios can not happen in order to avoid problems based on invalid states of model elements.

With the help of validation in ModuleStudio all subsequent components can process the given application model without having to revalidate common concerns.

Having clean and properly validated models is also very helpful for Zikula as a framework because in the long term this leads to a constantly high quality across third party extensions which is traditionally a weak point for every open source system. See this tutorial for more information about this aspect.

The constraints described here are evaluated within the graphical editor as well as during headless generator workflows.

5.2. Basic usage

This section gives a brief overview of how validation can be executed and what can be done if problems occur.

5.2.1. Triggering validation

In common use cases it is not required to do anything manually. ModuleStudio does automatically perform a *live validation* so you get immediate feedback after having done some model amendments.

For more complex model this can be come inconvenient with regards to performance as the live validation has to check many rules every few seconds. In these cases one can simply deactivate the live validation by stopping the according jobs in the process view (see this tutorial).

5.2.2. Validation consequences

As the generator requires a model without validation problems it can only be started in the main menu after your model has no errors left.

If the menu entry for starting the generator is inactive even if your models seems to be clean, please save the model in the main editor and click once in the diagram canvas to let the validation update it's state accordingly.

In case live validation has been deactivated you can start a *manual validation* in the main menu, too.

5.3. Application layer

5.3.1. Global rules

Context	Check	Remarks
Every element with a name	The name must be a valid identifier (e.g. no whitespace or special characters).	Camel case is recommended to get more readable messages generated, for example <i>mySpecialUser</i>

5.3.2. Application

Context	Check	Remarks
Application	The application must have a name.	Application name must have a length of at least five chars. However recommended are at least seven chars.
Application	Application name must start with a capital letter.	Since Zikula 1.3 extensions must start with a capital letter. Of course we could simply generate it this way. But for consistency we decided to follow this rule more strictly by enforcing it in the model already.
Application	The application must have an author.	
Application with email set	The value for the application email field must be a valid email address.	
Application with url set	The value for the application url field must be a valid url.	Allowed protocols are http, ftp and https.
Application	The application must have a prefix for it's database tables.	This prefix is required to prevent naming collisions between several modules. Otherwise it would be a problem if multiple extensions use common table names like <i>user</i> or <i>category</i> . The prefix must have a length of more than two chars, whereby a at least four is recommended.
Application	The prefix must be a valid identifier (e.g. no whitespace or special characters).	Essentially the same as the global rule for names above. You should use lowercase here, but it will be generated in lowercase in all cases.
Application	The application must have a version. The application version must conform to the pattern x.y.z.	Another requirement introduced in Zikula 1.3. Valid values are 1.0.0, 1.2.2, but not 1.1 or 2.1.0beta.
Application	The application must contain at least one model container.	At the moment ModuleStudio wants a model with at least one entity. If you are modeling an extension without any data storage, just create some dummy elements.
Application with model containers	Names of model containers must be unique.	Unique identification and separation of data sources.
Application with model containers	There should be at least one model container referring to the system's internal data source. There must be only one model container referring to the system's internal data source.	Each model container has a property marking it as default data source or not. Only one container can set this to true which will then be handled as normal by Zikula. Additional model containers

5.3.3. Model container

Context	Check	Remarks
Model container	The model container must have a name.	The container name must have a length of at least three chars. Recommended are at least four chars.
Model container	Model container must be assigned to an application.	Should not occur in practice, this is just for completeness.
Application with model container with entities	Exactly one entity must be declared as leading (leading=true).	More information in the Entity section (§5.4.2).

5.3.4. Controller container

Context	Check	Remarks
Controller container	The controller container must have a name.	The container name must have a length of at least three chars. Recommended are at least four chars.
Controller container	Controller container must be assigned to an application.	Should not occur in practice, this is just for completeness.
Controller container	Every controller container must rely on at least one model container.	You have to create relationships between controller and model containers. This specifies which controller functions may process data from which sources.
Controller container	Every controller container must reference at least one view container.	You have to create relationships between controller and view containers. This specifies which controller functions may use which view templates for their output. This rule is currently not active as the view editor is not ready yet.
Application with controller container with controllers	There must not exist more than one (admin user account ajax search) controller.	Predefined controllers are unique per application (i.e. there is only one admin area).
Application with controller container with controllers	Names of custom controllers must be unique.	For example there may not be two controllers which are both named <i>edit</i> .

5.3.5. View container

Context	Check	Remarks
View container	The view container must have a name.	The view name must have a length of at least three chars. Recommended are at least four chars.
View container	View container must be assigned to an application.	Should not occur in practice, this is just for completeness.

5.4. Model layer

5.4.1. Model container

Context	Check	Remarks
Model container	The model container must contain at least one entity.	
Model container with entities	Entity names must be unique.	For example there may not be two entities which are both named <i>person</i> .
Model container with entities	The amount of entities is getting quite high. Maybe it makes sense to split up the model into two single applications.	This warning appears if you have more than 14 entities. Remember Zikula is a modular system. You can design whole families of extensions with ModuleStudio, so please try keeping complexity low and apply separation of concerns.
Application with model container with entities	An entity must not have the same name as the application.	This case is reserved as it could make sense to use corresponding namespaces in generation for encapsulating some common code parts.

5.4.2. Entity

General entity settings

Context	Check	Remarks
Entity	The entity must be assigned to a model container.	Should not occur in practice, this is just for completeness.
Entity	Every entity must have a (name name for multiple instances).	Entity (multiple) name must have a length of at least two chars. Should have at least four chars.
Entity	Entity (multiple) name must not contain underscores.	Underscores are not allowed as they are used for class autoloading.
Entity	Every table must contain at least one field or inherit fields from a parent table.	Ensures that either there are some fields in this entity or an outgoing inheritance relationship.
Entity with fields	You may not mark a field as primary, this is done automatically - unless maybe you want to create composite primary keys.	This warning appears because ModuleStudio adds primary keys automatically and uses the Doctrine 2 default settings if nothing else is explicitly specified in the model. Beside special use cases like custom join conditions or composite primary keys you won't need to set primary keys manually.
Entity with fields	Remove ID fields... you do not need them ;-) unless maybe you want to create composite primary keys.	This warning appears if a field is named like <i>id</i> or <i>personid</i> or <i>person_id</i> for an entity named <i>person</i> . In these cases ModuleStudio adds primary keys to this table automatically before the generation.
Entity with composite primary keys	Composite entities can only have identifier strategies NONE and ASSIGNED.	Automatic identifier generation is not possible for composite primary keys.
Entity with fields	Every entity must contain one field defined as leading (leading = true).	The leading field is used in list views, auto completion as well as default sorting criteria (unless the entity is using the Sortable extension).

Inheritance-related entity settings

Context	Check	Remarks
Entity with (inheritance) relationships	All entities within a class hierarchy must have the same change tracking policy.	Requirement by Doctrine 2.
Entity with (inheritance) relationships	All entities within a class hierarchy must not have a field with the same name.	
Entity	All inheritance connections within a class hierarchy must have the same inheritance strategy.	
Entity	All inheritance connections within a class hierarchy must have the same discriminator column.	
Entity	An entity can not inherit from multiple entities.	
Entity defined as mapped super class	All associations outgoing from mapped super classes must be unidirectional.	
Entity defined as mapped super class	One-to-many relations are not possible for mapped super classes.	
Entity defined as mapped super class	Many-to-many relations are only possible for mapped super classes if it is used only in one entity at the same time.	
Entity	The length of all entity fields must not be higher than 21845.	The limit is 65535 bytes, while UTF-8 requires three bytes for each char.

Extension-related entity settings

Context	Check	Remarks
Entity defined as loggable	Loggable entities need one field with the version attribute set to true.	Can be either integer or date-time fields.
Entity defined as geographical	Entities with geographical behaviour should ideally contain a String Field with name zipcode with a length of at least 10.	Just a warning to support best practices.
Entity defined as loggable	There must not exist an entity named <code>FooLogEntry</code> as this is reserved by the corresponding extension.	For an entity named <i>person</i> ModuleStudio does not only generate corresponding <i>Person</i> classes, but also an entity named <i>PersonLogEntry</i> for managing the version log entry entities.
Entity with translatable field	There must not exist an entity named <code>FooTranslation</code> as this is reserved by the corresponding extension.	For an entity named <i>person</i> ModuleStudio does not only generate corresponding <i>Person</i> classes, but also an entity named <i>PersonTranslatable</i> for managing translation entities.
Entity defined with closure tree	There must not exist an entity named <code>FooClosure</code> as this is reserved by the corresponding extension.	For an entity named <i>person</i> ModuleStudio does not only generate corresponding <i>Person</i> classes, but also an entity named <i>PersonClosure</i> for managing the closure entities.
Entity defined as attributable	There must not exist an entity named <code>FooAttribute</code> as this is reserved by the corresponding extension.	For an entity named <i>person</i> ModuleStudio does not only generate corresponding <i>Person</i> classes, but also an entity named <i>PersonAttribute</i> for managing the attribute entities.
Entity defined as categorisable	There must not exist an entity named <code>FooCategory</code> as this is reserved by the corresponding extension.	For an entity named <i>person</i> ModuleStudio does not only generate corresponding <i>Person</i> classes, but also an entity named <i>PersonCategory</i> for managing the category entities.
Entity with meta data	There must not exist an entity named <code>FooMetaData</code> as this is reserved by the corresponding extension.	For an entity named <i>person</i> ModuleStudio does not only generate corresponding <i>Person</i> classes, but also an entity named <i>PersonMetaData</i> for managing the meta data entities.

5.4.3. Entity field

General field settings

Context	Check	Remarks
Entity field	Field must be assigned to an entity.	Should not occur in practice, this is just for completeness.
Entity field	Every field must have a name.	Field name must have a length of at least two chars. Should have more than three chars.
Entity field	Field names must be unique.	
Entity field	Field name is a reserved identifier (module, type, func, lang, theme).	These are reserved vars in traditional Zikula extensions.
Entity field	Field name is a reserved identifier (<code>_c</code> , <code>_a</code> , <code>_l</code>).	These are reserved vars in the zOO framework.
Entity field	Field name is a reserved database keyword.	ModuleStudio prevents the usage of keywords which are reserved in some database systems. Background is that there are no column prefixes anymore. For a list of all keywords see the following section (§5.4.3).
Entity field	Mandatory fields may not be nullable, too.	Occurs if you try to activate both <i>mandatory</i> and <i>nullable</i> properties for a field.

Reserved database keywords

The following list has been merged and includes therefore all keywords of all supported DBMS.

- abort, access, action, activate, add, after, alias, all, allocate, allow, alter, analyse, analyze, and, any, arraylen, as, asc, asensitive, associate, asutime, at, attach, attributes, audit, authorization, autoincrement, aux, auxiliary, avg
- backup, before, begin, between, bigint, binary, blob, both, break, browse, buffer-pool, bulk, by

- cache, call, called, capture, cardinality, cascade, cascaded, case, cast, ccsid, change, char, character, check, checkpoint, clone, close, cluster, clustered, coalesce, collate, collection, collid, column, comment, commit, committed, compress, compute, concat, condition, conflict, connect, connection, confirm, constraint, contains, containstable, continue, controlrow, convert, count, count_big, create, cross, current, current_date, current_lc_ctype, current_path, current_schema, current_server, current_time, current_timestamp, current_timezone, current_user, cursor, cycle
- data, database, databases, datapartitionname, datapartitionnum, date, day, days, day_hour, day_microsecond, day_minute, day_second, db2general, db2genrl, db2sql, dbcc, dbinfo, dbpartitionname, dbpartitionnum, deallocate, dec, decimal, declare, default, defaults, deferrable, deferred, definition, delayed, delete, dense_rank, denserank, deny, desc, describe, descriptor, detach, deterministic, diagnostics, disable, disallow, disconnect, disk, distinct, distinctrow, div, distributed, do, document, double, drop, dssize, dummy, dual, dynamic
- each, editproc, else, elseif, enable, encoding, encryption, end, enclosed, end-exec, ending, erase, errlvl, errexit, escape, escaped, every, except, exception, excluding, exclusive, exec, execute, exists, exit, explain, external, extract
- fail, false, fenced, fetch, fieldproc, file, final, fillfactor, float, float4, float8, floppy, for, force, foreign, free, freetext, freetexttable, freeze, from, full, fulltext, function
- general, generated, get, glob, global, go, goto, grant, graphic, group
- having, handler, hash, hashed_value, having, high_priority, hint, hold, holdlock, hour, hours, hour_microsecond, hour_minute, hour_second
- identified, identity, identitycol, identity_insert, if, ignore, ilike, immediate, in, including, inclusive, increment, index, indexed, indicator, inf, infile, infinity, inherit, initial, initially, inner, inout, insensitive, insert, instead, int, int1, int2, int3, int4, int8, integer, integrity, intersect, interval, into, is, isnull, isobid, isolation, iterate
- jar, java, join
- keep, key, keys, kill
- label, language, lateral, lc_ctype, leading, leave, left, level, like, limit, lineno, lines, linktype, load, local, localdate, locale, localtime, localtimestamp, locator, locators, lock, lockmax, locksize, long, longblob, longtext, loop, low_priority
- maintained, match, materialized, max, maxextents, maxvalue, mediumblob, mediumint, mediumtext, microsecond, microseconds, middleint, min, minus, minute, minutes, minute_microsecond, minute_second, minvalue, mirrorexit, mod, mode, modifies, modify, month, months
- nan, national, natural, new, new_table, nextval, no, noaudit, nocache, nocheck, nocompress, nocycle, nodename, nodenumber, nomaxvalue, nominvalue, nonclustered, none, noorder, normalized, not, notfound, notnull, nowait, no_write_to_binlog, null, nullif, nulls, number, numeric, numparts

- obid, of, off, offline, offset, offsets, old, old_table, on, once, online, only, open, open-datasource, openquery, openrowset, optimization, optimize, option, optionally, or, order, out, outer, outfile, over, overlaps, overriding
- package, padded, pagesize, parameter, part, partition, partitioned, partitioning, partitions, password, path, pctfree, percent, perm, permanent, piecesize, placing, pipe, plan, pragma, position, precision, prepare, prevval, primary, print, prior, priqty, privileges, proc, procedure, processexit, program, psid, public, purge
- query, queryno
- raid0, raise, raiserror, range, rank, raw, read, reads, readtext, real, reconfigure, recovery, references, referencing, refresh, regexp, reindex, release, rename, repeat, repeatable, replace, replication, require, reset, resignal, resource, restart, restore, restrict, result, result_set_locator, return, returns, revoke, right, rlike, role, rollback, round_ceiling, round_down, round_floor, round_half_down, round_half_even, round_half_up, round_up, routine, row, rowcount, rowguidcol, rowid, rowlabel, rownum, rownumber, rows, rowset, row_number, rrn, rule, run
- save, savepoint, schema, schemas, scratchpad, scroll, search, second, seconds, second_microsecond, secqty, security, select, sensitive, separator, sequence, serializable, session, session_user, set, setuser, share, show, shutdown, signal, similar, simple, size, smallint, snan, some, soname, source, spatial, specific, sql, sqlbuf, sqlexception, sqlid, sqlstate, sqlwarning, sql_big_result, sql_calc_found_rows, sql_small_result, ssl, stacked, standard, start, starting, statement, static, statistics, statment, stay, stogroup, stores, straight_join, style, substring, successful, sum, summary, synonym, sysdate, sysfun, sysibm, sysproc, system, system_user
- table, tablespace, tape, temp, temporary, terminated, textsize, then, time, timestamp, tinyblob, tinyint, tinytext, to, top, trailing, tran, transaction, trigger, trim, true, truncate, tsequal, type
- uid, uncommitted, undo, union, unique, unlock, unsigned, until, update, update-text, usage, use, user, using, utc_date, utc_time, utc_timestamp
- vacuum, validate, validproc, value, values, varbinary, varchar, varchar2, varchar-character, variable, variant, varying, vcat, verbose, version, view, virtual, volatile, volumes
- waitfor, when, whenever, where, while, with, without, write, writetext, work
- x509, xmlelement, xmlexists, xmlnamespaces, xor
- year, years, year_month
- zerofill

Extension-related field settings

Context	Check	Remarks
Entity field	Entities with sluggable behaviour may not include a field named slug.	If a field is named <i>slug</i> it gets this error as soon as at least one field in this entity has set the <i>sluggable position</i> attribute to a number greater than 0. ModuleStudio creates this slug field automatically in addition to the other fields in the model.
Entity field	The sluggable position values must be unique per entity.	If fields are included into a slug by setting a value greater than 0, this value must be unique per entity. The position defines in which order the field values are considered as permalink parts.
Entity field	Only one field per entity may store the sortable position.	Can occur if one tries to use multiple integer or user fields as position for the sortable extension.
Entity field	The sortable position may not be the sortable group, too.	As soon as a field is used as sortable position it can not also act as the grouping criteria at the same time.
Entity field	Only one field per entity may store the sortable group.	Optional grouping of sorting is only possible with one field as criteria.
Entity field	Only one field per entity may store the version.	Can appear for integer and datetime fields if you enabled the <i>version</i> property for more than one field in the same entity.
Entity field	Entities with a version field should use optimistic locking.	Can appear for integer and datetime fields with the <i>version</i> property enabled. You must set the locking type of the corresponding entity to either <i>OPTIMISTIC</i> or <i>PAGELOCK_OPTIMISTIC</i> , depending on whether you want support the Zikula PageLock functionality in addition or not.
Entity	Entities with optimistic locking must contain a field declared as version field.	The opposite rule to ensure that every entity using optimistic locking has a version field for storing and comparison the version as locking criteria.
Entity field	The version attribute can not be combined with a primary key.	A field can only act as either a version or a primary key, not both at the same time.
Entity field	Entities with geographical extension have additional fields	If an entity has activated the <i>geographical extension</i> Mod

Numeric fields

Context	Check	Remarks
Integer and user fields	The default value for an integer field must contain only digits.	
Integer field	The maximum length for an integer field is 18.	Corresponds to a <i>bigint</i> mapping in Doctrine 2.
Integer field	Minimum value must not be larger than maximum value.	
Integer field	Entities with an aggregate field should use a locking strategy (optimistic or pessimistic read).	If an integer field acts as aggregate field the corresponding entity must use one locking strategy of <i>OPTIMISTIC</i> , <i>PAGE-LOCK_OPTIMISTIC</i> , <i>PESSIMISTIC_READ</i> , <i>PAGE-LOCK_PESSIMISTIC_READ</i> , depending on whether you want support the Zikula PageLock functionality in addition or not.
Integer field	Aggregate fields work only in combination with an outgoing and bidirectional one-to-many relationship with persist cascade.	
Integer field	Naming of <code>aggregateFor</code> attribute values must follow the syntax <code>targetAlias#targetFieldName</code> (for example <code>views#amount</code>).	If an integer field acts as aggregate field the property <i>aggregate for</i> must define the target alias of corresponding outgoing and bidirectional one-to-many relationship with persist cascade. After a <i># char</i> as delimiter the name of the target field (to be aggregated) follows.
Decimal field	The default value for a decimal field must be a floating point number.	
Decimal field	The length (precision) of a decimal field must be greater than the scale.	For example <i>1234.12</i> is valid ($4 > 2$), but <i>123.1234</i> is not.
Decimal field	Minimum value must not be larger than maximum value.	
Float field	The default value for a float field must be a floating point number. 34	
Float field	Minimum value must not be larger than maximum value.	

String and text fields

Context	Check	Remarks
All string fields	String size must not be smaller than 1.	Occurs if length is set to 0.
All string fields	String size must be larger than minimum length.	If you set a minimum length it must not be larger than the actual field length.
String field	String length for country codes must be at least 2 chars.	Occurs if you activate the <i>country</i> property for a field with a length smaller than 2.
String field	String length for languages must be at least 5 chars.	Occurs if you activate the <i>language</i> property for a field with a length smaller than 5.
String field	String length for HTML colour codes must be at least 7 chars.	Occurs if you activate the <i>html colour</i> property for a field with a length smaller than 7.
String field	A string can only be one of country, language, html-colour and password.	
String field	String fields for countries, languages and colour must also activate the nospace validator.	The <i>nospace</i> property ensures that spaces are not allowed as part of the input value. The generator could use it without having set this to true, but as the setting is there anyway the proper solution is to enforce the user activating it for consistency.
String field	String length must not be greater than 255; for bigger sizes use text fields.	
Email field	The default value for an email field must be a valid email address.	
Url field	The default value for an url field must be a valid url.	Allowed protocols are http, ftp and https.

Date and time fields

This section includes rules which apply only for datetime, date and time fields.

Context	Check	Remarks
All date fields	A value can not be in the past and in the future at the same time.	You can only activate either <i>past</i> or <i>future</i> validators.
All date fields	The timestampable change trigger field must point to the name of a field or a relation (property.field).	If the <i>timestampable</i> property for a field has been set to <i>CHANGE</i> then this error can appear to remind you to set also the required attribute <i>timestampableChangeTriggerField</i> . Either you did not set anything there or the value does neither correspond to the name of an entity field nor an incoming relation.
Datetime field	It would be preferable to use an integer column as version field, as datetimes could potentially lead to conflicts for high traffic sites depending on the timestamp resolution in the database.	If you use datetime fields with <i>version</i> set to true this warning will appear. As the Doctrine 2 manual points out integers are more robust against race conditions in high traffic environments where timestamp comparisons are limited due to how precise the used database does it. Therefore you should prefer integer fields for storing versions except side conditions require the usage of datetime fields for that.
Datetime field	The default value for a datetime field must conform to the pattern YYYY-MM-DD HH:MM:SS .	
Date field	The default value for a date field must conform to the pattern YYYY-MM-DD .	
Time field	The default value for a time field must conform to the pattern HH:MM:SS .	

Other fields

Upload field	The <code>allowedExtensions</code> attribute must contain a comma separated list of the file types to be allowed during the upload (example: gif, jpeg, jpg, png). Note that the separator is <code>,</code> including the space char.
Upload field	There must not exist a field named <code>fooMeta</code> because this is reserved for an automatic field storing meta data for this upload.

5.4.4. Relationship

Context	Check	Remarks
Relationship	Relation must be assigned to a model container.	Should not occur in practice, this is just for completeness.
Relationship	Every relation must have a (source target) entity.	Should not occur in practice, this is just for completeness.

Join relationship

Includes basically all relationships in the data layer except inheritance.

Context	Check	Remarks
Join relationship	Every join relation must have a (source target) alias.	Aliases must have a length of at least two chars. Recommended are at least four chars.
Join relationship	Relationship source aliases must be unique for all incoming relations of an entity.	
Join relationship	Relationship target aliases must be unique for all outgoing relations of an entity.	
Join relationship	The (sourceField targetField) attribute must contain either 'id' for automatic primary key or a comma separated list of (source target) fields to be referenced. Note that the separator is ', ' including the space char.	
Join relationship	The amount of join columns must be equal for association source and target sides.	If source or target field are not <i>id</i> ModuleStudio splits both values by the separator above and compares the amount of elements on both sides. This enables joins over multiple fields.
Join relationship	The field <i>sourceField</i> <i>targetField</i> must be an integer field with a length of 11.	Is only checked if the (source target) field is named <i>id</i> .
Join relationship	The fields <i>sourceField</i> and <i>targetField</i> must have the same type and length values.	Is only checked if the source and target fields are both not named <i>id</i> .
Join relationship	Self relation must not reference the <i>sourceField</i> field for both source and target.	For self relation the source field must not be equal to the target field (as the database needs two fields in order to store both sides).
Join relationship	Between two entities there must not be multiple join relations with cascade options.	

One to one relationship

Context	Check	Remarks
One to one relationship	The edit type ACTIVE_CHOOSE_PASSIVE_NONE is only valid for many to many relationships.	
One to one relationship	The edit type ACTIVE_EDIT_PASSIVE_NONE is only valid for many to many relationships.	

Many to one relationship

Context	Check	Remarks
Many to one relationship	The edit type ACTIVE_CHOOSE_PASSIVE_NONE is only valid for many to many relationships.	
Many to one relationship	The edit type ACTIVE_EDIT_PASSIVE_NONE is only valid for many to many relationships.	
Many to one relationship	A many-to-one relation must not be bidirectional.	

One to many relationship

Context	Check	Remarks
One to many relationship	The edit type <code>ACTIVE_CHOOSE_PASSIVE_NONE</code> is only valid for many to many relationships.	
One to many relationship	The edit type <code>ACTIVE_EDIT_PASSIVE_NONE</code> is only valid for many to many relationships.	
One to many relationship	The <code>indexBy</code> attribute points to an invalid field of the target entity.	Checks whether the target entity contains a field equally named like the <i>indexBy</i> property.
One to many relationship	<code>IndexBy</code> fields must be unique.	The target entity field used for <i>indexBy</i> must have <i>unique</i> validator enabled.
One to many relationship	The <code>orderBy</code> attribute points to an invalid field of the target entity.	Checks whether the target entity contains a field equally named like the <i>orderBy</i> property. Currently disabled because also expressions like <i>status ASC</i> , <i>createdDate DESC</i> are allowed. Will probably be removed in a future version.

Many to many relationship

Context	Check	Remarks
Many to many relationship	The <code>indexBy</code> attribute points to an invalid field of the target entity.	Checks whether the target entity contains a field equally named like the <code>indexBy</code> property.
Many to many relationship	<code>IndexBy</code> fields must be unique.	The target entity field used for <code>indexBy</code> must have <code>unique</code> validator enabled.
Many to many relationship	The <code>orderBy</code> attribute points to an invalid field of the target entity.	Checks whether the target entity contains a field equally named like the <code>orderBy</code> property. Currently disabled because also expressions like <code>status ASC</code> , <code>createdDate DESC</code> are allowed. Will probably be removed in a future version.
Many to many relationship	A many-to-many relation must have the <code>refClass</code> attribute defined.	The <code>reference class</code> defines the name of the entity managing the many to many relationship. If for example many <code>persons</code> have many <code>addresses</code> , you could choose <code>personAddress</code> as reference class.
Many to many relationship	The <code>refClass</code> attribute must be unique for all relations.	
Many to many relationship	The <code>refClass</code> attribute must not be equal to any entity name.	The generator creates additional entity classes so unique naming is ensured to prevent naming collisions.

Inheritance relationship

Context	Check	Remarks
Inheritance relationship	Self relations are not allowed for inheritance relationships.	
Inheritance relationship	The discriminatorColumn attribute must not be empty for inheritance relations.	Per default this attribute has the value <i>discr</i> so this shouldn't happen as long as you don't delete this value. ModuleStudio uses this value to tell Doctrine 2 where to store the type of stored entities.
Inheritance relationship	Please rename the <i>discriminatorColumn</i> field as it is reserved for the discriminator column, or change corresponding attribute.	Happens if the target entity includes a field with the same name as specified in the <i>discriminator setting</i> .
Model container with inheritance relationships	Inheritance cycles are not allowed.	Recursive check to detect cycles within inheritance hierarchies.

5.4.5. Entity index

Context	Check	Remarks
Entity index	Index must be assigned to an entity.	Should not occur in practice, this is just for completeness.
Entity index	Every index must have a name.	The index name must have a length of at least two chars. Recommended are at least four chars.
Entity index	Every index must contain at least one item referencing an entity field.	
Entity index	The length of all index fields must not be higher than 333.	The limit is 1000 bytes, while UTF-8 requires three bytes for each char.
Entity index	Index names must be unique per entity.	
Entity index	Index item names must be unique per index.	
Entity index item	The index item must be assigned to an index.	Should not occur in practice, this is just for completeness.
Entity index item	Every index item must have the same name as the referenced entity field.	This occurs if no equally named field is found in the entity.

5.4.6. Variables

Context	Check	Remarks
Variable container	Every var container must have a name.	
Model container with variable containers	Var container names must be unique.	
Model container with variable containers	Var container sort positions must be unique.	
Variable container	Every var container must contain at least one variable.	
Variable container	Var container must be assigned to a model container.	Should not occur in practice, this is just for completeness.
Variable	The var must be assigned to a container.	Should not occur in practice, this is just for completeness.
Variable	Every var must have a name.	The var name must have a length of at least two chars. Recommended are at least four chars.
Variable	A var must not have the same name as an entity.	
List var	The list must contain at least one item.	

5.4.7. Other elements

Context	Check	Remarks
Calculated field	The field must rely on at least one derived field.	Not relevant yet, this is for future.
Entity event listener	The listener must contain at least one operation.	Not relevant yet, this is for future.
Transform object	Every transformation must be assigned to an event listener.	Not relevant yet, this is for future. Should not occur in practice, this is just for completeness.

5.5. Controller layer

5.5.1. Controller container

Context	Check	Remarks
Controller container	The controller container must contain at least one controller.	
Controller container with controllers	You must have an admin controller for hook subscriber functionality.	Since Zikula 1.3 it is highly recommended to have an admin area as there is an additional page for defining hook assignments for different areas.
Controller container with controllers	Controller names must be unique.	For example there may not be two controllers which are both named <i>admin</i> .
Controller container	Please create an ajax controller including an arbitrary action, like main, because it is required for processing your model properly.	This happens if your model layer contains relationships or an entity with either user fields or tree extension. As the generator creates additional ajax methods in these cases it requires an ajax controller for keeping this information also in the model for documentation.

5.5.2. Controller

Context	Check	Remarks
Controller	The controller must be assigned to a controller container.	Should not occur in practice, this is just for completeness.
Controller	Every controller must have a name.	Controller name must be longer than at least three chars. Should be more than five chars.
Custom controller	Controller name must not contain underscores.	Underscores are not allowed as they are used for class autoloading.
Custom controller	Controller name must not be Admin, User, Account, Ajax, Search or Init.	These are reserved names and may therefore not be used for custom controllers.
Controller	Every controller must contain at least one action.	
Controller with actions	There must not exist more than one (main view display edit delete) action in one controller.	
Controller with actions	Names of custom states in one controller must be unique.	
Controller with actions	This controller may not contain a (view display edit delete) action. Please remove it.	Appears for special controllers, like account, ajax and search. Those are reserved for special purposes and not intended to be called by the user as normal controllers.

5.5.3. Action

Context	Check	Remarks
Action	The action must be assigned to a controller.	Should not occur in practice, this is just for completeness.
Custom action	The action must have a name.	Action name must not be <i>New</i> . Must have a length of at least four chars, whereby at least five chars are recommended.
Custom action	Action name must not be Main, View, Display, Edit or Delete.	These are reserved names and may therefore not be used for custom actions.
Action	Every action must be assigned to an action handler.	Not active yet as the controller editor needs some additional features first.

5.5.4. Action handler

Context	Check	Remarks
Action handler	Every action handler must have a name.	Must have a length of at least four chars, whereby at least six chars are recommended.
Action handler	Action handler name must not be List, Detail or Edit.	These are reserved names and may therefore not be used for custom handlers.
Action handler	The action handler must contain at least one action events.	
Action handler with events	The action handler must contain at least one start event.	There must not exist more than one start event in one action handler.
Action handler	Every action handler must have the same name as the main entity it refers to (e.g. <i>CustomerHandler</i>).	Not active yet as the controller editor needs some additional features first.

5.5.5. Action event

Context	Check	Remarks
Action event	Action event must be assigned to an action handler.	Should not occur in practice, this is just for completeness.
Action event	Every action event must have a name.	Must have a length of at least four chars, whereby at least six chars are recommended.

5.5.6. Transition

Context	Check	Remarks
Transition	Transition must be assigned to a controller container.	Should not occur in practice, this is just for completeness. Inactive at the moment anyway.
Transition	Every transition must have a name.	Must have a length of at least two chars, whereby at least four chars are recommended.
Transition	Every transition must have a (source destination) action.	A transition can not link to itself (this is inactive at the moment though).
Controller container with transitions	There must not exist more than one transition between two actions.	

5.6. View layer

5.6.1. View container

Context	Check	Remarks
View container	The view container must contain at least one view.	This rule is currently not active as the view editor is not ready yet.
View container with views	View names must be unique.	For example there may not be two views which are both named <i>display</i> .

5.6.2. View

Context	Check	Remarks
View	Every view must have a name.	View name must be longer than at least three chars. Should be more than five chars.
View	Every view must have a root panel.	
View	Every view must have the same name as the main entity it's controller refers to (e.g. <i>CustomerView</i>).	

5.6.3. Root panel

Context	Check	Remarks
Root panel	Every root panel must have a presenter.	
Root panel	Every root panel must contain at least one composite container.	
Root panel	Every view must contain at least one submit activator.	Ideally every view should contain also at least one cancel activator.
Root panel	Check	

5.6.4. Other UI elements

Context	Check	Remarks
Composite container	Composite container must be assigned to a root panel.	
Tabbed pane	A tabbed pane must contain at least two child containers.	
Sub container	Child container must be assigned to a parent container.	
Sub container	Every sub container must contain at least one field.	
Field	Field must be assigned to a container.	
Form label	Label must point to a field.	

5.6.5. Layout order

Context	Check	Remarks
Layout order	Layout order must be assigned to a presentation container.	
Layout order	Every transition must have a (source destination) container.	A layout order can not link to itself.
View container with layout orders	There must not exist more than one layout order between two containers.	

5.7. Workflow layer

The workflow layer is not implemented yet. This section is just a dummy for future.

5.8. Additional notes

None yet.

6. Application generator

6.1. Introduction

The main use case for ModuleStudio application models is the creation of Zikula extensions. This section describes how the generator works and which artifacts are created from which model elements.

ModuleStudio brings also other generator cartridges (§8.1) for creating further information with regards to documentation and reporting tasks.

6.2. Basic idea

Every application consists of different types of code parts. While some code is unique for each application, most parts can be derived from that and is therefore very similar for a whole software systems family. Those code parts are known as *boilerplate code*.

A very simple example will make this clear very quickly:

```
/**
 * imagine some long comments about this class here
 * @ORM\Entity ...
 * some more annotations
 */
class MyModule_Entity_Person extends Zikula_EntityAccess
{
    /**
     * imagine some long comments about this field here
     * @ORM\Column ...
     * some more annotations
     */
    protected $firstName;
    /**
     * imagine some long comments about this field here
     * @ORM\Column ...
     * some more annotations
     */
    protected $lastName;
}
```

This code has obviously not very much knowledge which is essential for this certain application. Reduced to what is really required from a functional view one would get something like:

```
entity person {  
    string firstName  
    string lastName  
}
```

Thought a little further the generator helps reaching a constantly high code quality, as all implementation details are always considered completely. For example if a new extension is activated for an entity this is not forgotten anywhere inside the code.

6.3. How it works

There is a tutorial showing how calling the generator within ModuleStudio looks like: Using the generator. The only requirement is that you have opened your application model and there are no validation errors (§5) left.

6.3.1. Input dialogs

First you have to select which generator cartridges you want to execute.

After that you have to select which reports you want to create (only if you did not unselect the *reporting* cartridge before).

6.4. The workflow

The rough steps of the generator workflow are as follows:

1. Ask for input parameters (for example desired output folder, cartridges and reports).
2. Empty output directory.
3. Export dumps of editor diagrams (if reporting cartridge is selected).
4. Read input model.
5. Perform validation.
6. Transformation to add primary and foreign key fields.
7. Call generator inner workflow for each selected cartridge.
8. Call code beautifier (if zclassic cartridge is selected).

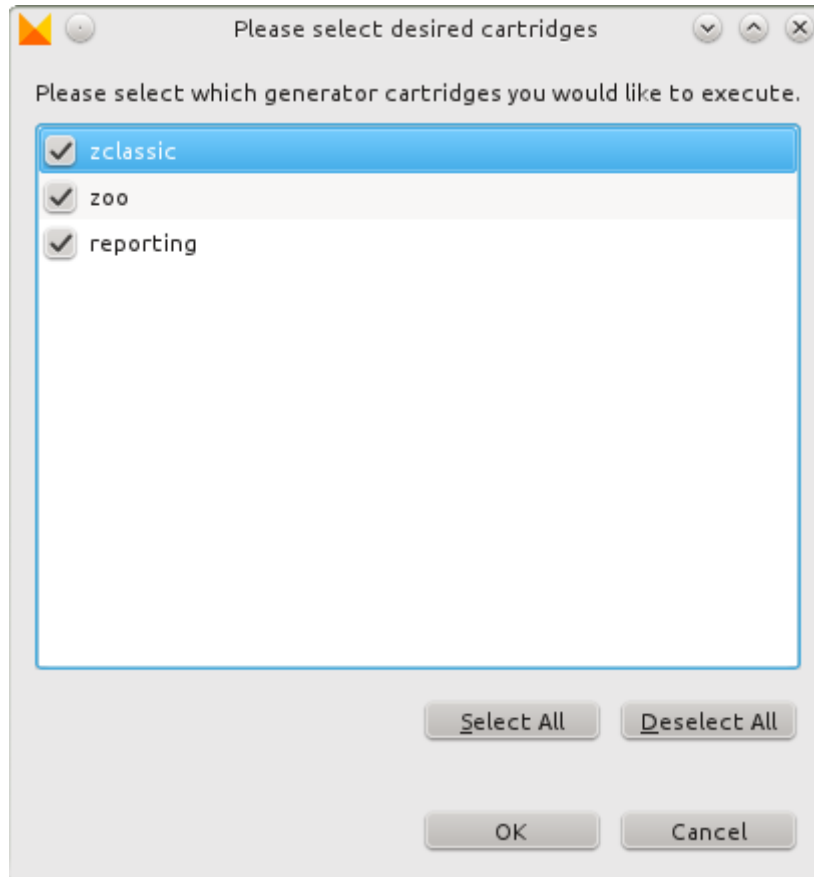


Figure 6.1.: Cartridge selection

6.5. Generator reference

This reference goes through all available model elements as well as their properties, describing what the generator does with this information and which things are still missing in the created implementation.

You will notice that there are also some elements included which are not showing up in ModuleStudio itself yet. This is for showing up the picture we have in mind when designing the modeling language as a whole. In most cases where the generator is not flexible yet this is caused by limitations in the current editors.

Most screenshots in this section are taken from the example application called *Recipe-Manager*.

6.5.1. Application layer

Language elements

Named object This is the common base class of almost all model elements.

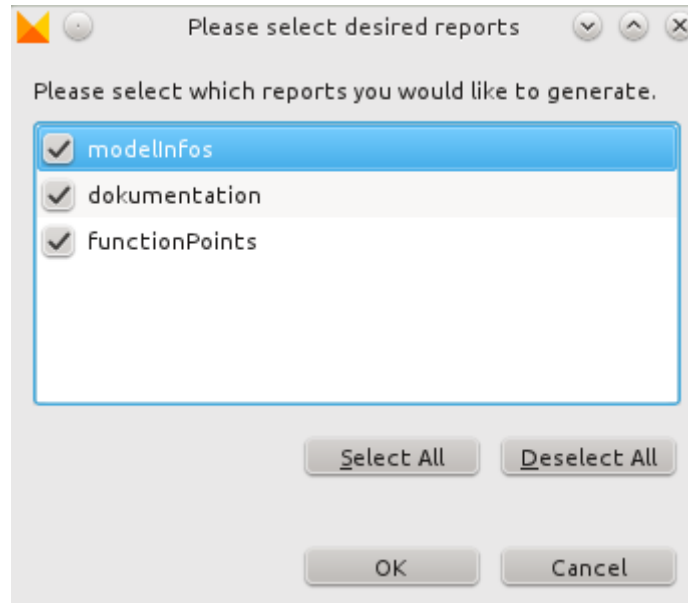


Figure 6.2.: Report selection

It includes the following properties:

- *name* - The name of the element.
- *documentation* - A description for documenting the element.

If a *documentation* is defined for an entity this will be shown right after the heading of the corresponding view template. So you could for example add a description for the *person* entity explaining what persons are and what information they store. If a user does then what the persons list he knows immediately what he is looking at.

Application Represents an application described by the model.

It includes the following basic properties:

- *author* - The author of the application. Usually this is the full name of the developer.
- *email* - The email address of the developer.
- *license* - The license of this application. Defaults to LGPL. If either GPL or LGPL are used the generator creates corresponding license files, too.
- *url* - The homepage of the developer.
- *version* - The application version. Must conform to the pattern *x.y.z* - for example *1.0.0* which is also the default value. Will be used in the version class of the created extension.

These basic fields are mainly used by the generator to create a meaningful file header. An application has some more fields for specifying specific aspects:

- *applicationType* - The kind of application described by the model. See below (§6.5.1).
- *interactiveInstallation* - A boolean specifying whether an interactive installation should be used or not. The default value is *false* which lets the generator create a normal installer without any required user input. If you set it to true it will additionally generate some init templates as well as a corresponding controller for the interactive installer.
- *modelPath* - Physical file path to the application model. This is not used at all by the generator, but only required by ModuleStudio to let the generator workflow find your model.
- *prefix* - A prefix for all database tables of this application. Will be used in entity classes.

An application may furthermore have the following references:

- *controllers* - Allows referencing one or more controller layers (§6.5.1).
- *models* - Allows referencing one or more model layers (§6.5.1).
- *referredApplications* - Allows referencing other applications. See below (§6.5.1).
- *views* - Allows referencing one or more view layers (§6.5.1).

Application type The kind of application described by the model.

Can be one of the following options:

- *WEB* - Web applications. This is the default value.
- *NATIVE* - Native applications.
- *EMBEDDED* - Embedded applications.

At the moment this value is completely ignored by the generator. You should nevertheless keep it set to *WEB* to be on the safe side for future.

Referred application Subclass of Application representing referred applications (e.g. other extensions which are incorporated by api calls). At the moment this is not used yet.

Property	Value
Basis-Daten	
Autor	Axel Guckelsberger
Dokumentation	Comprehensive recipe and menu management.
E-Mail	info@guite.de
Name	RecipeManager
Präfix	recman
URL	http://modulestudio.de
Version	1.0.0
Weitere Optionen	
Anwendungsart	WEB
Interaktive Installation	<input checked="" type="checkbox"/> False
Lizenz	http://www.gnu.org/licenses/lgpl.html GNU Lesser General Public License
Modell-Pfad	/home/axel/RecipeManager.mostapp
Referenzierte Anwendungen	

Figure 6.3.: Application properties

Model container Container class for carrying elements in the model layer.

A model container may have the following references:

- *application* - Reference to the owning element.
- *defaultDataSource* - Whether this container represents the default data source or not. The default value is *true* which can only be assigned to one data source which is treated like an internal Zikula storage. Additional containers are processed as external data sources.
- *entities* - Allows referencing one or more entities (§6.5.2).
- *numExampleRows* - The amount of example rows to create for entities in this model layer. Default value is 5.
- *relations* - Allows referencing one or more relationships (§6.5.2).
- *variables* - Allows referencing one or more variables (§6.5.2).

For each entity as well as for many to many relationships according classes are generated. More details are explained in the model layer section (§6.5.2). However external data sources are not treated correctly (see #5 for more information).

Controller container Container class for carrying elements in the controller layer.

A controller container may have the following references:

- *application* - Reference to the owning element.

- *controllers* - Allows referencing one or more controllers (§6.5.3).
- *handlers* - Allows referencing one or more action handlers (§6.5.3).
- *modelContext* - Allows referencing one or more model containers (§6.5.1).
- *processViews* - Allows referencing a view container (§6.5.1).
- *transitions* - Allows referencing one or more transitions (§6.5.3).

As all the container elements also the controller container is primarily a composite element containing a sublayer. The generator creates according controller classes, but does not separate different controller containers on code level.

View container Container class for carrying elements in the view layer.

A view container may have the following references:

- *application* - Reference to the owning element.
- *controller* - Allows referencing a controller container (§6.5.1).
- *layoutOrders* - Allows referencing one or more layout orders (§6.5.4).
- *views* - Allows referencing one or more views (§6.5.4).

As there is no view editor available yet this element is not relevant for the generator yet. Instead it only creates default templates foreach existing controller action (§6.5.3) and entity (§6.5.2), as well as some common templates which are included or required for some extensions. There are no templates generated though for custom actions (§6.5.3) as this has not been implemented yet.

Combinations and edge cases

Multiple containers may not be considered properly in all implementation areas yet. There are some expressions which must be rechecked in order to limit some generation parts to certain container element instances only. So it can happen for example that several data layer artifacts from additional data sources are treated like if they were part of the primary internal one.

Therefore it can also be that the relations between your containers are not reflected as you would expect it. To some degree this can also be a result of missing dependencies, like for example missing editor capabilities to design the view areas.

So for the moment multiple containers are often treated like if they were one big merged container. With time this will change though, as more and more expressions become more explicit including or rejecting certain model elements more precisely.

6.5.2. Model layer

The model layer in ModuleStudio has been designed for a precise description of entities and associations. To understand all the elements and properties please read the Doctrine 2 documentation before.

Language elements

Entity Represents an entity in the data layer which is mapped to a database table.

It includes the following properties:

- *attributable* - A boolean specifying whether this entity should have attributes or not. If set to *true* the generator creates an additional entity for managing the attributes. During edit (§6.5.3) actions it is possible to input values for three predefined attributes. These will also be shown again on display (§6.5.3) pages. There is no included support yet for arbitrary attributes like they are known from the Categories administration area. While the display side is ready for that, the edit page needs some dynamic support for creating new attributes on the fly.
- *categorisable* - A boolean specifying whether this entity should have categories or not. If set to *true* the generator creates an additional entity for managing the categories. During edit (§6.5.3) actions it is possible to select a desired category. This category will also be shown again on display (§6.5.3) pages. There is no included support yet for having multiple category registries, the generated implementation does only use *Main* as registry. Also there is no built-in filtering or permission scheme based on categories implemented yet.
- *changeTrackingPolicy* - How change detection is being done (see below (§6.5.2)). The default value is *DEFERRED_IMPLICIT*.
- *geographical* - A boolean specifying whether the geographical behaviour is used or not. If set to *true* the generator will create two additional fields named *latitude* and *longitude*. Also it will consider them in all important application areas. During the creation of a new entity with geographical support a nice geolocation feature is used to ask the user for his current location. There is no integration of Google Maps or something else yet in the output, this will be done as part of #34.
- *identifierStrategy* - Whether and which identifier strategy (§6.5.2) is applied. The default value is *NONE*.
- *leading* - A boolean specifying whether this is the primary (and default) entity or not.
- *lockType* - Whether and which locking strategy (§6.5.2) is applied.
- *loggable* - A boolean specifying whether the loggable behavior is used or not. The generator will create an additional entity for managing the log entries if set to *true*. There is no user interface for the version management yet (see #30 for more information).

- *mappedSuperClass* - A boolean specifying this is a mapped superclass or a normal entity. A mapped super class is not able to store data. At the moment the generator creates many unrequired things for mapped super classes though, like for example view templates. The only thing where mapped super classes are already rejected correctly are repository classes.
- *metaData* - A boolean specifying whether this entity should have support for meta data. If set to *true* the generator creates additional inclusion templates for displaying and changing corresponding fields.
- *nameMultiple* - Plural form of the name. The generator uses this for collections, list views and other areas where multiple entities are used.
- *readOnly* - A boolean specifying whether this entity is read only or not. If set to *true* editing will not be possible.
- *slugLength* - Length of slug field. Defaults to 255. An entity is sluggable as soon as at least one of its fields set *sluggable position* to a value greater than 0.
- *slugSeparator* - Separator which will separate words in slug. Default value is - like in Zikula, too.
- *slugStyle* - Which slug style (§6.5.2) is used.
- *slugUnique* - A boolean specifying if the slug is unique or not. Default value is *true*.
- *slugUpdatable* - A boolean specifying if the slug can be changed or not. Default value is *true*.
- *tree* - Whether and which tree strategy is applied. More information about what the generator creates for trees can be found in the the section about entity tree types (§6.5.2).
- *standardFields* - A boolean specifying whether the standard fields behaviour is used or not. If set to *true* the entity will get four additional fields for storing the id of the user who created the item, the id of the user who did the last update, as well as the creation and update dates. This information will be included on display (§6.5.3) and edit (§6.5.3) actions.

Normally all created classes are generated twice. Thereby an empty concrete class inherits from an abstract base class containing the whole generator code. The motivation behind this separation is that your own code keeps free from generated artifacts.

Example:

```
class MyModule_Entity_Validator_Base_Person
    extends MyModule_Validator
{
    // generator code
}
class MyModule_Entity_Validator_Person
```

```
extends MyModule_Entity_Validator_Base_Person
{
    // manual code
}
```

One exception for this scheme is inheritance (§6.5.2).

Please note that arbitrary *filtering is not possible at the moment* until FilterUtil has been upgraded to support Doctrine 2.

Whenever you want to change the default implementation you can add corresponding extensions. If you recognise that you are doing the same changes again and again please submit them as patches for the generator.

An entity may have the following references:

- *container* - Reference to the owning element.
- *fields* - Allows referencing one or more entity fields (§6.5.2).
- *incoming* - Allows referencing one or more incoming relationships (§6.5.2).
- *indexes* - Allows referencing one or more indexes (§6.5.2).
- *listeners* - Allows referencing one or more event listeners (§6.5.2).
- *outgoing* - Allows referencing one or more outgoing relationships (§6.5.2).

One additional note about slugs and permalinks: the generated short url handlers will understand different url schemes for the display pages (§6.5.3) depending on the entity settings.

- Entities which are not sluggable use the identifier for display urls, for example *mymodule/person/5.html*.
- Entities with unique slugs use the slug for display urls, for example *mymodule/person/william-smith.html*.
- Entities with non-unique slugs combine both methods, for example *mymodule/person/william-smith.5.html*.

Entity field Represents an entity field in the data layer.

This base class has the following children at the moment:

- Derived fields (§6.5.2) correspond to normal columns which are stored in a database.
- Calculated fields (§6.5.2) correspond to fields which can calculate their values based on other fields.

An entity field may have the following references:

- *entity* - Reference to the owning element.

Derived field Represents an entity field in the data layer which is mapped to a database column. A derived field comes straight from the data source.

A calculated field has the following properties in addition to the common entity field (§6.5.2) settings:

- *defaultValue* - The default value of the field. This default value is used when creating new entities with the edit action (§6.5.3).
- *leading* - A boolean specifying whether this is the primary (and default) field in this entity. Default value is *false*. Every entity must contain one leading field which is used for headings, dropdown values and default sorting. Usually this is something like *name* or *title* - if a textual field is part of the entity at all.
- *mandatory* - A boolean specifying whether this field is mandatory or not. The default value is *true*.
- *nullable* - A boolean specifying whether the field may be null or not. The default value is *false*. A nullable field may not be mandatory at the same time.
- *primaryKey* - A boolean specifying whether this is a primary key field or not. Default value is *false*. Usually there is no need to enable this for any fields as the generator adds primary and foreign key fields automatically. The only use case where the manual definition of primary keys makes sense is having composite keys. This should work in general with regards to the generated data layer, but support on controller and view layers in the created application may not be prepared properly yet for that.
- *readonly* - A boolean specifying whether this a read only field or not. The default value is *false*. If set to *true* then this field may not be changed during editing.
- *sluggablePosition* - Position of this field in the created slugs. A value of 0 means that this field is not part of the slug at all. If at least one field in an entity has a sluggable position greater than 0 then this entity is considered as sluggable. In this case a permalink is built automatically from all fields in ascending position. See the slug properties on entity level (§6.5.2) for slug-related configuration options.
- *sortableGroup* - A boolean specifying whether this field acts as grouping criteria for the sortable extension. The default value is *false*.
- *translatable* - A boolean specifying whether this field is translatable or not. The default value is *false*. If at least one field in an entity is translatable the generator creates an additional class for managing the translation entities. Overall support for translations in the application is almost finished (see #31 for more information).
- *unique* - A boolean specifying whether this field is unique or not. The default value is *false*. If set to *true* then an additional validator cares for enforcing the unique constraint on client and server level.

All fields are implemented as entity class member vars. The following sections will look at the different field types in detail.

Calculated field Represents an entity field which can dynamically compute its value based on other fields.

A calculated field may have the following references in addition to the common entity field (§6.5.2) settings:

- *operands* - Allows referencing one or more derived fields (§6.5.2).

Calculated fields are not part of the model editor yet and will therefore be ignored by the generator.

Boolean field Represents a field type for storing boolean values.

A boolean field has no fields or references in addition to the common entity field (§6.5.2) settings.

The generator will treat boolean values as checkbox input elements in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *yesno* modifier is used to show an image indicating the boolean value (green check or red cross).

Abstract integer field Represents an abstract integer field for grouping different implementations of this field type.

An abstract integer field has the following properties in addition to the common entity field (§6.5.2) settings:

- *length* - The length of this field. This controls whether the Doctrine mapping type will be *integer*, *bigint* or *smallint*. Default value is 11.
- *sortablePosition* - A boolean specifying whether this field stores the position for the sortable extension or not. If set to *true* this field will be used as default sorting criteria. There is no built-in reordering possibility, for example with drag n drop, implemented yet (see #29 for more information).

Integer field Represents a field type for storing integer numbers.

An integer field has the following properties in addition to the common abstract integer field (§6.5.2) settings:

- *aggregateFor* - Aggregate field: one-to-many target alias and field name (syntax: *views#amount*) which causes the generator creating special methods for aggregation.
- *maxValue* - Maximal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *minValue* - Minimal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *version* - A boolean specifying whether this field should act as a version. If set to *true* the owning entity will need to use optimistic locking (§6.5.2). There is no user interface for version management generated yet.

In edit (§6.5.3) pages the generator will use int input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) the value will just be shown.

Decimal field Represents a field type for storing decimal numbers.

A decimal field has the following properties in addition to the common entity field (§6.5.2) settings:

- *aggregationField* - A boolean specifying whether this field should act as an aggregate field. If set to *true* the generator creates special methods for aggregation.
- *length* - The length of this field. Default value is 10.
- *maxValue* - Maximal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *minValue* - Minimal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *scale* - The amount of digits after the dot. Default value is 2.

In edit (§6.5.3) pages the generator will use float input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) the value will just be shown using the *formatnumber* modifier.

Float field Represents a field type for storing float numbers.

A float field has the following properties in addition to the common entity field (§6.5.2) settings:

- *aggregationField* - A boolean specifying whether this field should act as an aggregate field. If set to *true* the generator creates special methods for aggregation.
- *length* - The length of this field. Default value is 10.
- *maxValue* - Maximal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *minValue* - Minimal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.

In edit (§6.5.3) pages the generator will use float input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) the value will just be shown using the *formatnumber* modifier.

Abstract string field Represents an abstract string field for grouping different implementations of this field type.

An abstract string field has the following properties in addition to the common entity field (§6.5.2) settings:

- *fixed* - A boolean specifying whether this field has a fixed length or not.
- *minLength* - Minimal length. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *nospace* - A boolean specifying whether space chars are forbidden or not.
- *regex* - Regular expression to validate against.

If one of these properties is set to *true* a corresponding validator will check this constraint on client and server level.

String field Represents a field type for storing string values.

A string field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *country* - A boolean specifying whether this field represents a country code or not. If set to *true* this still results in a normal text input element as there is no country selector available yet in Zikula.
- *htmlcolour* - A boolean specifying whether this field represents a html color code (like #003399) or not. If set to *true* a colour picker is used in edit (§6.5.3) pages for convenient selection of colour codes.
- *language* - A boolean specifying whether this field represents a language code or not. If set to *true* a language selector is used in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *getlanguagename* modifier is used to display the full name instead of the unreadable language code.
- *length* - The length of this field. Default value is 10000.
- *password* - A boolean specifying whether this field represents a password or not. If set to *true* a password input element will be used instead of a normal one in edit (§6.5.3) pages.

In edit (§6.5.3) pages the generator will use singleline input elements for string fields - except you defined something else (like language or password). Other validations are added together and applied as well.

Text field Represents a field type for storing larger text.

A text field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *length* - The length of this field. Default value is 10000.

In edit (§6.5.3) pages the generator will use multiline input elements (textarea).

User field Extension of abstract integer field (§6.5.2) for storing user ids.

An user field has no fields or references in addition to the common abstract integer field (§6.5.2) settings.

In edit (§6.5.3) pages the generator will implement an auto completion element allowing searching users by their name. For the output in view (§6.5.3) and display (§6.5.3) templates the user name is shown and linked to the corresponding user profile in case a profile module has been set in the Settings module administration.

(Note: The implementation of the auto completion should be replaced by a central forms plugin, see this ticket for more information.)

Email field Represents a field type for storing email addresses.

An email field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *length* - The length of this field. Default value is 255.

In edit (§6.5.3) pages the generator will use email input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) an icon will be shown linking the email address.

Url field Represents a field type for storing urls.

An url field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *length* - The length of this field. Default value is 255.

In edit (§6.5.3) pages the generator will use url input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) an icon will be shown linking the url.

Upload field Represents a field type for storing upload files.

An upload field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *allowedExtensions* - List of file extensions to be accepted during the upload, separated by a comma with a space char. Default value is *gif, jpeg, jpg, png*.
- *length* - The length of this field. Default value is 255.
- *namingScheme* - Defines how uploaded files are named (§6.5.2).
- *subFolderName* - Name of sub folder for storing uploaded files. If this is empty the field name will be used as folder name.

In edit (§6.5.3) pages the generator will use upload input elements. If a field is mandatory the upload will be required when creating a new entity, but not when editing an existing one. If a field is optional (not mandatory) then it will be possible to delete existing uploads on editing.

For the output in view (§6.5.3) and display (§6.5.3) a download link is shown together with the filesize. If the file is an image then a small version of it is shown instead of a text link (on edit pages too by the way).

If an application has any upload fields the generator creates an additional util class containing methods for image processing. The generated application uses it to create and store thumbnails on demand with the help of the Imagine library which is included in Zikula 1.3. There is a view modifier available which works together with the mentioned util class and understands many parameters to use arbitrary images in the templates.

For every upload field *foo* there will be another array field (§6.5.2) created which is named *fooMeta*. This field stores some meta information about the uploaded files for convenience, like the file size, the image format (portrait, landscape, square) and the image dimensions.

Upload naming scheme Represents different schemes for naming uploaded files.

Can be one of the following options:

- *ORIGINALWITHCOUNTER* - Keep the original file name. Add a counter if required to avoid duplicated file names.
- *RANDOMCHECKSUM* - Use a random checksum. This results in quite cryptic filenames.
- *FIELDNAMEWITHCOUNTER* - Use the field name as a prefix together with a counter. For example *image1*, *image2*, and so on.

Within the generated *upload handler* class one of those strategies will be selected depending on from which entity the currently treated upload file.

Array field Represents a field type for storing arrays.

An array field has no fields or references in addition to the common entity field (§6.5.2) settings.

The generator will exclude arrays in edit (§6.5.3) pages as well as for the output in view (§6.5.3) and display (§6.5.3) templates.

Object field Represents a field type for storing objects.

An object field has no fields or references in addition to the common entity field (§6.5.2) settings.

The generator will exclude objects in edit (§6.5.3) pages as well as for the output in view (§6.5.3) and display (§6.5.3) templates.

Abstract date field Represents an abstract date dependant field for grouping those field types.

An abstract date field has the following properties in addition to the common entity field (§6.5.2) settings:

- *future* - A boolean specifying whether the value must be in the future or not.
- *past* - A boolean specifying whether the value must be in the past or not.
- *timestampable* - Which timestampable type (§6.5.2) is used.
- *timestampableChangeTriggerField* - Optional name of field to use as change trigger (if type is *CHANGE*).
- *timestampableChangeTriggerValue* - Optional value of field to use as change trigger (if type is *CHANGE*).

The *past* and *future* properties are implemented as client-side and server-side validators.

The generator transforms the timestampable attributes to the corresponding implementation as is. There are no differences made between the different timestampable types.

Datetime field Represents a field type for storing datetime values with the format *YYYY-MM-DD H:i:s*.

A datetime field has the following properties in addition to the common abstract date field (§6.5.2) settings:

- *version* - A boolean specifying whether this field should act as a version. If set to *true* the owning entity will need to use optimistic locking (§6.5.2). Please read more at the integer field (§6.5.2) section. Also please note that it is preferred to use integer fields instead of datetime fields for version storage (read more in the validation chapter (§5.4.3)).

The generator will treat datetime values as date input elements with the *includeTime* attribute set to true in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *datetime* modifier is used to format the datetime according to the current locale.

Date field Represents a field type for storing date values with the format *YYYY-MM-DD*.

A date field has no fields or references in addition to the common abstract date field (§6.5.2) settings.

The generator will treat date values as date input elements with the *includeTime* attribute set to false in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *datetime* modifier is used to format the date according to the current locale.

Time field Represents a field type for storing time values with the format *H:i:s*.

A time field has no fields or references in addition to the common abstract date field (§6.5.2) settings.

The generator will treat time values as text input elements with a maximum length in edit (§6.5.3) pages as long as there is no time field available in Zikula). For the output in view (§6.5.3) and display (§6.5.3) templates the *datetime* modifier is used to format the time according to the current locale.

Entity identifier strategy Represents different strategies for identifier generation.

Can be one of the following options:

- NONE - No explicit strategy.
- AUTO - Choose automatically.
- SEQUENCE - Uses a database sequence.
- TABLE - Uses a single-row database table and a hi/lo algorithm.
- IDENTITY - Obtains IDs from special identity columns (auto_increment).

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different index types. So beside the actual entity class there won't be any code parts affected based on which identifier strategy you use.

Entity change tracking policy Represents different policies defining how changes are determined.

Can be one of the following options:

- *DEFERRED_IMPLICIT* - Compare properties during commit. Convenient, but not good for performance.
- *DEFERRED_EXPLICIT* - Scan only entities marked for change detection. Better performance, but no dirty checking.
- *NOTIFY* - Assume that entities inform listeners about their changes.

The generator transforms these values to the corresponding implementation as is. For *notify* the generator creates according notification calls within the entity setter methods.

Entity lock type Represents different locking strategies for entities.

Can be one of the following options:

- NONE - No locking support.
- OPTIMISTIC - Optimistic locking.

- PESSIMISTIC_READ - Pessimistic read locking.
- PESSIMISTIC_WRITE - Pessimistic write locking.
- PAGELOCK - Use PageLock module.
- PAGELOCK_OPTIMISTIC - Use PageLock module combined with optimistic locking.
- PAGELOCK_PESSIMISTIC_READ - Use PageLock module combined with pessimistic read locking.
- PAGELOCK_PESSIMISTIC_WRITE - Use PageLock module combined with pessimistic write locking.

The generator transforms these values to the corresponding implementation as is. If you use optimistic locking the entity needs a version field which can be an integer (§6.5.2) or datetime (§6.5.2) field whereby an integer is preferred.

If you choose an option including the PageLock module the form handlers generated for the edit (§6.5.3) pages will call some api functions of the PageLock extension in order to incorporate these features.

Entity tree type Represents different tree strategies for entities.

Can be one of the following options:

- NONE - No tree.
- NESTED - Nested set.
- CLOSURE - Closure.

If an entity has a tree type other than *NONE* then the generator creates several additional artifacts, like for example:

- An additional template for managing the tree in a hierarchy view.
- An additional view plugin for including the Zikula tree javascript.
- Some ajax functions used by the hierarchy view.
- For closure: separate classes for the closure entities.

Entity slug style Represents different slug styles for the creation of permalinks.

Can be one of the following options:

- LOWERCASE - Lowercase.
- CAMEL - Camelcase.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different slug styles. So beside the actual entity class there won't be any code parts affected based on which slug style you use.

Entity timestampable type Represents different events for triggering the Timestampable extension.

Can be one of the following options:

- NONE - No Timestampable behaviour.
- UPDATE - On update.
- CREATE - On create.
- CHANGE - On property change.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different timestampable types. So beside the actual entity class there won't be any code parts affected based on which timestampable type you use.

Entity index Represents an entity index.

It includes the following properties:

- *type* - The index type.

An index may have the following references:

- *entity* - Reference to the owning element.
- *items* - Allows referencing one or more index items (§6.5.2).

Entity index type Represents different types of entity indexes (§6.5.2).

Can be one of the following options:

- NORMAL - Normal index.
- UNIQUE - Unique constraint.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different index types. So beside the actual entity class there won't be any code parts affected based on which index type you use.

Entity index item Represents a part of an index (§6.5.2), referencing to an equally-named entity field (§6.5.2).

An index item may have the following references:

- *index* - Reference to the owning element.

Relationship Base class for all types of associations between entities.

It includes the following properties:

- *bidirectional* - A boolean specifying whether this relationship is bidirectional or not. The default value is *false* for performance reasons.

A relationship may have the following references:

- *container* - Reference to the owning element.
- *source* - Allows referencing a target entity (§6.5.2).
- *target* - Allows referencing a source entity (§6.5.2).

Join relationship Collects all foreign key and join relationships.

It includes the following properties in addition to the common relationship (§6.5.2) settings:

- *cascade* - The cascade type (§6.5.2) used on application level.
- *editType* - The edit type (§6.5.2) for this association.
- *fetchType* - The fetch type (§6.5.2) for this association.
- *nullable* - A boolean specifying whether the field for this relationship may be null or not. The default value is *true*.
- *onDelete* - String for optional update cascade options on database level (for example *RESTRICT* or *SETNULL*).
- *onDelete* - String for optional delete cascade options on database level (for example *RESTRICT* or *SETNULL*).
- *sourceAlias* - The alias for the source entity, required to have multiple associations between the same entities. The name should reflect the cardinality on the source side (singular or plural forms) depending on the relationship type. As with all names camel case is preferred, for example *personAddresses*.
- *sourceField* - Name of the source entity fields used for the join. The default value is *id* which means that the source entity is joined by it's primary key. It is possible to change that value for custom join conditions. Furthermore it is possible to use multiple field names separated by a comma with a space in order to join entities with composite keys.
- *targetAlias* - The alias for the target entity, required to have multiple associations between the same entities. The name should reflect the cardinality on the target side (singular or plural forms) depending on the relationship type. As with all names camel case is preferred, for example *personAddresses*.

- *targetField* - Name of the target entity fields used for the join. The default value is *id* which means that the target entity is joined by its primary key. It is possible to change that value for custom join conditions. Furthermore it is possible to use multiple field names separated by a comma with a space in order to join entities with composite keys.
- *unique* - A boolean specifying whether the field for this relationship is unique or not. The default value is *false*.

The generator transforms most of these settings to the corresponding implementation as is. The only thing which is used outside of the entity classes is the edit type (§6.5.2) which controls how relationships are handled in edit actions (§6.5.3).

Join relationships are automatically incorporated into the dql queries which are placed in the entity repository classes. You can override these methods for changing selection details if required.

One to one relationship Represents one-to-one relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *orphanRemoval* - Default value is *false*. If set to *true* orphans get removed automatically.
- *primaryKey* - A boolean specifying whether the foreign key of this relation should act as a primary key. The default value is *false*. Please note that this has not been tested yet and probably won't be supported properly yet by the controller layers in the generated application.

One to many relationship Represents one-to-many relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *indexBy* - Set to target field name (must be unique) to specify the index by criteria for the relation. Please note that this has not been tested very well yet.
- *orderBy* - Set to target field name to specify the sorting criteria for the relation.
- *orphanRemoval* - Default value is *false*. If set to *true* orphans get removed automatically.

Many to one relationship Represents many-to-one relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *primaryKey* - A boolean specifying whether the foreign key of this relation should act as a primary key. The default value is *false*. Please note that this has not been tested yet and probably won't be supported properly yet by the controller layers in the generated application.

Many to many relationship Represents many-to-many relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *indexBy* - Set to target field name (must be unique) to specify the index by criteria for the relation. Please note that this has not been tested very well yet.
- *orderBy* - Set to target field name to specify the sorting criteria for the relation.
- *refClass* - Specifies the reference class created for the linking table (for example *personAddress*). The generator creates additional classes for this reference-managing entity.

Cascade type Represents different cascade types on application level.

Can be one of the following options:

- NONE
- PERSIST
- REMOVE
- MERGE
- DETACH
- PERSIST_REMOVE
- PERSIST_MERGE
- PERSIST_DETACH
- REMOVE_MERGE
- REMOVE_DETACH
- MERGE_DETACH
- PERSIST_REMOVE_MERGE
- PERSIST_REMOVE_DETACH
- PERSIST_MERGE_DETACH
- ALL

The cascade type is implemented as defined in the association's annotation within the corresponding entity classes. At the moment there are no other code parts depending on that.

Relation fetch type Represents different fetch types for join relationships.

Can be one of the following options:

- LAZY - Lazy.
- EAGER - Eager.
- EXTRA_LAZY - Extra lazy.

The generator transforms these values to the corresponding implementation. There are no differences made yet between the different fetch types as the generator uses DQL for almost all selections. So beside the actual entity class there won't be any code parts affected based on which fetch type you use.

Relation edit type Represents different edit types for join relationships.

Can be one of the following options:

- ACTIVE_NONE_PASSIVE_CHOOSE - Editing the parent does nothing. Editing the child includes choosing the parent.
- ACTIVE_NONE_PASSIVE_EDIT - Editing the parent does nothing. Editing the child includes choosing, adding and editing the parent.
- ACTIVE_CHOOSE_PASSIVE_NONE - Only for many-to-many: Editing the parent includes choosing the children. Editing the child does nothing.
- ACTIVE_EDIT_PASSIVE_CHOOSE - Editing the parent includes choosing, adding and editing the children. Editing the child includes choosing the parent.
- ACTIVE_EDIT_PASSIVE_EDIT - Editing the parent includes choosing, adding and editing the children. Editing the child includes choosing, adding and editing the parent.
- ACTIVE_EDIT_PASSIVE_NONE - Only for many-to-many: Editing the parent includes choosing, adding and editing the children. Editing the child does nothing.

Actually these settings are part of the view layer (§6.5.4) and actually they are going to be moved somewhere in future. But this is a case where we decided that we don't want to wait until the view layer is ready to be used (as priorities require to do other things first).

For each entity the generator creates some templates to be included in the edit templates of related entities (for example a display list and another one for edit). Depending on which edit type is defined for a relationship the corresponding edit template (choose or edit) is included or not.

- NONE means that there is no possibility to take influence on the association.
- CHOOSE means that it is possible to select a related entity with the help of auto completion.
- EDIT means the same as CHOOSE plus that it is also possible to create and edit related entities during editing the main entity.

Inheritance relationship Represents inheritance relationships for describing entity class hierarchies.

It includes the following properties in addition to the common relationship (§6.5.2) settings:

- *discriminatorColumn* - Name of the field used for storing the entity type.
- *strategy* - The inheritance strategy used for data storage.

The generator considers inheritance for all classes which are created for each entity. This includes naturally the entity classes itself, but also additional classes like repositories, validators or additional entities for extensions like attributes, categories, log entries, translations and more.

As explained in the entity section (§6.5.2) all generated concrete classes inherit from corresponding abstract base classes. As soon as an entity does inherit from another one, there will be no base class created for it. Instead the concrete implementation class will inherit from the concrete class of the parent entity.

Example:

```
class MyModule_Entity_Validator_Base_Person
    extends MyModule_Validator
{
    // generator code
}
class MyModule_Entity_Validator_Person
    extends MyModule_Entity_Validator_Base_Person
{
    // manual code
}
class MyModule_Entity_Validator_Customer
    extends MyModule_Entity_Validator_Person
{
    // manual code
}
```

While this implementation approach is quite elegant it is not completed yet in all areas unfortunately. At least during installation everything should be fine. When working with the application you will notice that inherited fields are handled well, but additional fields from the parent classes are not considered yet everywhere. See #46 for more information.

Inheritance strategy type The strategy type defines which kind of inheritance strategy should be used.

Can be one of the following options:

- `SINGLE_TABLE` - Simple inheritance: share everything and store it in the parent table.
- `JOINED` - Concrete inheritance: each entity stores everything in its own table.

The generator transforms these values to the corresponding implementation. There are no differences made yet between the different strategies. So beside the actual entity class there won't be any code parts affected based on which strategy you use.

Variables Container class for carrying module vars.

It includes the following properties:

- *sortOrder* - The sorting position for when using multiple variable sections.

A var container may have the following references:

- *container* - Reference to the owning element.
- *vars* - Allows referencing one or more variables (§6.5.2).

As soon as at least one variable container exists the generator creates a *config* page in the admin area to let the site admin manage corresponding settings.

If a model contains multiple variable containers the config page will use a tabbed panel containing a tab for each container, sorted by the *sortOrder* field. This allows separating settings in bigger models into logical semantic groups.

Variable Represents a module variable.

It includes the following properties:

- *name* - Name of the variable.
- *value* - Default value of the variable.

A variable may have the following references:

- *container* - Reference to the owning element.

For each variable the generator creates an according input element in the config page. Also the variable is handled properly in the installer classes which takes care for initialisation and removal on uninstallation.

If you enabled interactive installation there will also be an init page asking for the initial values for all variables. However this is not matured very well yet.

Text var Represents a setting with alphanumeric values.

It includes the following properties in addition to the common variable (§6.5.2) settings:

- *maxLength* - The maximum length.

The generator creates an input for text for a text variable. The maximum length is not considered anywhere in the generated code yet.

Int var Represents a setting with numeric (integer) values.

The generator creates an input element for integers (digits) for an integer variable.

Bool var Represents a setting with boolean values.

The generator creates a checkbox input element for a boolean variable.

File path var Represents a setting with file path values.

It includes the following properties in addition to the common variable (§6.5.2) settings:

- *withinDocRoot* - A boolean specifying whether this path is placed inside the web root or not. The default value is *true*.
- *writable* - A boolean specifying whether this file path is writable or not. The default value is *false*.

The generator will create a text input element for a file path variable as well as additional behaviour to consider it's fields properly. At the moment this has not been done yet though, so file path vars are currently handled in the same way like text vars.

List var Represents a setting with list values.

It includes the following properties in addition to the common variable (§6.5.2) settings:

- *multiple* - A boolean specifying whether multiple items can be selected concurrently or not. The default value is *false*.

A list variable may have the following references:

- *items* - Allows referencing one or more items (§6.5.2).

The generator will create a select element for a list variable. At the moment this has not been done yet though, so list vars are currently handled in the same way like text vars.

List var item Represents an entry for a setting with list values.

It includes the following properties:

- *default* - A boolean specifying whether this entry is selected by default or not. The default value is *false*.
- *name* - Name of the item.

The generator will create an option element for the corresponding select element. At the moment this has not been done yet though, so list vars are currently handled in the same way like text vars.

Entity event listener Base class for model elements representing entity event listeners.
List of supported events:

- PrePersist / PostPersist
- PreUpdate / PostUpdate
- PreRemove / PostRemove
- PostLoad

An event listener may have the following references:

- *operations* - Allows referencing one or more transform objects (§6.5.2).

Event listeners are not part of the model editor yet and are therefore ignored by the generator at the moment. Instead the generator creates simply all listener methods for all entities.

Transform object Base class for transformation objects encapsulating algorithm puzzle pieces.

Examples:

- Arithmetic operations
- Datetime operations
- Financial operations
- Logical operations
- String operations, e.g. replacements
- System calls

A transform object may have the following references:

- *container* - Reference to the owning element.

Transform objects are not part of the model editor yet and are therefore ignored by the generator at the moment.

Combinations and edge cases

This section is going to collect certain combinations of elements in practical scenarios. The intention of this section is to point out dependencies and other essential aspects relating the combination of model elements in various ways.

6.5.3. Controller layer

Language elements

Controller A controller represents an area with functions which are called actions (§6.5.3). In Zikula a controller is identified with the *type* parameter.

The following controller types are available:

- Admin controller - for admin areas.
- User controller - for user areas.
- Account controller - for account sections.
- Ajax controller - for ajax functions.
- Search controller - for a search-related controller and search api.
- Custom controller - for custom areas, like *edit*.

A controller may have the following references:

- *actions* - Allows referencing one or more actions (§6.5.3).
- *container* - Reference to the owning element.
- *handlers* - Allows referencing one or more action handlers (§6.5.3).

The generator creates controller classes with methods for each actions. Action handlers are not considered yet. Also special functionalities for account and search controllers are not implemented yet.

If you have added some variables (§6.5.2) the admin controller will contain a *config* method for managing the modvar settings.

Action An action represents a controller function which can be called by the user. In Zikula an action is identified with the *func* parameter.

The following action types are available:

- Main action - default function.
- View action - processes a collection of entities.
- Display action - shows a certain entity in detail.
- Edit action - an action for editing an entity.
- Delete action - an action for deleting an entity.
- Custom action - for custom actions, like *mySpecialFunction*.

An action may have the following references:

- *controller* - Reference to the owning element.
- *handler* - Allows referencing an action handler (§6.5.3).
- *incoming* - Allows referencing one or more transitions (§6.5.3).
- *outgoing* - Allows referencing one or more transitions (§6.5.3).

The generator creates sensitive default implementations for all action types except custom actions which are not treated very well yet.

It is possible to create special versions for all templates by adding a suffix which will be assigned by the *tpl* parameter. For example you can call *display_myversion.tpl* by adding *tpl=myversion* to the url.

Main action A main action implementation does either do a simple redirect to the view function or (if no view is available) fetch a simple template file.

View action The view implementation offers a generic list view of multiple items which can be sorted and filtered. Also there are alternative template formats created for atom, csv, json, rss and xml support. Just add *userssect=1* to the url or, even easier with shorturls, change *persons.html* to *persons.rss*.

Display action A display action results in a generic detail view of an entity. Again there are alternative formats supported, like for example csv, json and xml.

Edit action The edit implementation creates form pages for changing entities and their relations. Beside the form handler classes this involves according template files.

Delete action For a delete action the generator creates the well-known confirmation page asking the user whether he really wants to delete the given entity.

Custom action A custom action is only created as a mockup which contains the permission check as well as some other stuff which is always required. It is not complete yet, as for example a template file is missing.

Action handler Action handlers encapsulate the reaction on user interactions. The most common instance is a form handler processing and validating some input fields.

The following handler types are available:

- List handler - reacts on interactions with a view action (§6.5.3) (e.g. for filtering and sorting).
- Detail handler - reacts on interactions with a display action (§6.5.3).

- Edit handler - reacts on interactions with an edit action (§6.5.3).
- Custom handler - reacts on interactions with a custom action (§6.5.3).

A handler may have the following references:

- *events* - Allows referencing one or more events (§6.5.3).
- *action* - Reference to the owning element.
- *container* - Reference to the owning element.
- *controller* - Allows referencing a controller (§6.5.3).

Not used yet by the generator at all.

Action event An action event represents something which can happen within the scope of an action handler. For example this could be the press of a button.

The following event types are available:

- Start / initial event.
- Normal event.
- End / final event.

An event may have the following references:

- *handler* - Reference to the owning element.

Not used yet by the generator at all.

Transition Transitions define how the user may move between the available use cases defined by the controller actions. (§6.5.3)

The following transition types are available:

- Redirect - automatic.
- User transition - manual.

It includes the following properties:

- *condition* - A conditional expression which must be or become true to activate the transition.

A transition may have the following references:

- *container* - Reference to the owning element.
- *source* - Reference to source action.
- *sourceEvent* - Reference to the source action's event causing this transition.
- *target* - Reference to target action.

Not used yet by the generator at all.

Combinations and edge cases

At the moment the controller editor is almost limited to the creation of controller and action elements. As stated above action handlers and action events are not really used yet. Therefore there the variation in this layer is not that huge yet, so there are no special aspects to explain here.

6.5.4. View layer

As there is no view editor available yet all the following elements are not relevant for the generator yet. Instead it only creates default templates for each existing controller action (§6.5.3) and entity (§6.5.2), as well as some common templates which are included or required for some extensions. There are no templates generated though for custom actions (§6.5.3) as this has not been implemented yet.

Language elements

View The base view class. Each view instance would correspond to a template in the generated Zikula application.

The following view types are available:

- List view.
- Detail view.
- Edit view.
- Custom view.

A view may have the following references:

- *viewContainer* - Reference to the owning element.
- *viewRoot* - Reference to root panel.

Tables The following table types are available:

- Layout table (is a composite container (§6.5.4)).
- Data table (is a sub container (§6.5.4)).

Container Represents a very generic container.

A container may have the following references:

- *cancelActivators* - Allows referencing one or more cancel activators (§6.5.4).
- *submitActivators* - Allows referencing one or more submit activators (§6.5.4).

Root panel Top-level container for a certain view.

A root panel may have the following references in addition to the generic container fields:

- *containers* - Allows referencing one or more composite containers (§6.5.4).
- *presenter* - Reference to the owning element.

Composite container A composite container which may include other sub containers.

A composite container may have the following references in addition to the generic container fields:

- *childContainer* - Allows referencing one or more sub containers (§6.5.4).
- *rootPanel* - Reference to the owning element.

Tabbed pane A composite container which groups its sub containers with tabs.

Sub container A sub container which may not include other sub containers.

A sub container may have the following references in addition to the generic container fields:

- *fields* - Allows referencing one or more fields (§6.5.4).
- *parentContainer* - Reference to the owning element.

Panel Inherits from both composite container (§6.5.4) and sub container (§6.5.4).

Search panel A dedicated panel for search functions.

A search panel may have the following references in addition to the sub container fields:

- *searchButton* - Allows referencing a submit button (§6.5.4).

Form Extension of sub container (§6.5.4) for form elements.

Field Represents a field with information.

It includes the following properties:

- *cssClass* - Optional specification of arbitrary css classes.
- *defaultValue* - The default value (which may be different from that in the data layer).

A field may have the following references:

- *fieldContainer* - Reference to the owning element.

Display field Shows the value of a certain field from the data layer.

Input field Shows an input for a certain field from the data layer.

It includes the following properties in addition to the common field (§6.5.4) settings:

- *hasInitialFocus* - Whether this input should receive the initial focus or not.
- *id* - Defines the markup element id.
- *isReadOnly* - Whether this input is read only or not. *Must be revalidated later to see if this is really required.*
- *isMandatory* - Whether this input is mandatory or not (might be different as specified in the data layer).
- *isVisible* - Whether this input is visible or not.

Design field Shows some additional information which is not data-driven.

It includes the following properties in addition to the common field (§6.5.4) settings:

- *tagName* - Defines which markup tag should be used (for example *p* or *h4*).

Form label Design field (§6.5.4) extension for form labels.

It includes the following properties in addition to the design field (§6.5.4) settings:

- *for* - Allows referencing an input field (§6.5.4).

Button Abstract representation for buttons.

A button may have the following references:

- *label* - String for the label text.

Link Abstract representation for hyperlinks.

Activator An activator is something which submits a form or start another way of interaction.

Submit button Inherits from both submit activator (§6.5.4) and button (§6.5.4).

A submit button may have the following references:

- *searchContext* - Allows referencing a search panel (§6.5.4).

The submit button type defines which kind of cancel button should be used:

- OK
- YES
- CONTINUE
- NEXT

Cancel button Inherits from both cancel activator (§6.5.4) and button (§6.5.4).
The cancel button type defines which kind of cancel button should be used:

- CANCEL
- NO
- PREVIOUS

Submit activator A submit activator may have the following references:

- *submitContext* - Allows referencing a container (§6.5.4).

Cancel activator A cancel activator may have the following references:

- *cancelContext* - Allows referencing a container (§6.5.4).

Submit link Inherits from both submit activator (§6.5.4) and link (§6.5.4).

Back link Inherits from both cancel activator (§6.5.4) and link (§6.5.4).

Layout order With layout orders you can add relations to panels to define whether they should float beside each other or not.

It includes the following properties:

- *length* - The length value.
- *lengthtype* - The length type (see below). Default is PIXELS.
- *type* - The type of layout order (see below). Default is HORIZ_LEFT.

A layout order may have the following references:

- *source* - Reference to source sub container.
- *target* - Reference to target sub container.
- *viewContext* - Reference to the owning element.

Layout order type Defines how the panels are connected to each other visually.
Can be one of the following options:

- HORIZ_LEFT
- HORIZ_RIGHT
- VERTICAL

Layout order length type Specifies the unit which is used for the length definition.
Can be one of the following options:

- PIXELS
- PERCENT
- EM
- EX
- CENTIMETER
- MILLIMETER
- INCH
- POINTS
- PICAS

Combinations and edge cases

As there is no view editor available yet this section is empty for now.

6.5.5. Workflow layer

The workflow layer is not implemented yet. At the moment there is only a mockup created by the generator which is also not completed yet though. For more information regarding this see issue #32.

It is planned to create a dedicated layer for modeling only possible workflow states and actions.

6.6. Additional notes

None yet.

7. Customisation and maintenance

7.1. Introduction

This section gives a few hints for changing arbitrary things in your generated applications while keeping in sync with upcoming releases of Zikula and ModuleStudio.

7.2. Long-term maintenance

To prevent losing the benefits of ModuleStudio (§1.2) in future you have to follow a few simple basic rules in your development process.

7.2.1. Keep consistent

When using ModuleStudio then your model is not only some sort of bootstrapping or documentation artifact. The model describes or better is the real application. Therefore you should do all important changes (like adding or moving table columns, renaming an entity or other amendments, introducing a new controller action, etc.) on model level although it might look a bit inconvenient first. Do not let your model become obsolete, as this would mean losing lots of advantages. You will thank yourself when generating again with a newer version.

7.2.2. Document your changes

Document your changes to simplify the merging process you will have to do after regeneration. For example after you added some fields later on, or you got a new generator version fixing some bugs, and so on you will want to know again where you did which changes for which reason. Usually a good place for this documentation is *modules/YourModule/docs/customisation.txt*.

7.2.3. Use overriding

All cosmetic enhancements can be done by template overriding in Zikula. Placing templates in */config/templates/YourMod/...* for example is a good idea for development.

If you need display-oriented additional logic, simply create a view plugin encapsulating your efforts in a file which is not affected by the generator at all.

7.2.4. Code additions

Perform logical enhancements in the generated implementation classes. These extend from abstract base classes containing the actual generator implementation code. So almost all concrete classes are empty waiting for your custom extension.

The file *lib/YourModule/Util/Controller.php* can be used to enable/disable specific controller actions (like view, display, ...) for particular entity types within custom conditions. See the parent class for the base implementation.

7.2.5. Use versioning

Using a version control system, like git or svn, gives you another additional level of rollback safety and is a good idea anyway.

7.3. Additional notes

Please see also this tutorial. Some parts are a little outdated due to the migration from Doctrine 1.3 to 2.1, but this shouldn't be a problem for the understandability of the overall messages shown there.

8. Other cartridges

8.1. Introduction

Beside an application it is possible to create other interesting artifacts from a given input model. This section explains what the other generator cartridges of ModuleStudio are doing.

8.2. Reporting

The *reporting* cartridge aims on generating helpful documents for the project management and application design. The following sections summarise the existing reports which are not feature complete, but act as a proof of concept for the integration of reporting technology.

8.2.1. Documentation

This document is a draft for a structural documentation of the model. It shows all essential model elements together with their most important properties. In case you added information to the generic *documentation* field of your model elements those descriptions will be shown here, too.

8.2.2. Model information

General document (used as a playground to experiment with things). Beside some statistics it shows some diagrams illustrating some measures with regards to the entities in the data layer. For example one can see which entities have the most relationships (and thus complexity to be managed).

8.2.3. Function points

This report shows a calculation of (unadjusted) function points (wikipedia). These are a measurement for the amount of complexity in a software system. It can be used as an input value for other methods like COCOMO II (wikipedia).

8.3. zOO

The *zOO* cartridge aims on creating code for a future equally-named framework. Therefore it is not of any interest yet.

8.4. Additional notes

None yet.

9. AddOns

9.1. Introduction

This chapter is about extending ModuleStudio.

There will be different types of extensions for ModuleStudio, but this page is primarily intended for future versions.

9.1.1. Language packs

At the moment only German locales are shipped with ModuleStudio. These involve all own translation strings, but not those from the Eclipse platform and used frameworks yet. It is planned to make language packs available as optional feature bundles at a later stage.

9.1.2. Figure galleries

Not implemented yet though as the modeling language itself must become matured first.

9.1.3. Template sets

Not implemented yet though as the modeling language itself must become matured first.

9.1.4. Generator cartridges

Not implemented yet though as the modeling language itself must become matured first.

9.1.5. Other extensions

As the roadmap is quite open in the long term many aspects are not decided yet. Therefore things may change significantly.

9.2. Extension Points

None yet.

9.3. Additional notes

None yet.

Part III.
Appendix

10. Glossary

Sorry, there is no comprehensive glossary here yet.

Please refer to the glossary on our homepage for now.

List of External Links

<http://modulestudio.de/en/tutorial/using-the-generator.html>
<http://modulestudio.de/en/tutorial/validation-instead-of-certification-of-3rd-party-frame.html>
<http://modulestudio.de/en/tutorial/customise-palette.html>
<http://modulestudio.de/en/tutorial/modeling-the-controllers.html>
<https://github.com/Guite/MostGenerator/issues/5>
<http://modulestudio.de/en/tutorial/creating-multiple-elements-quickly.html>
<http://help.eclipse.org/helios/topic/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Keyboard%20Accessibility.html>
<http://modulestudio.de/en/tutorial/moving-fields-with-drag-n-drop.html>
<http://modulestudio.de/en/tutorial/basic-usage.html>
<http://en.wikipedia.org/wiki/COCOMO>
<http://modulestudio.de/en/tutorial/multiple-container-elements.html>
<https://github.com/Guite/MostGenerator/issues/34>
<https://github.com/Guite/MostGenerator/issues/32>
<http://modulestudio.de/en/tutorial/structure-and-customisation-of-generated-applications.html>
<https://github.com/Guite/MostGenerator/issues/30>
<http://modulestudio.de/en/tutorial/controlling-validation-manually.html>
<https://github.com/Guite/MostGenerator/issues/31>
<http://modulestudio.de/en/tutorial/how-mdsd-reduces-costs-for-long-term-maintenance-of-c.html>
<http://modulestudio.de/en/tutorial/from-scaffolding-and-uml-to-mdsd-and-dsl.html>
<http://modulestudio.de/en/tutorial/the-first-zikula-application-in-10-minutes.html>
<http://modulestudio.de/en/tutorial/installation-on-various-platforms.html>
<http://modulestudio.de/en/tutorial/working-with-multiple-windows.html>
<http://modulestudio.de/en/product/advantages-of-modulestudio.html>
<http://modulestudio.de/en/tutorial/describing-the-model.html>
<http://modulestudio.de/en/product/what-is-modulestudio.html>
<https://github.com/Guite/MostGenerator/issues/29>
<https://github.com/Guite/MostGenerator/issues/46>
<http://modulestudio.de/mdsdglossary/>
<http://code.zikula.org/core/ticket/3138>
<http://modulestudio.de/en/tutorial/basic-settings-in-main-editor.html>
<http://www.doctrine-project.org/projects/orm/2.1/docs/en>

<http://code.zikula.org/core/ticket/2446>
<http://code.zikula.org/core/ticket/2445>
http://en.wikipedia.org/wiki/Function_point
<http://code.zikula.org/core/ticket/2444>