CaptainCasa

Structured Data Manager

Content

Overview

CaptainCasa Structured Data Manager (CC-SDM)

CC-SDM is a technology solution for quickly creating the data driven part of applications. This includes:

- Creation of data structures (table, columns) and their relations
- Generation of data access, logic and UI layers
- Creation of User Interfaces and role-based workplaces

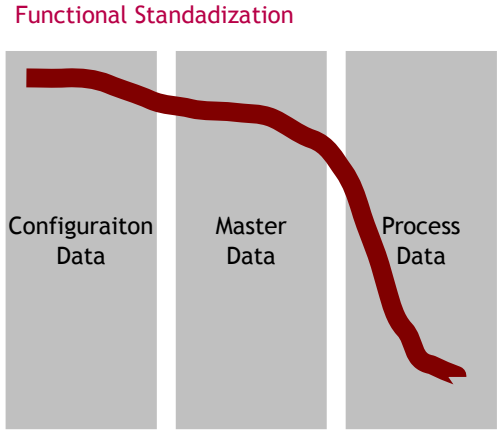
CC-SDM is highly configurable and extensible for applying modifications and extensions.

Scenarios

An application typically consists out of data definitions. These can be categorized into three aspects:

- Configuration Data
- Master Data
- Process Data

When looking on application functions around these different data aspects, then there is a clear standardization of functions in the areas of configuration data and master data. These are the typical functions like listing, creating, removing instances – with managing the consistency of data input. The underlying data structure is dominating these functions – whereas in the area of process data, it more and more is the process structure that drives the functions.



Typical Scenarios in which CC-SDM can be applied are:

- Usage of CC-SDM for covering the configuration-/master-data part of your application.
- Usage of CC-SDM for creating simple, data-driven applications, in which even the process data part is more or less a reflection of data structures.

Advantages

CC-SDM covers the boring part of applications – the part which is to a great extend defined by its data structures.

Instead of providing a “lousy user interface” for this part, CC-SDM enables you to provide a first class user interface with a set of functions that are consistently applied to all data objects.

CC-SDM automatically brings in a high level of functions around the maintenance of structured data objects:

- Single objects, objects with relations and sub-objects
- Multi tenancy
- Customize-able screen layout
- Language-dependent literal management per object
- Attachment management, image management for objects
- List reporting with flexible queries, which can be saved in private or public mode
- User-Role-based definition of input status (down to property level)
- (pessimistic) lock management

CC-SDM can be easily integrated into web applications by its RIA-user interface. CC-SDM can be a good starting point for creating applications that are based on the RIA solution of CaptainCasa (CaptainCasa Enterprise Client).

CC-SDM enables you to jump-start into your application development, by quickly providing the “80% screens” for maintaining all your application's basic objects, and allowing you to concentrate on the business process driven part of your application from early point of time on.

Technology

The technologies internally used are:

- Hibernate for accessing the data, any SQL database for persisting the data
- JSF based server layer for providing data access objects, logical objects and server-side user interface processing
- Java based rich client (CaptainCasa Enterprise Client)

Status

CC-SDM was made available as first beta version in July 2010.

Installation & Project Creation

Installation Procedure

CC-SDM is installed as extension of CaptainCasa Enterprise Client. The installation consists out of the following steps:

- Install CaptainCasa Enterprise Client, version >= 3.0 20100719
  - Install in any directory of your choice, in the following documentation we will refer to the default location "c:\EnterpriseClient"
- Install the CC-SDM Extension which is available as extension zip-file "eclntccappor.zip". ("ccappor" is the internal id of CC-SDM, that you will also find in corresponding package names etc.)
  - Unzip the content of the file into main direcotry of your CaptainCasa installation ("c:\EnterpriseClient").

That's it!

Starting

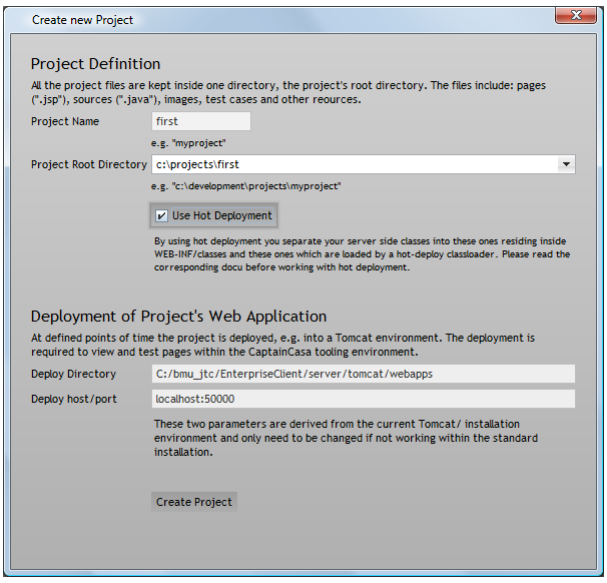
Start the following programs:

- Start the CaptainCasa Server from the CaptainCasa Start Menu ((A) Start Server)
- Start the Development Tools from the CaptainCasa Start Menu ((C) Start Development Tools)

Creation of CC-SDM Project

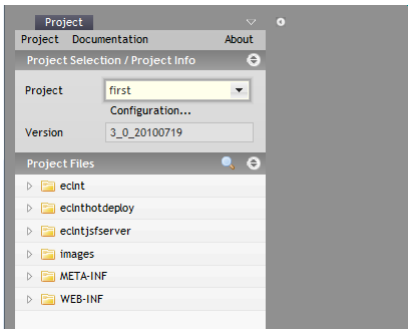
Create "normal" CaptainCasa Project

From the layout editor create a new project by selecting "Project > New...". In the dialog that is shown, specify the following values:



- The name of the project, e.g. “first”
- The location of the project, e.g. “c:\projects\first” - it is not required, but reasonable to use the project name also as name for the project directory.
- And, IMPORTANT, switch Hot Deployment to “ON”.

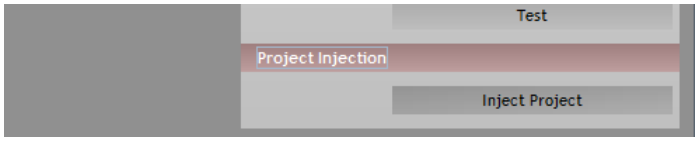
Press the “Create Project Button”, then select the project in the combo box on the left:



Inject CC-SDM into the Project

Now select the tool “Structured Data Manager” from the set of tools on the right:

This is the main working environment for working with CC-SDM. The first thing you need to do, one time only, is to inject CC-SDM into your project. Open the corresponding area on the bottom of the tool and press the “Inject Project” button:



That's it – the project is ready for CC-SDM and the tool for maintaining the project is opened.

Some further Information

- Is it required to also import the project to Eclipse? - No. The CC-SDM tool contains a code compiler on its own and does not require an IDE to do so. Of course: in case you add own code to the project – then you need to also tell Eclipse (or other IDE) about.
- Is it required to install a database? - No. CC-SDM comes with the Java-based database “Hypersonic SQL”. This is a very nice database for small(er) usage scenarios supporting a wide range of SQL standards and supporting (of course...) transactions. Of course you can connect later on to any relational database that is available via JDBC/ Hibernate.

Upgrading CC-SDM

CC.SDM comes in versions, that are upgraded regularly. There is a dependency between CC-SDM and the CaptainCasa Enterprise Client, so when upgrading, you always have to:

- download the newest version of CaptainCasa Enterprise Client
- download the newest version of CC-SDM

Before starting the installation make sure thate CaptainCasa Server (Tomcat) and CaptainCasa Development Tools are stopped.

Installation

The installation is done in the same way as the normal installation. Run the CaptainCasa Enterprise Client setup first, then unzip the CC-SDM zip file into the installation directory.

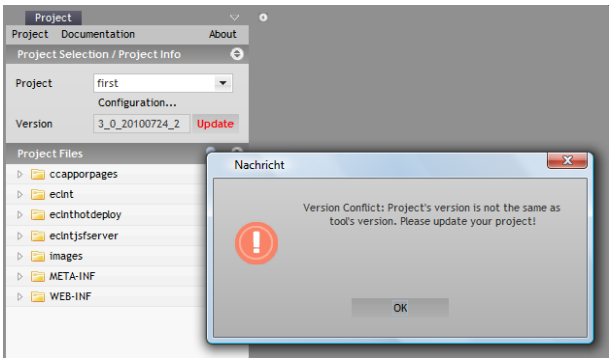
Use the same directory as with your initial installation. (The initial installation directory will be proposed automatically.)

Project Upgrade

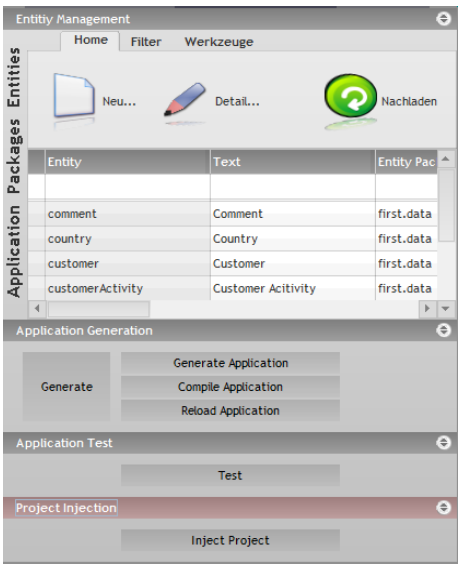
After installation start the CaptainCasa Development Tools and select the project that you created before. A popup will already warn you that your project's version is not in synch with the installation version.

The upgrade of the project is done in two steps:

- CaptainCasa update: First press the button “Upgrade” next to the project selection.



- Then open the CC-SDM Tools, open the “Project Injection” area at the bottom and re-inject CC-SDM into the project.



- Finally press the “Generate” button in order to re-generate your sources, and to re-apply them into your runtime.

Now start the CaptainCasa Server (Tomcat) – upgrading your project is finished.

Please note: it is important to do the project upgrade without CaptainCasa being started! First upgrade the project, then start the server. If not following this sequence, then the project will not properly work until you have restarted the server.

Working with CC-SDM Projects

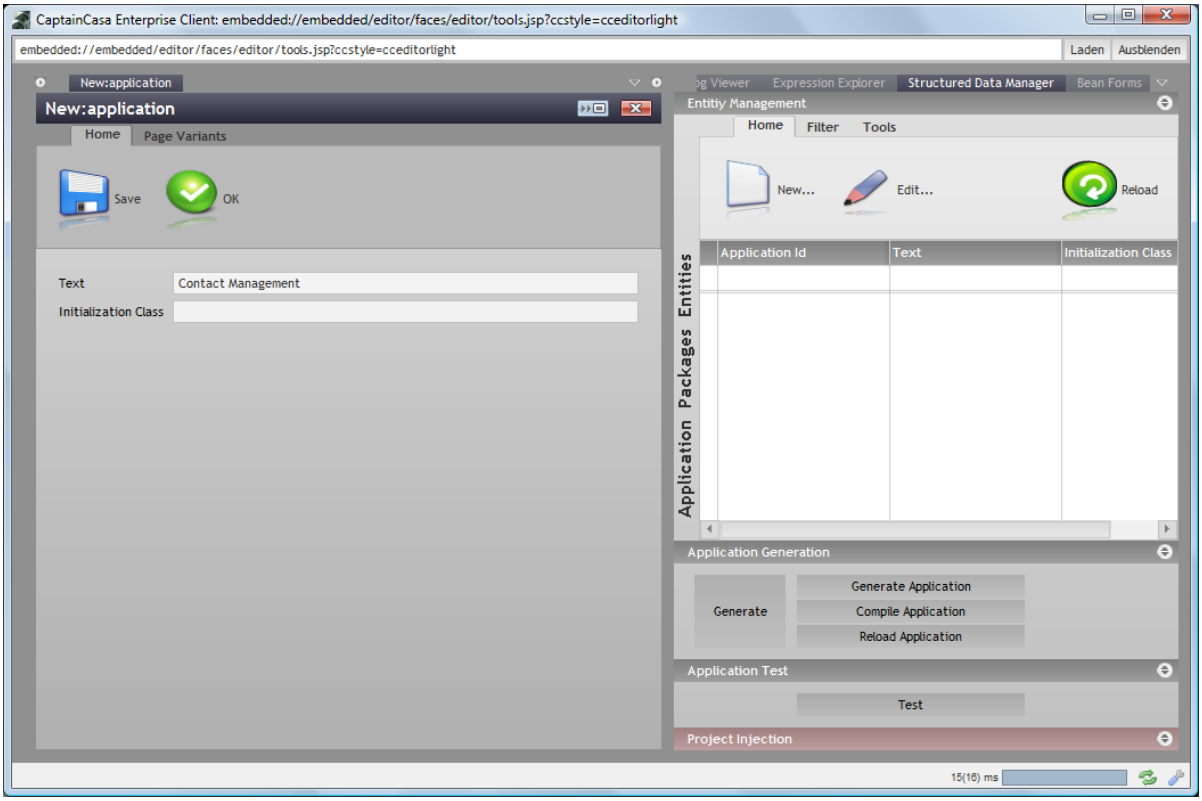
After having installed CC-SDM and created the project you can start with editing your application structure. In the following documentation we will use the example of a “Customer Management System” for demonstration purpose.

One Time Definitions

There are some definitions that you need to define one time only within the project.

Application

In the CC-SDM tool click the link “Application”:



You can maintain (exactly one) item, which is holding the core data of your application:

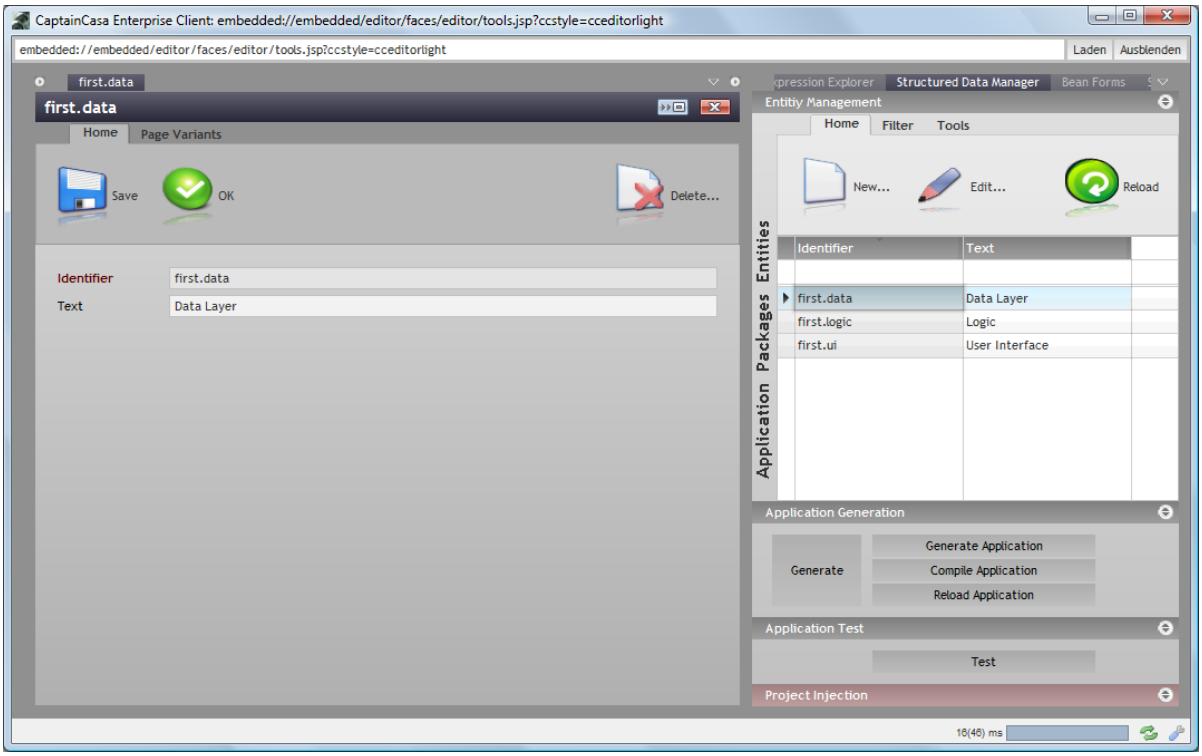
- “Text” - This is the name of your application.
- “Initialization Class” - This is the class name of a class that internally registers interfaces implementations so that they can be reached by CC-SDM. CC-SDM provides various extension points, each one available through a defined interface. - You only need to pay attention to this attribute if coding extensions. Leave this attribute to blank otherwise.

Package

In the CC-SDM tool click the link “Packages”. Here you define the Java package names, that will be used for generating code. The principle idea behind CC-SDM is to create simple, understandable, extensible code for three layers:

- the data layer
- the logic layer
- the UI layer

Per entity you explicitly assign package names for these layers later on.



Per Package you need to define:

- “Identifier” - this is the package name. Use the normal Java package naming rules here: packages are structured hierarchically, each level separated from the other one by “.”. Each package level typically starts with a lowercase character.
- “Text” - some text behind the package.

In the example we used the packages “first.data”, “first.logic” and “first.ui”.

It's important to point out, that the packages do not have to pre-exist somewhere. You just can specify “what you want” during the package definition – it's just defining the folder structure of what is generated later on. - In case you are not interested in the generation at all you may define one central package and relate all the package references later on to this one package.

Entity Definitions

Having defined the application and package settings, you now are ready to define entities.

Entity relates to Table

An entity is related to exactly one data table. Editing an entity is very similar to creating a table in other tools, i.e. you have to define columns, data types and information about the primary key.

Entities can be linked in two ways:

- “link-way” - You may define foreign key relation ships between tables.
- “embed-way” - You may define that one entity is embedded into another entity

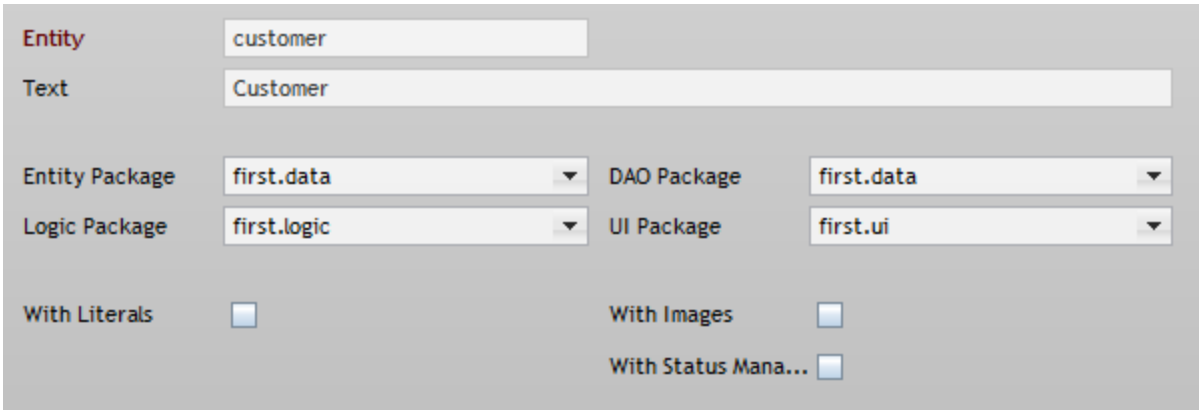
Within the the “link way” (e.g. entity “contact” contains a link to entity “contactType”), both entities are stand alone entities. - Within the “embed way” (e.g. entity “contact” embeds an entity “address”) the embedded entity does only exist within the context of its owning entity.

Creating an Entity, basic Definitions

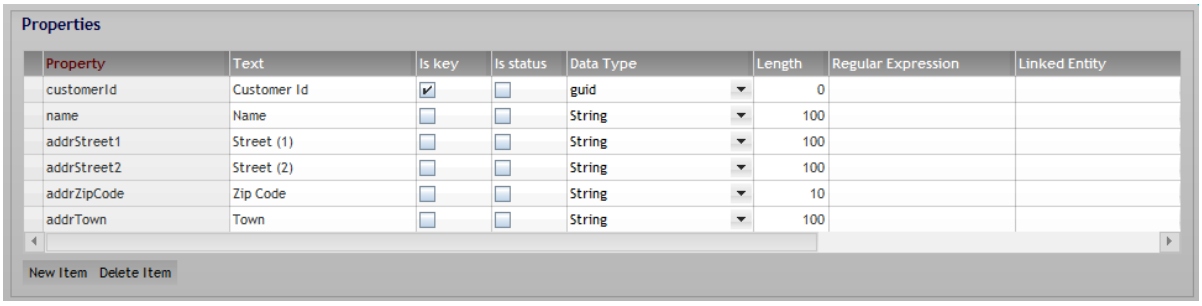
When creating an entity, there are some header parameters that you have to define:

- “Id” - the id of the entity
- “Text” - Some text for the entity
- “Package-Assignments” - For the levels “data”, “logic” and “ui” you need to assign the packages. Packages need to be created before.

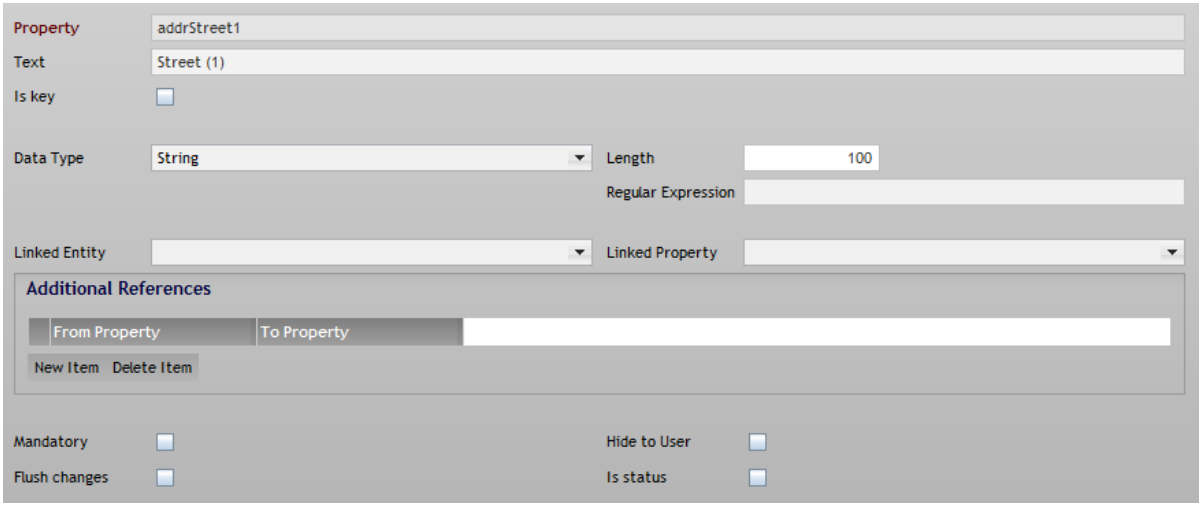
In addition there are some flags (“with Literals” etc.) which will be explained later on.



Then, the most important part of an entity definition is the list of properties (“columns”):



You can edit the property details by double clicking into a property line (or pressing return within the line):



The attributes you need to define per entity are:

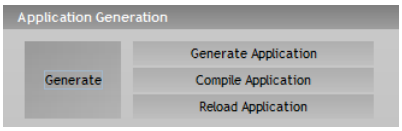
- “Property” - the id of the property
- “Text” - some text
- “Is Key” - flag, that this property is part of the primary key of the entity. A primary key may consist out of several properties. Important: once having generated the application (and the table structures) primary keys only can be changed by dropping the corresponding table from the database before.
- “Data Type” - the data type of the property. There's a list of data types available, please open the corresponding value help.
- “Length” - in case of using “String” as data type, this is the length of the string.
- “Regular Expression” - a regular expression that is used for checking the user input. Regular Expressions are a defined syntax for describing a String-format, e.g. the regular expression “[a-z]{1}[a-zA-Z0-9.]*” defines the first character of a string to be out of the “a-z”-range, and the subsequent ones to be out of the range “a-zA-Z0-9.”.
- “Linked Entity”, “Linked Property”, “Additional References” - In case the property references to an other entity (i.e. foreign key relationship) you define the entity to link to, and the property in the linked entity to link to.
- “Mandatory” - the user need to define this property
- “Hide to User” - the property is hidden, e.g. because it is an internal data property
- “Flush Changes” - in case of applying own Java-rules when maintaining the entity at runtime, it may be useful to directly send the value of the property to the server side processing after user input.
- “Is status” - the property is a status property – there is a certain status management which is part of CC-SDM, please refer to the corresponding chapter of this documentation.

When maintaining the properties of the entity, make sure that at least one property is defined as key-property.

Generating and Testing the Application

Generation

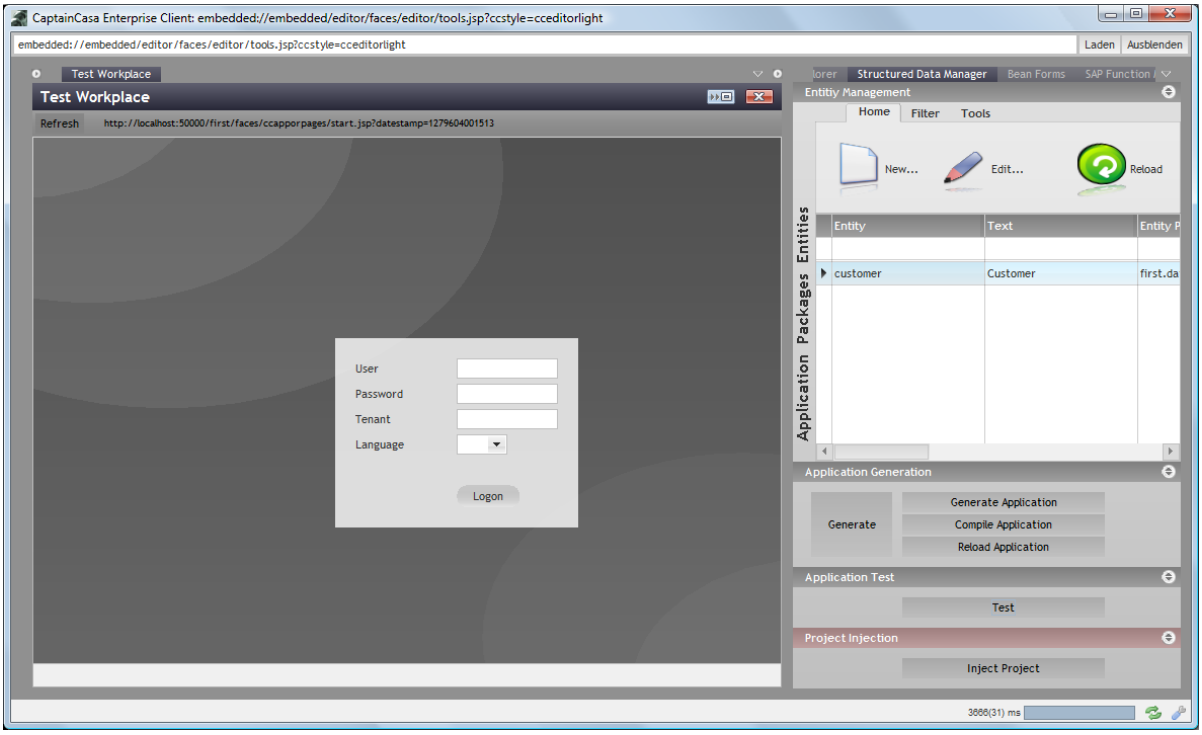
After having defined and saved the entity, press the “Generate” button:



The generation internally includes three phases: the generation of source code into the packages defined, the compilation of the source code and the internal deployment to the Tomcat server that is part of the CaptainCasa Enterprise Client. You can also trigger each of these phases by the corresponding button – which makes sometimes makes sense when adding Java rules.

Testing

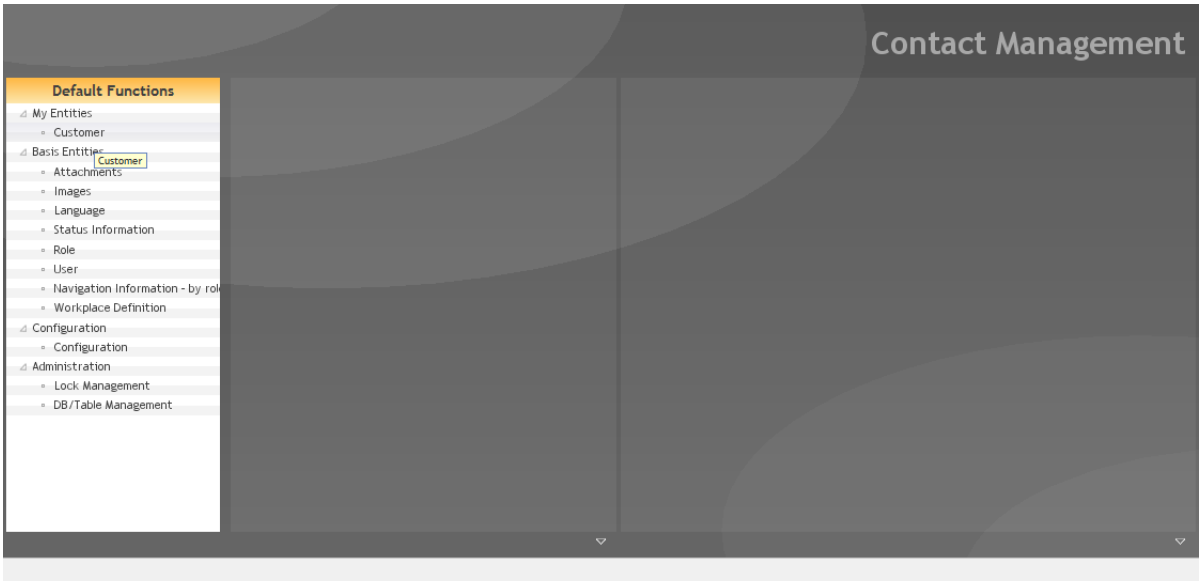
After Generation you can test. Test is most simply done by pressing the “Test” button:



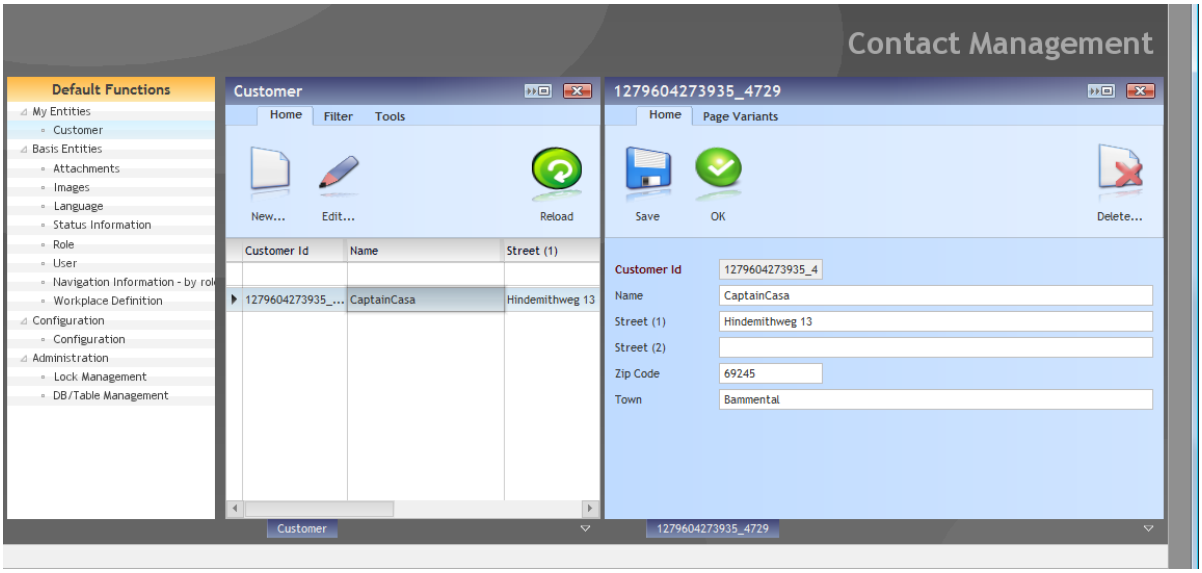
This is the logon screen. For initial usage define the following values:

- “User” - “initial”
- “Password” - “initial”
- “Tenant” - Any tenant/instance id, e.g. “test”
- “Language” - “de”/”en”, if nothing is defined, then the server's default language will be applied

After logon you see the following screen:



The entity “Customer” that was defined before is part of the menu of the left. You can double click the item and start the maintenance of data:



You see: the full application – from database definition up to the user interface is available without any step of coding in between. All meta data defined with the application definition is transferred into the application processing correspondingly.

Starting in the Browser

You can open the application within your browser by simply opening url:

```
http://<host>:<port>/<project>/index.html   ==> Applet way
http://<host>:<port>/<project>/index.jnlp   ==> Webstartway

Example: http://localhost:50000/first/index.html
```

Defining Entity Relations

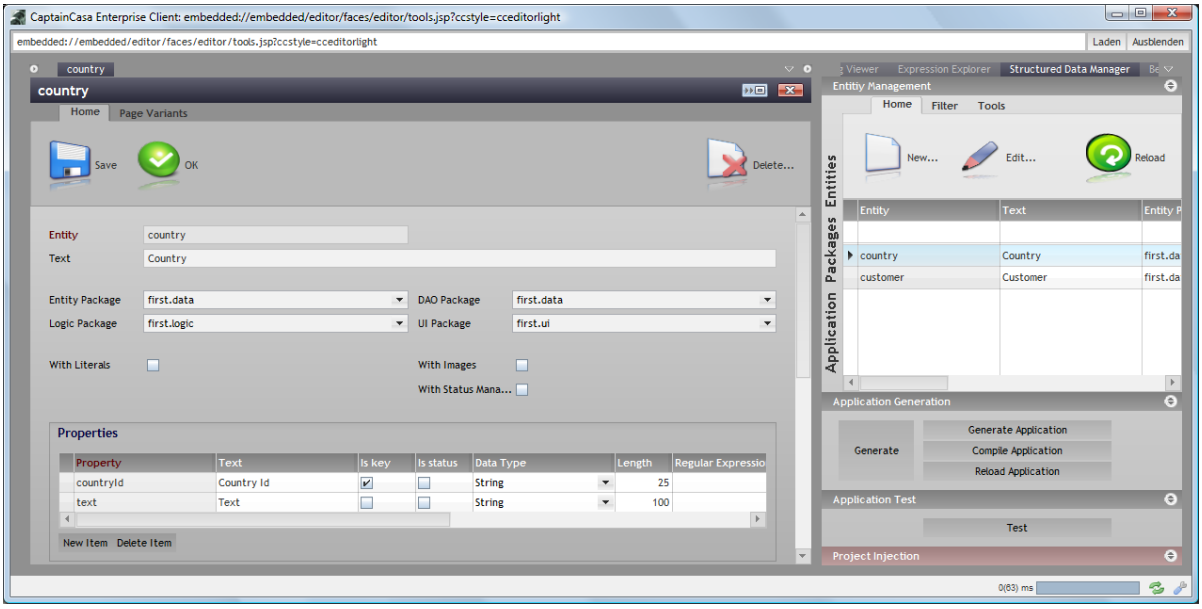
There are two types of relations:

- “link” relations - in this case one property of one entity references to a key property of another entity. Both entities are visible entities to the user. Link relations represent foreign key relation ships between tables.
- “embedded” relations – in this case one entity instance is owning other instances. There are tow sub-types of embedded relations:
  - 1:1 embedded relations, so called “extension aspects”: in this case one entity instance owns a second entity instance, which add extended information
  - 1:n embedded relations, so called “sub aspects”: in this case one entity instance owns a list of other entity instances

Link Relations between Entities

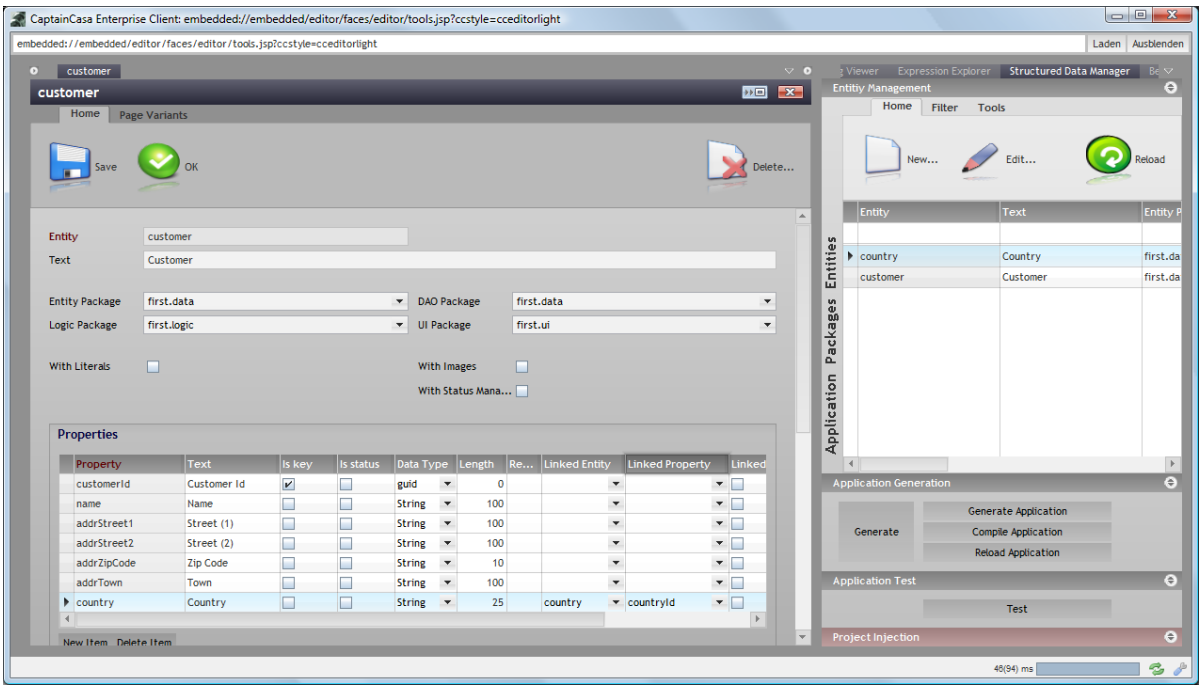
In the example “Customer Management System” we want to add a country field to the customer, the country field should contain a key of the country entity.

So: first we create the country entity:



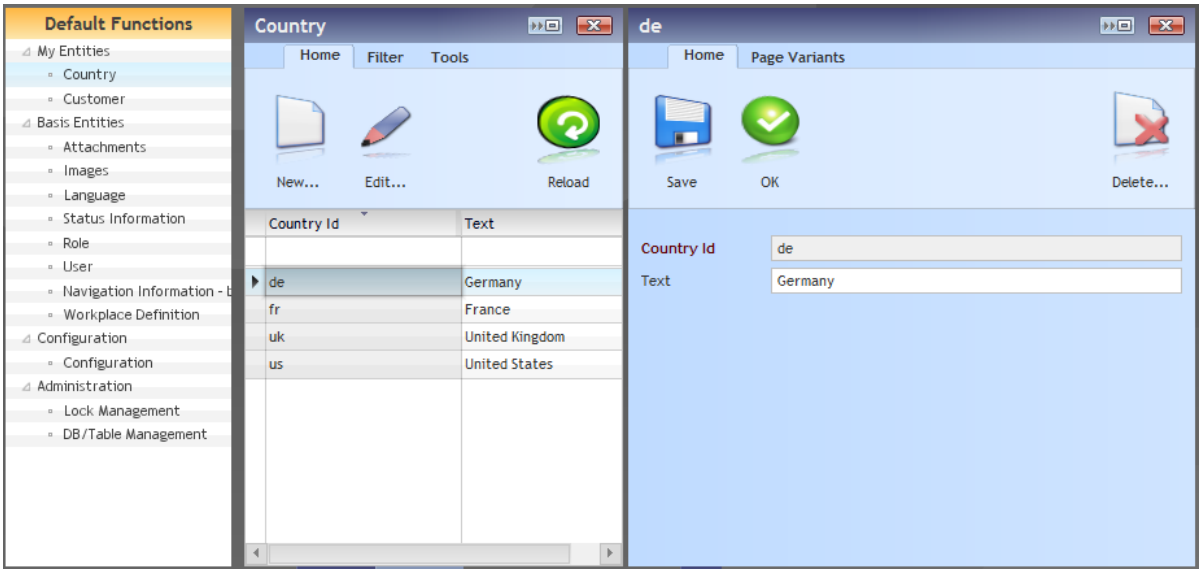
It provides a single property “countryId” as key.

Now, in the customer entity we define a new property “country”:

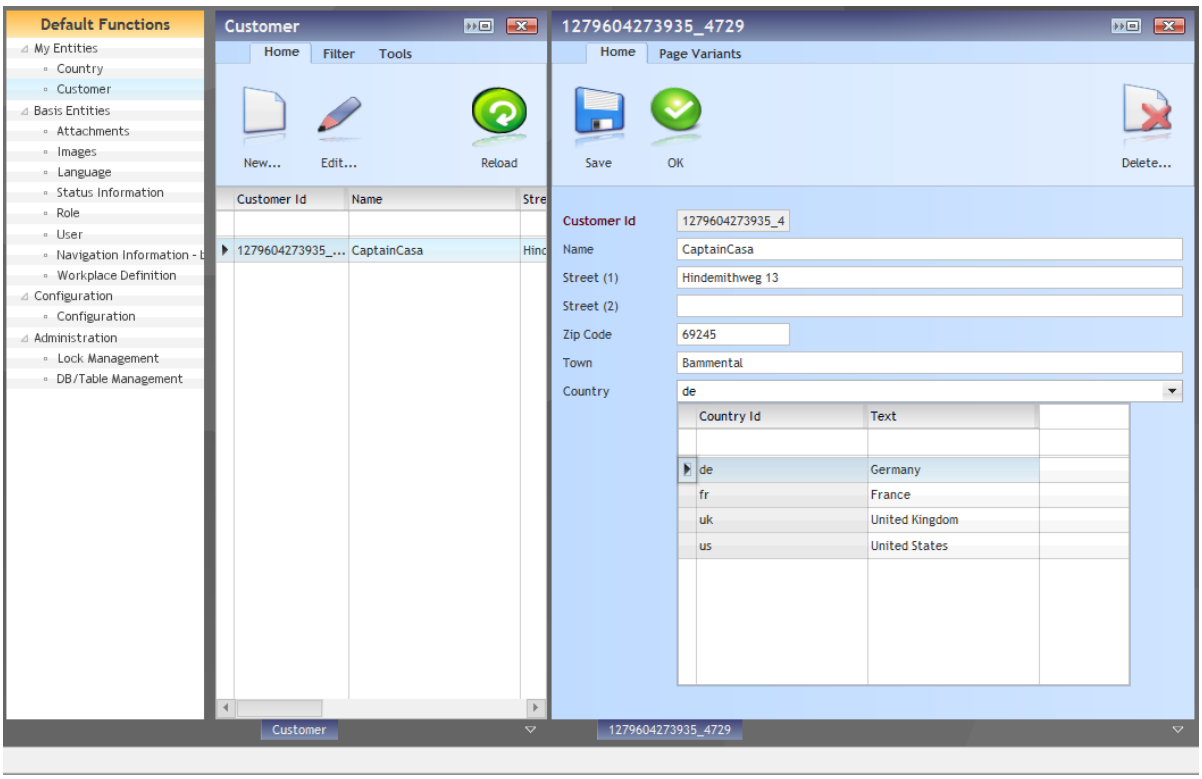


The “country” property has the same data type as “country-countryId”, and it points via the attributes “Linked Entity” and “Linked Property” to the country entity.

After generating (“Generate” button) and testing (“Test” button), you now can maintain countries...



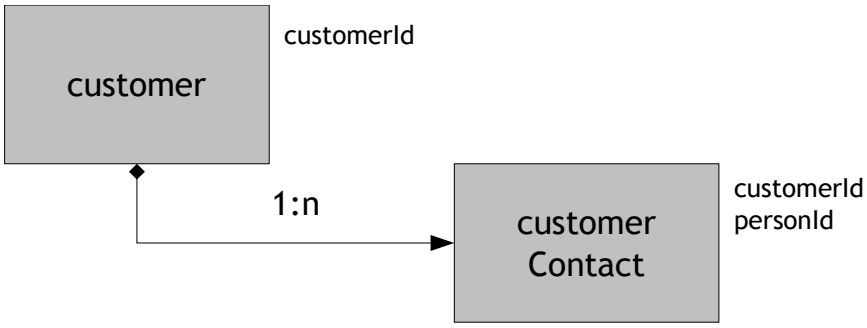
...and now can assign the country entities to your customer data:



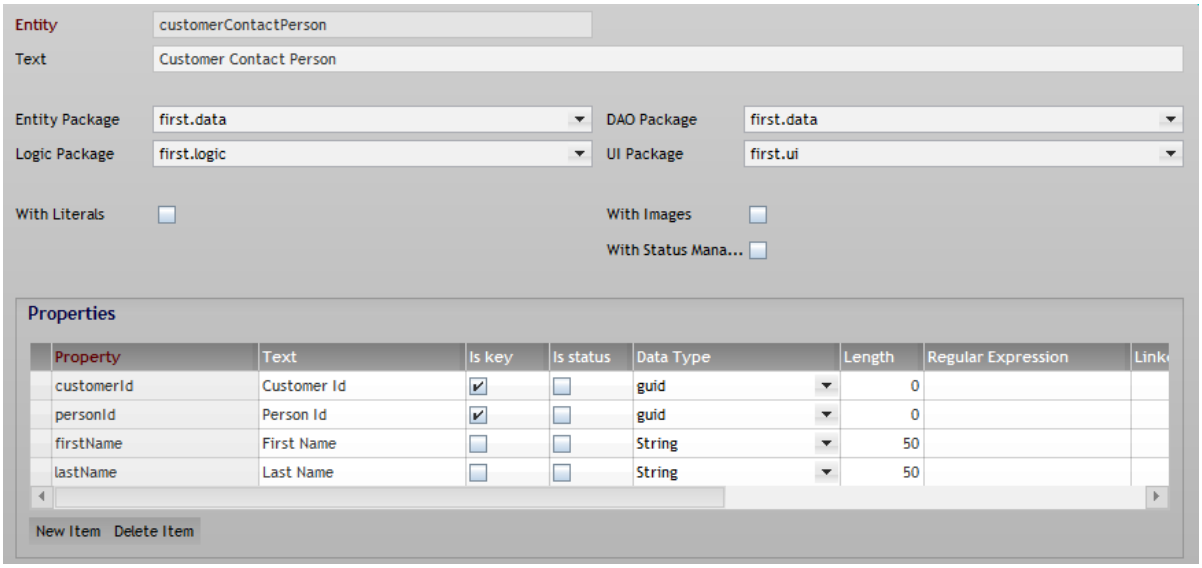
The cusomter-country field automatically is provided as selectable field, you can either enter the id of the country directly via keyboard or your can pick from the list.

Embedded 1:n Relations - “Sub Aspects”

Imagine you want to hold contact persons for each customer. The contact persons should not be maintained as individual instances but should be a sub aspect of the customer, i.e. you only define them in the context of one customer.



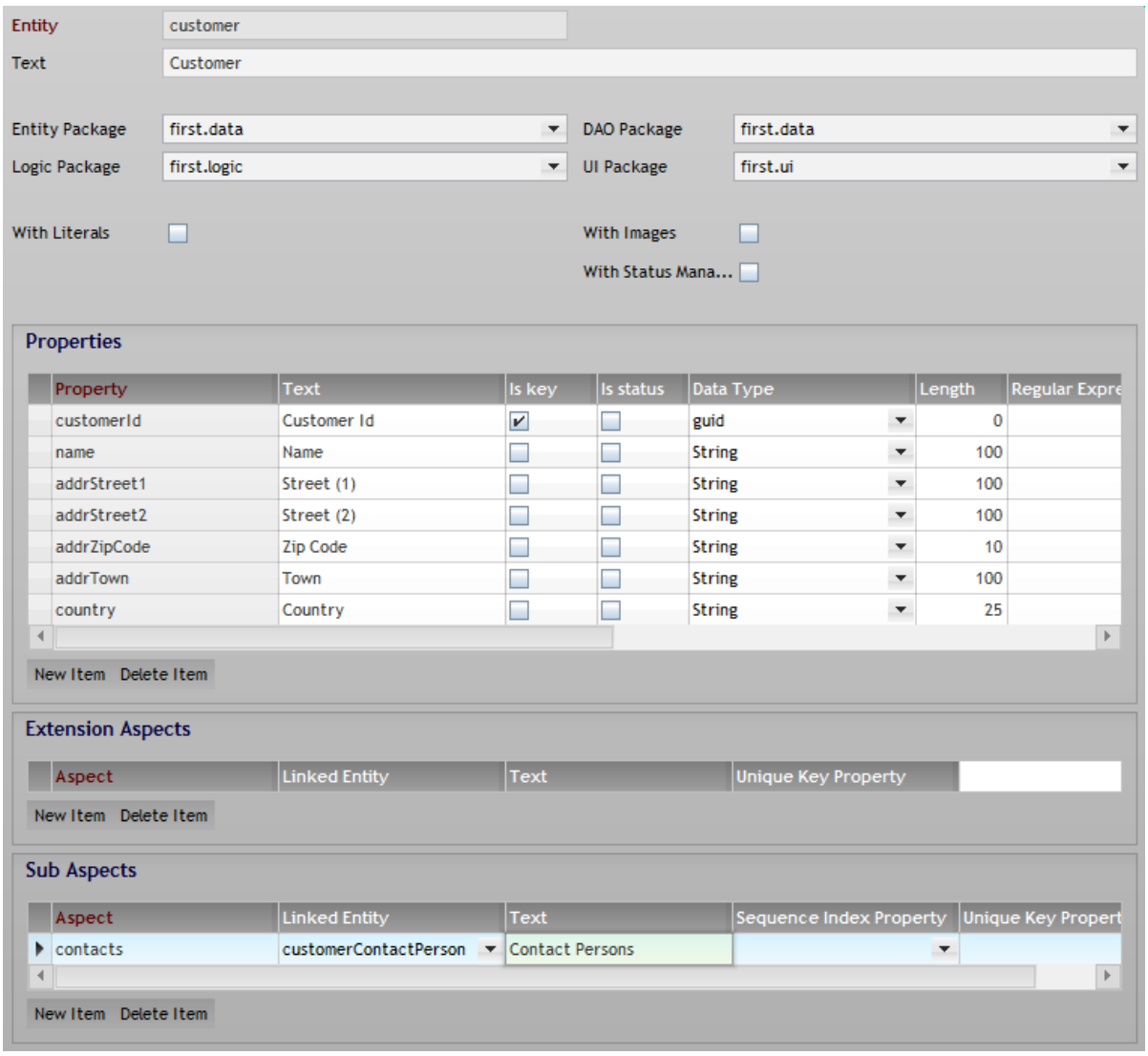
First, we create a corresponding entity “customerContactPerson”:



You see: because the contact-person belongs to the customer you have to take over the key of the customer. This is a must! - The “customerId” of the key definition must be exactly the same definition as made in the customer-entity. It's for example not allowed to name this property e.g. “customerid” (lowercase “i”)!

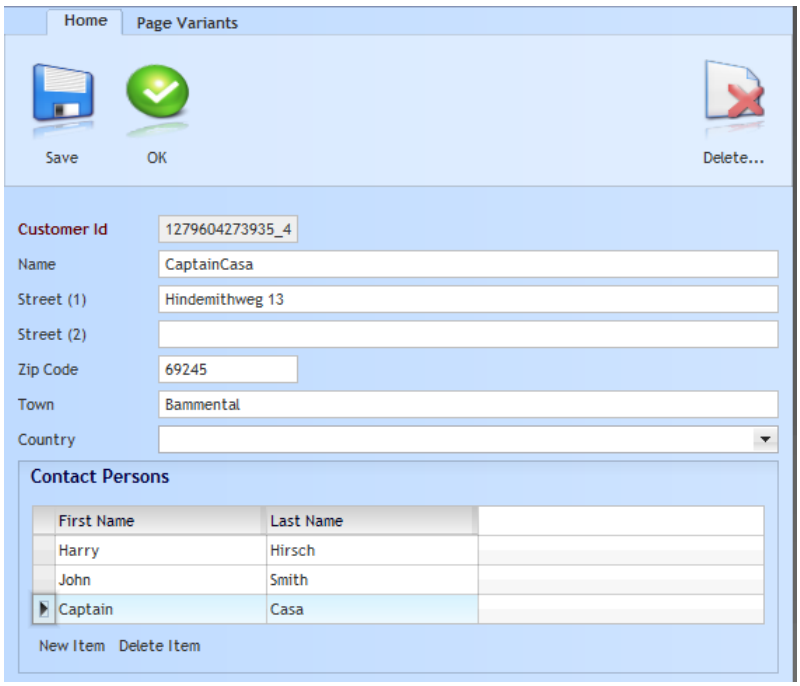
Because it's a one-many relationship, there needs to be an additional key property. In our case we defined “personId”, and also assign data type “guid”. You, of course, can also assign other data types! - Because in our case a “guid” is not really relevant for showing to the user, we decide to set the “hide” flag for this property as well.

Now, we have to tell the customer-entity about, that it owns a “sub aspect”. For this purpose we add a corresponding item in the section “Sub Aspects”.



You see: the Contact Persons were added as “contacts” aspect into the customer-entity.

After generating and testing the customer maintenance will now look the following way:



The sub aspect was added as grid. You can navigate into a detail screen per item by double clicking the item.

Defining a List Sequence

By default the embedded items of one entity are read from database and are sorted either by their primary key or by the explicit sort sequence, defined with the entity definition.

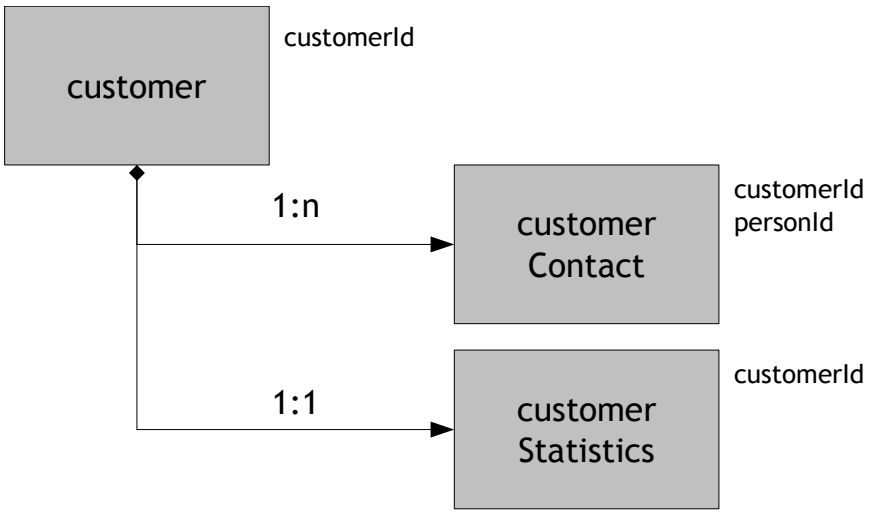
There are (many) cases in which you want a certain sequence to be defined by the user, so that the list on the screen directly reflects the sequence of the elements in the database. (Btw.: you can drag & drop items within a grid in order to rearrange their order.)

A sequence needs to be defined as part of the data of the emebdded entity – it needs to provide a sequence field- Typically the sequence is an integer field. In the definition of the Sub Aspect there is the possibility to define this “Sequence Index Property”. As result of defining this attribute the CC-SDM logic will automatically assign sequence numbers to each of the lines (first line: 0, second line: 1, etc.) before saving,

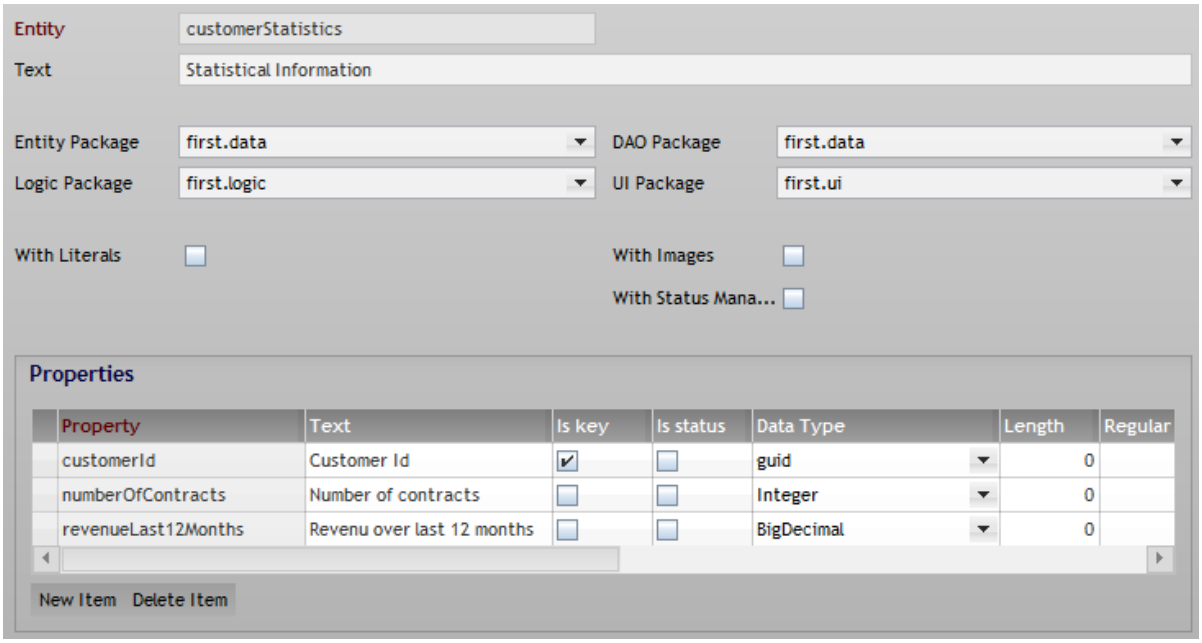
1:1 Embedded Relations - “Extension Aspects”

Sometimes you want to arrange certain properties of an entity into a segment (table) on its own. In this case there is exactly one additional information that is plugged to the owning entity.

Example: in the “Customer Management System” we want to add statistic information to the customer, as optional, additional data segment.

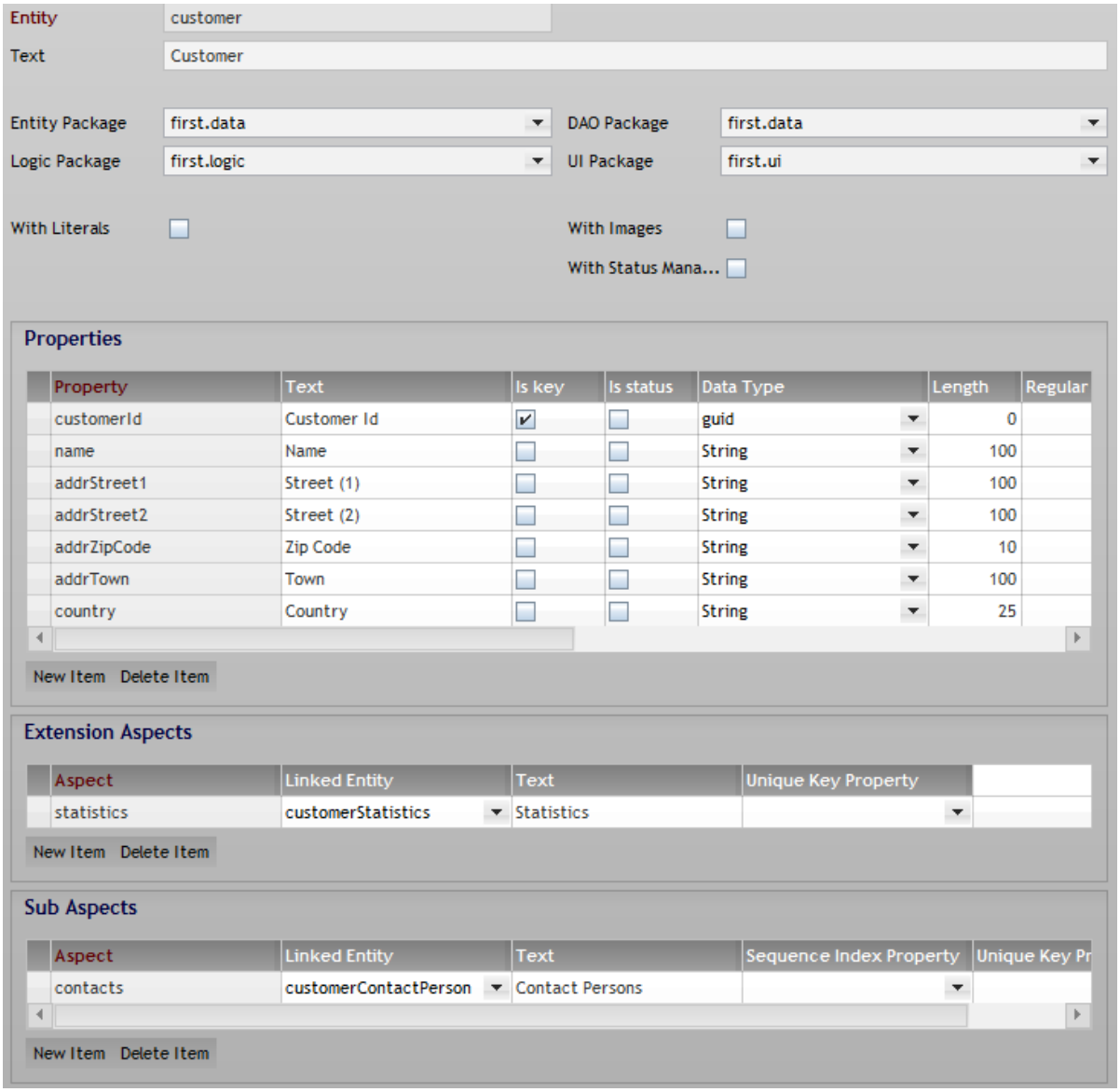


The entity we define is the entity “customerStatistics”:

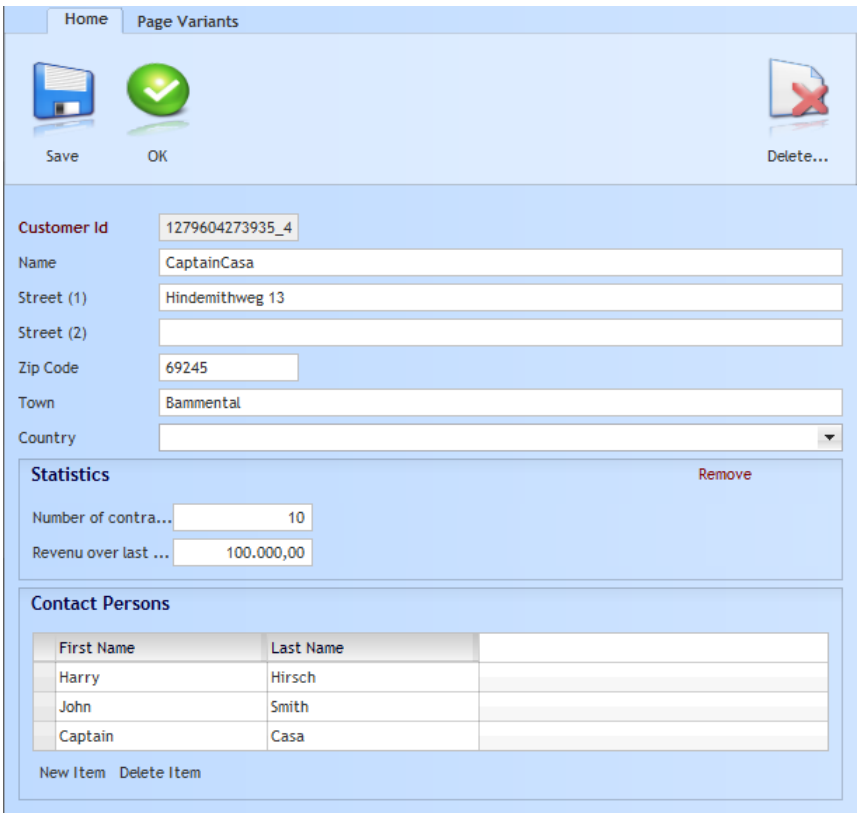


You see: the key of this entity is exactly the same as the key of the customer. Again, this is a must!

In the customer-entity we now define the extension aspect by adding a corresponding item within the ara “Extension Aspects”:



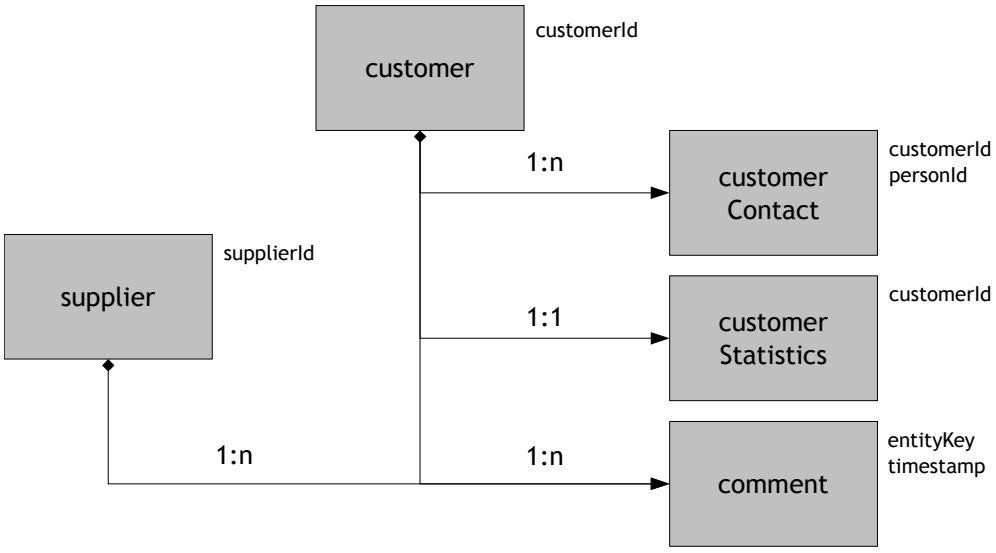
The resulting maintenance screen is:



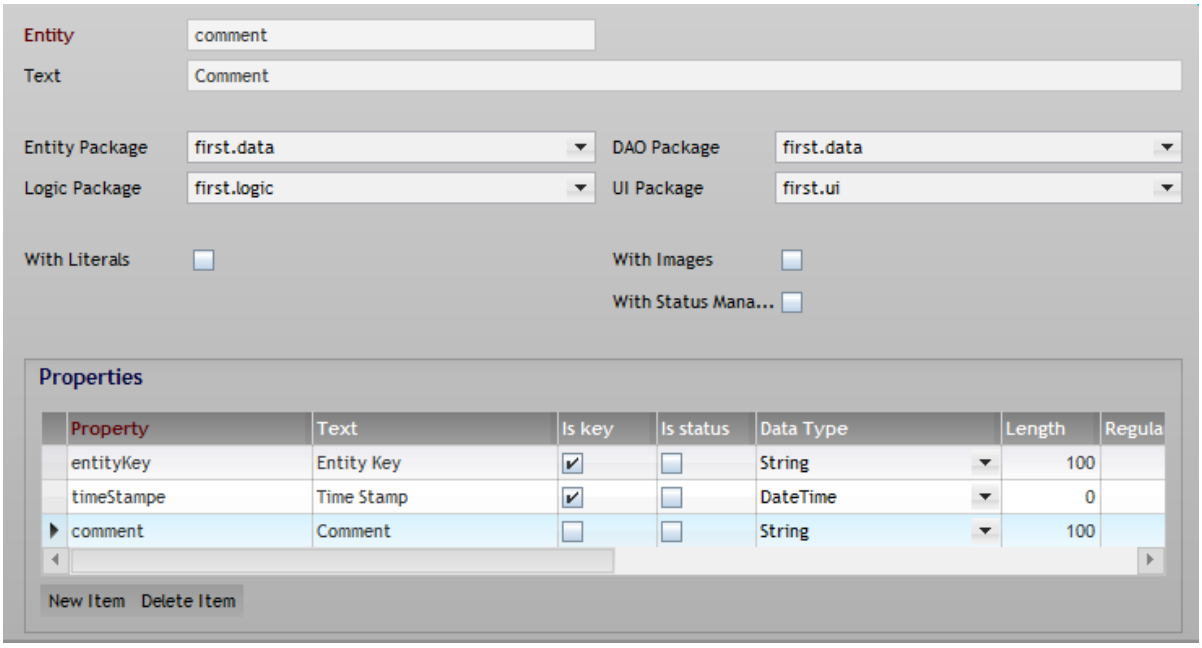
Re-Usable, embedded Relationships

In the previous chapters embedded entities were part of one owning entity. As result the key of the owning entity was directly taken over into the key of the embedded entity.

There are cases, as well in which you want to define entities that are not bound to one owning entity. Example: you want to keep comments for an entity. If the entity is a “customer” or a “supplier” (or whatever) should not be relevant – the comments should be apply-able to any entity.



The entity “comment” is built in the following way:

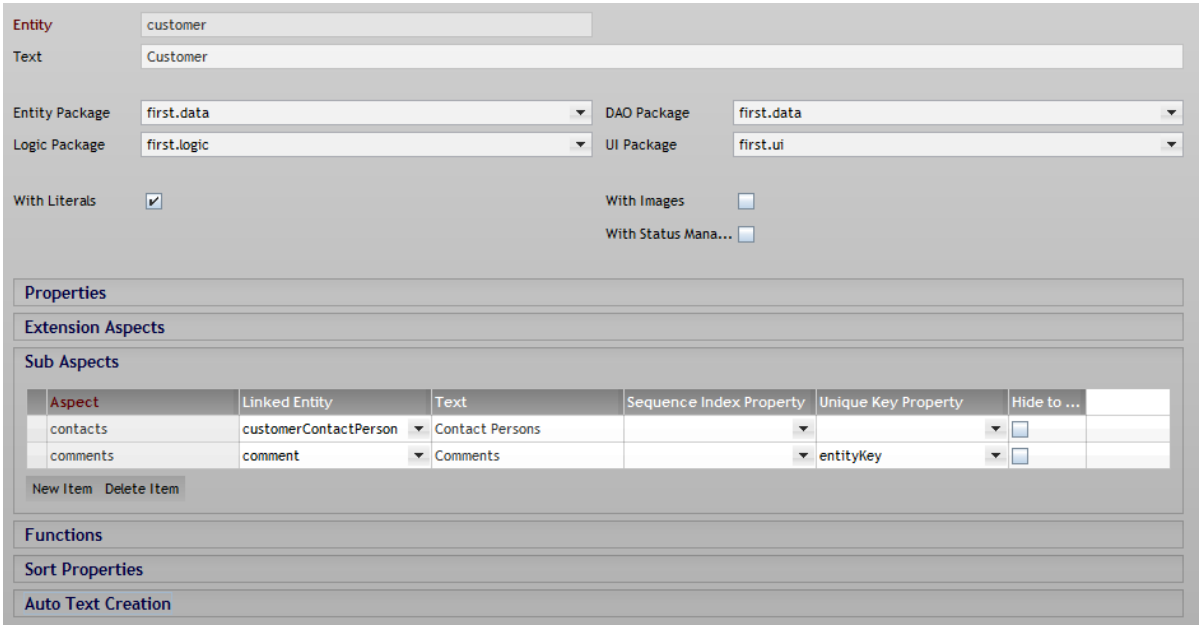


The key contains a long String attribute, name “entityKey” in this example.

Each entity instance - whether it's a customer-entity or a supplier-entity holds a unique id. This id consists out of the concatenation of all primary key attributes and the name of the entity. Example: “customer:395252635”.

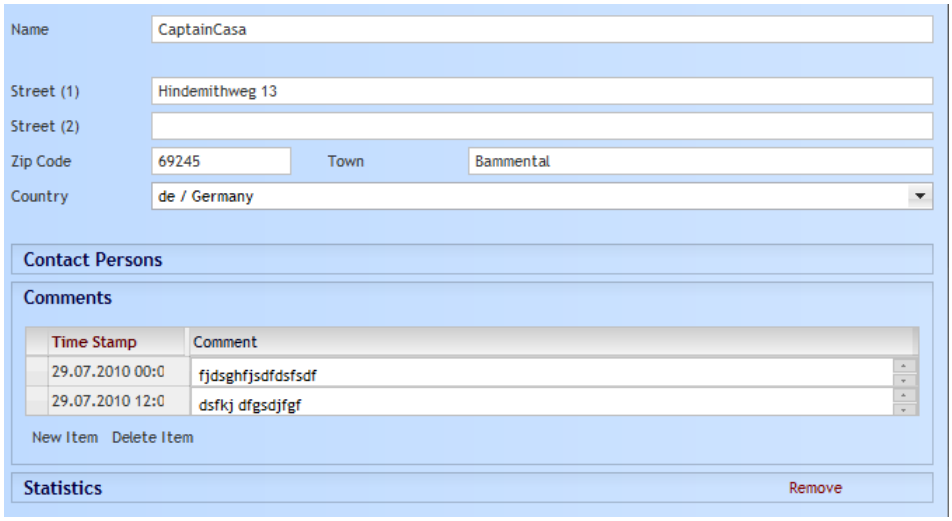
The comment-entity takes over this unique id into its column “entityKey” - so that it now can server multiple owning entity types.

Within the entity definition of the customer-entity, the sub aspect is defined in the following way:



In the definition of the comment-sub-aspect the attribute ”Unique Key Property” is defined: this definition tells CC-SDM to write the unique id of the entity (in this case: the customer-entity) into the “entityKey”-attribute of the comment-entity.

The screen result of this sub-aspect is the same as with “normal” sub-aspects:



We now took a look onto “re-usable, embedded 1:n sub-aspects” - the same is possible when building “re-usable , embedded 1:1 extension-aspects”, with the same concept in mind. Now of course the only key that the embedded aspects need to provide is the entity-key, that holds the unique id of the owning entity.

Built-In Concepts

There are certain issues, which are always “the same” when defining structured data. For these issues CC-SDM provides pre-defined frameworks.

Literal Management

Overview

An entity instance is technically defined via a corresponding key that you define as part of the entity definition. But: the key typically is not the best level to present an entity instance to the user – as result the entity instance needs to be associated with a certain name or text, so that the user always sees the id of the entity and its text.

There are several issues:

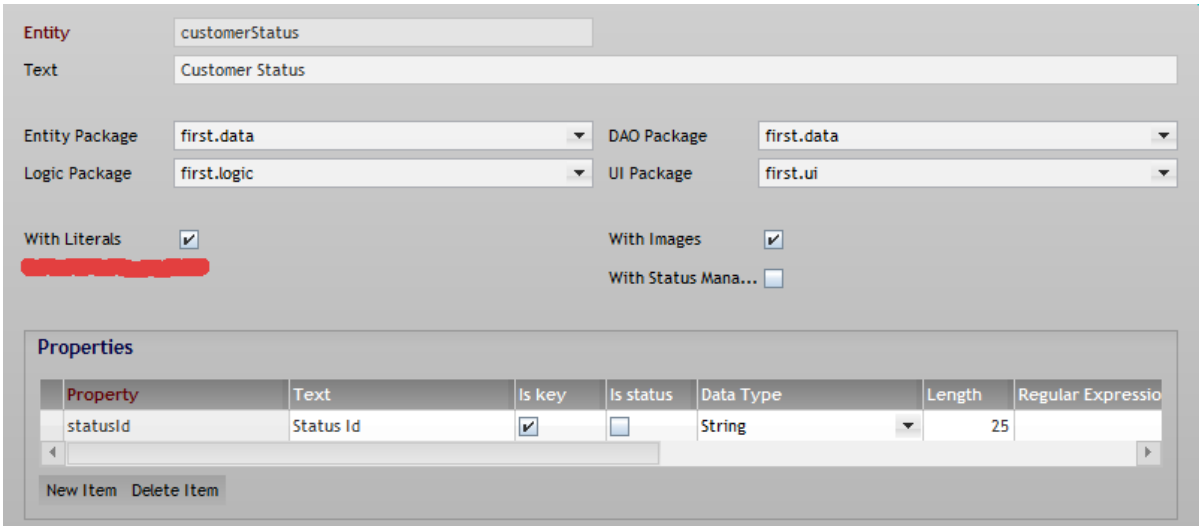
- The text typically depends from the language, so there are several texts for the same entity instance, each in a different language.
- The text may be implicitly defined: for example you edit a person with attributes “firstName” and “lastName” - the text associated with the person should be “firstName, lastName”.

CC-SDM comes with a powerful literal management that can be very simply applied and that is consistently used through all screens of CC-SDM.

Defining that an explicit Text is associated with an Entity

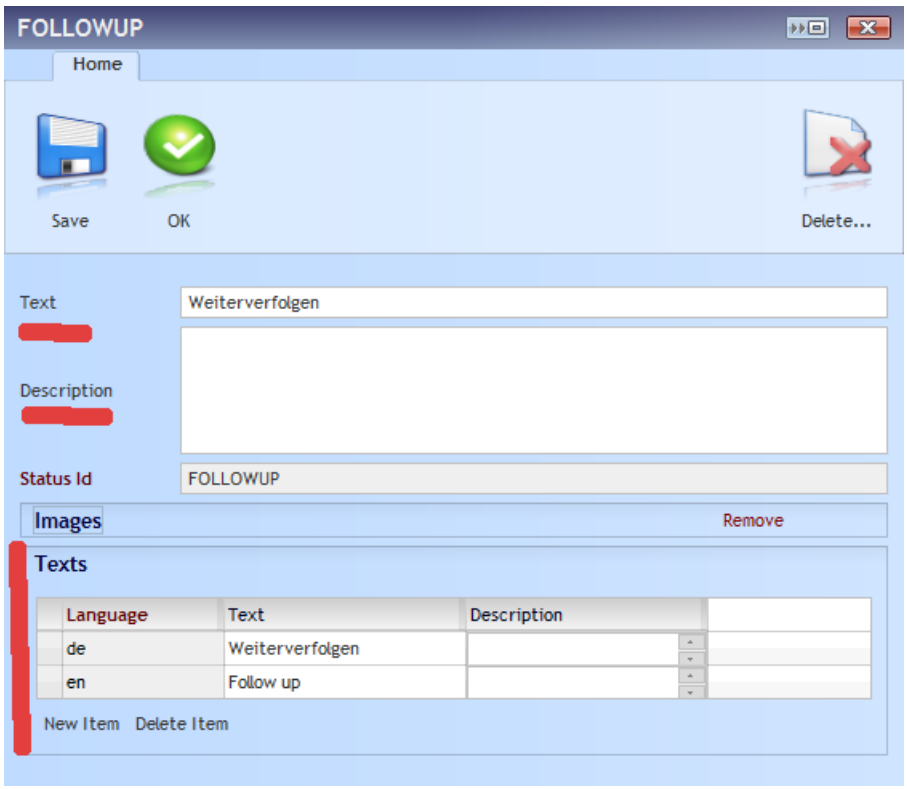
This is the typical case: an entity has a text, the text needs to be defined in a multi-language way.

Example: in our “customer management system” we want to assign a status to each customer, indicating if the sales process with this customer is still active or was cancelled etc.



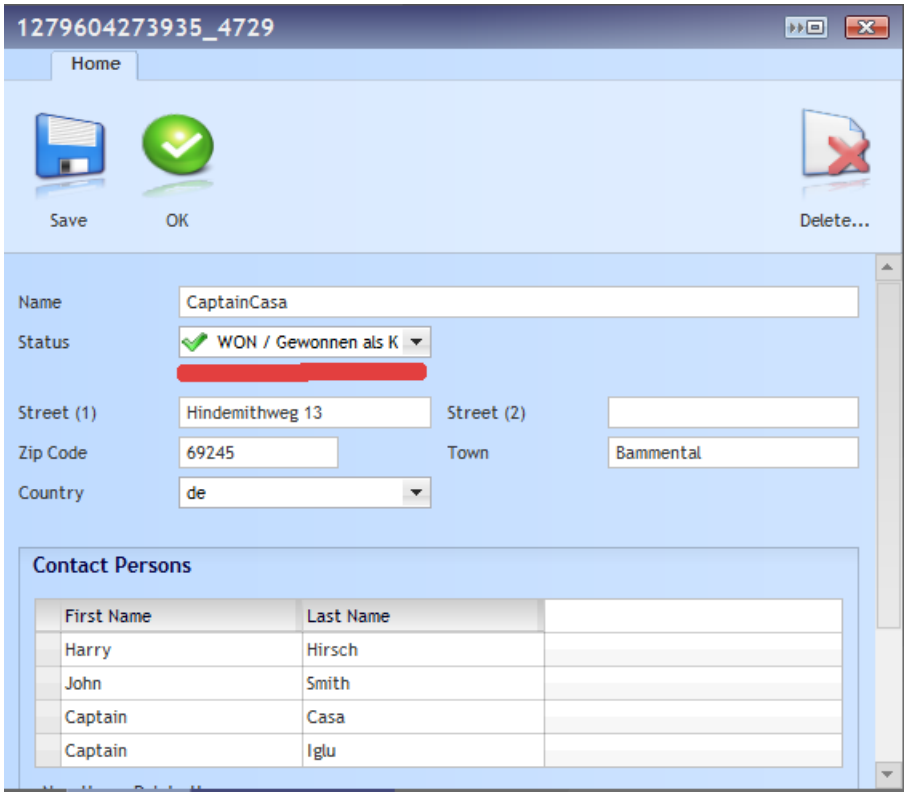
The entity only has one property definition – the “statusId”. And the checkbox “with Litrals” is defined to be “true”. That's all you have to do!

After generation you will see the following screen for maintaining status instances:



In the normal attribute area you see two new properties: text and description. In the bottom area you see the same properties, this time arranged as table, with the language key as additional language. When updating the top fields, then automatically the bottom grid will be updated, according to the language the user is logged on.

After registering the entity “customerStatus” to be a linked entity of “customer” (foreign key relationship), the customer screen now shows id and text of the assigned status:

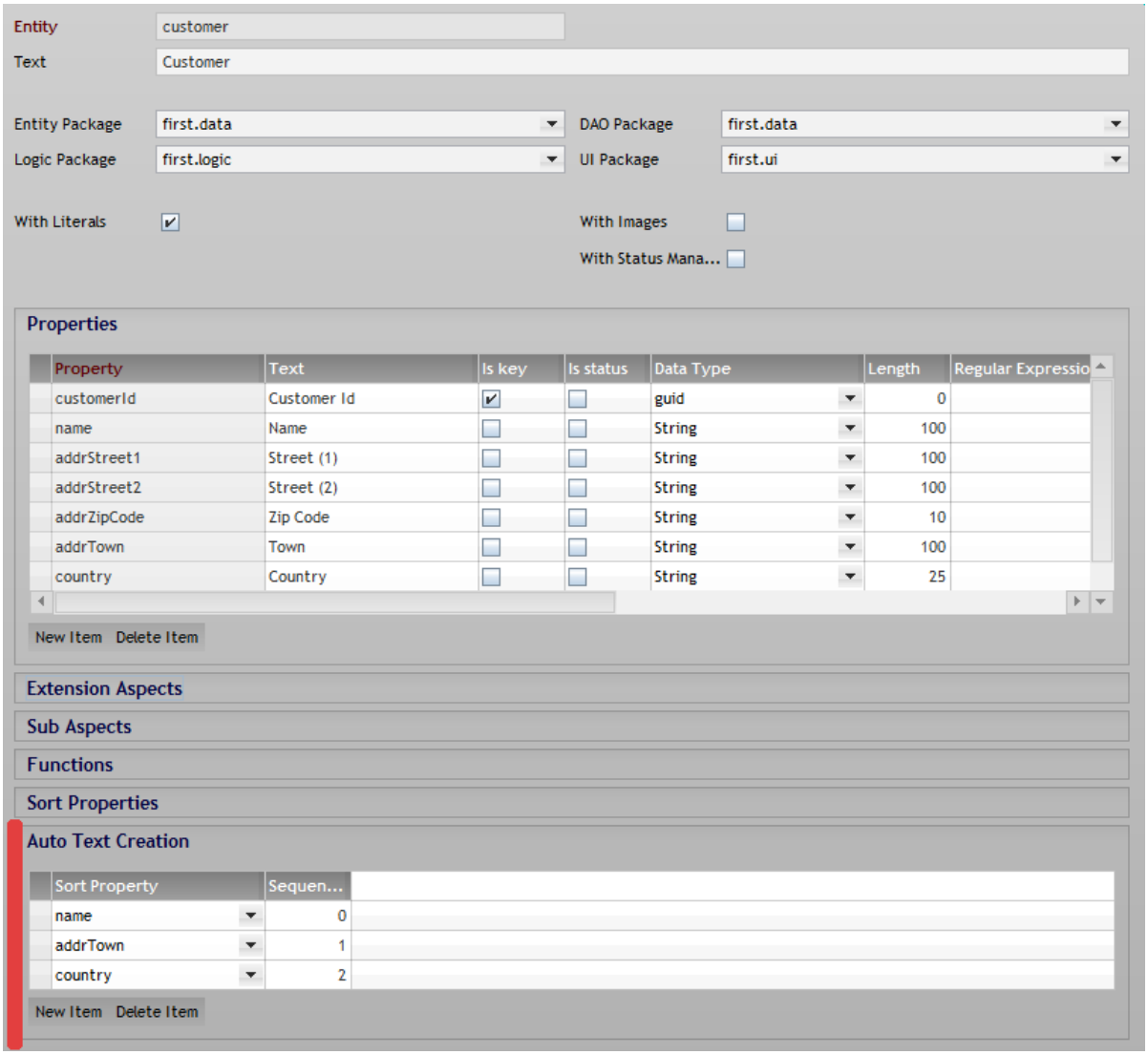


Automatically deriving the Text from other Properties

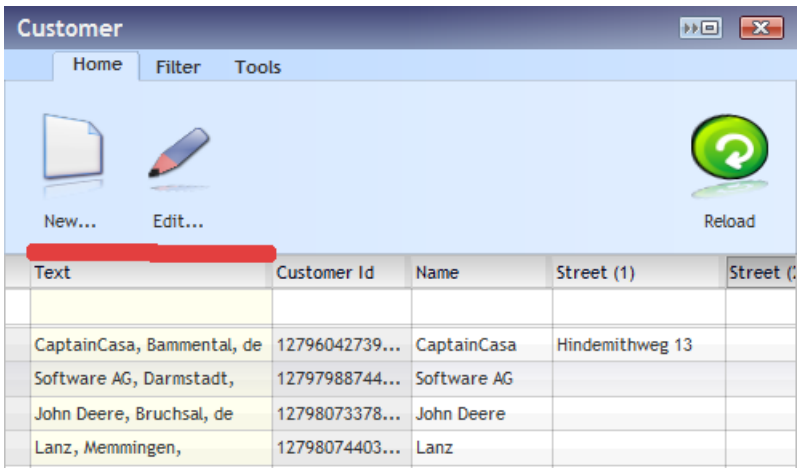
Sometimes you do not want to define an explicit text for an entity, but you want to use existing properties of the entity to form the text. Example: it does not make sense to define a text for a person – the text should be derived from the person's last and first name.

You do so by maintaining the section “Auto Text Creation” of an entity definition.

Example: for the entity “customer”, we want a text to be maintained. The text should contain the name of the customer, the town and the country. The definition is:



As with normal texts, the automatically derived text will be shown everywhere, where a customer is referenced:



Generating the text automatically is quite powerful feature – that you can/should apply to all entities that do not have a "self-explaining" key.

Detail Information about how literals are stored

All literals are stored within one basis entity - "cclmLiterals". The corresponding table of the entity looks as follows:

Language	Entity Key	Text
ALL	country:us	United States
ALL	customer:1279604273935_4729	CaptainCasa, Bammental, de
ALL	customer:1279798874425_8760	Software AG, Darmstadt, de
ALL	customer:1279807337880_4077	John Deere, Bruchsal, de
ALL	customer:1279807440370_3851	Lanz, Memmingen,

The key of the table is:

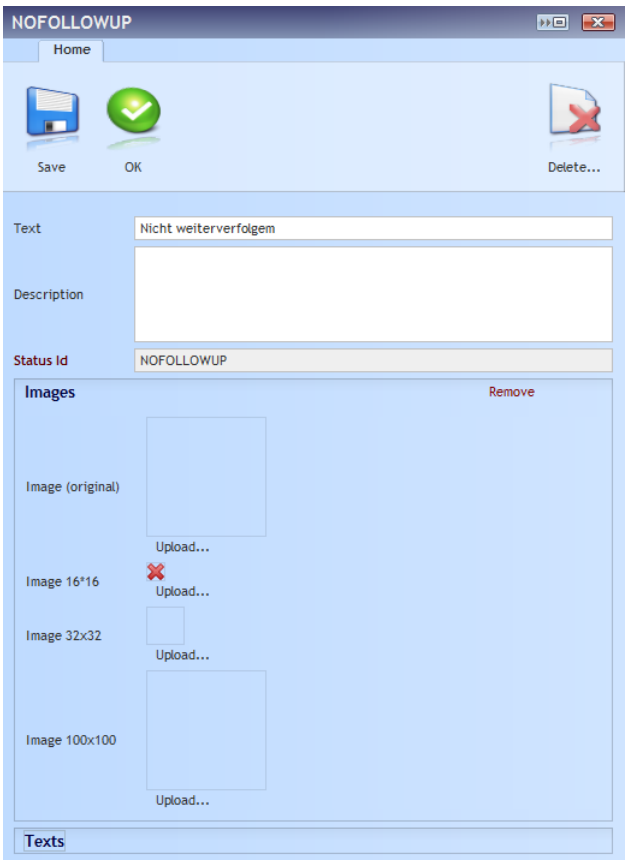
- "language" - either "de"/"en" (or what you have defined) or "ALL" for these entities for which the text is automatically derived from the attributes.
- "entity key" - a one-string-field of the entity, that uniquely identifies the entity within the database. You see that the name of the entity comes first, followed by its primary key
- "text" - the text

By keeping all literals in one table, we will be able to apply central buffering of text information and we are able to collect literal data for various entities in "one big fetch" rather than individual fetches per entity type.

Image Management

The image management is very similar to the text management – you can define that an entity holds images by clickig the "with Images" checkbox within the entity definition. As result the entity will be extended by an image aspect, that allows to define 4 differently sized images with the entity instance.

Example: the "customerStatus" entity is also configured to be associated:



The “16*16” image is the one that automatically is embedded everywhere where the entity is referenced – lists, combo boxes, etc.

Status Management

...

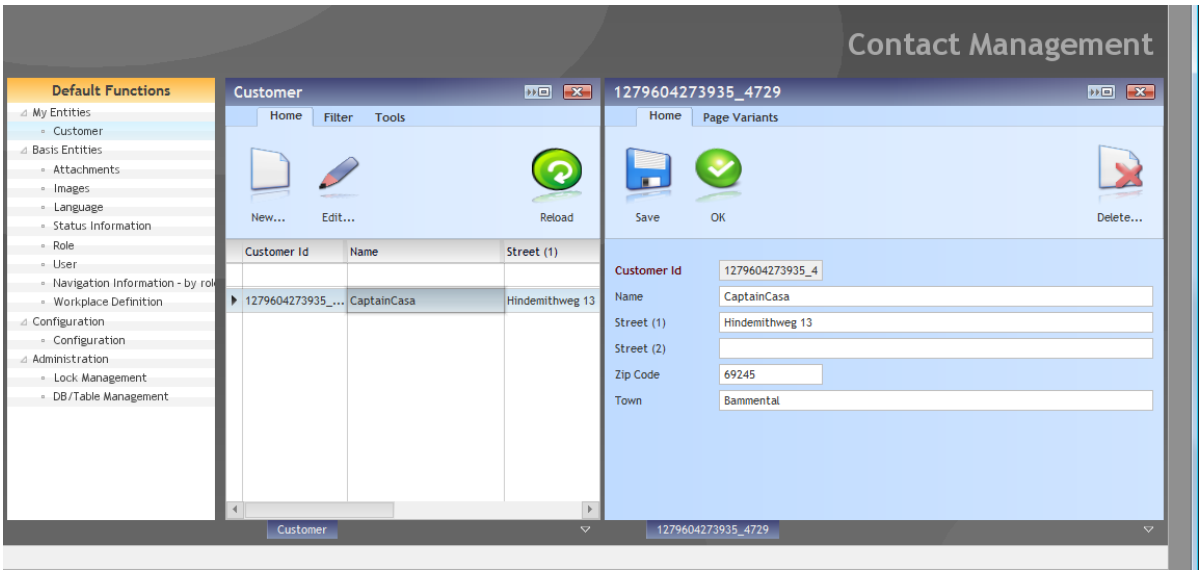
Using CC-SDM Applications

This chapter describes how to use applications that are created on base of CC-SDM.

Workplace

The workplace consists of 3 parts (tpyically):

- function selection (lef)
- list area (center)
- detail area (right)



Both the list area and the detail area are so called “workpage containers”, i.e. they may hold multiple content areas (“workpages”). You can switch between the currently opened content areas by pressing the corresponding tabs on the bottom of the area.

Entity List

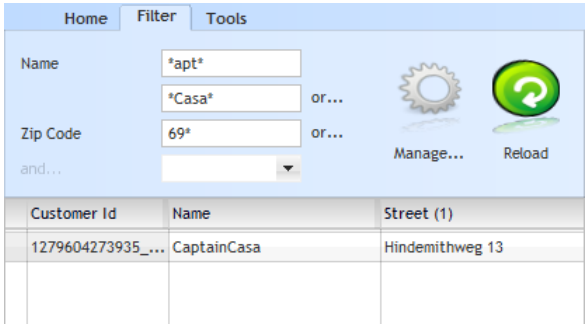
Basic Functions

Within the list of entities there are a couple of functions available:

- Sorting – you can sort the list by double clicking the corresponding header title
- Ad hoc filtering – you can filter the list by directly defining the filter in the header line of the grid. After specifying the filter fields you need to press the “Reload” icon to re-query the database.
- Editing – By double clicking an item you open up the detail view of the item in the detail area

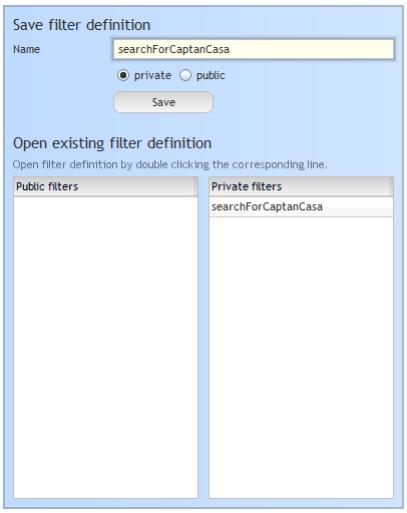
Extended Filter

There is an extended filtering – available in the “Filter-Tab”:



Here you can add complex queries in a simple way. Again, after defining the filter criteria you need to press the “Reload” icon in order to start the query.

When pressing the “Manage...” icon you will see a window, in which you can save your filter definition under a certain id. You either can save the filter as private filter (bound to your user) – or you can save the filter as public filter (for all users).



Tools

In the tools-tab there are functions to re-arrange the column layout and to export and import the data shown in the list into various output formats.

Entity Detail

The detail view is a plain form. When the entity has multiple embedded entities (1:n embedded relationship) then you'll see a corresponding grid. By double clicking a grid item you will get into the detail of the clicked item.

Inside the entity detail there are two main functions – aside providing all the data fields...:

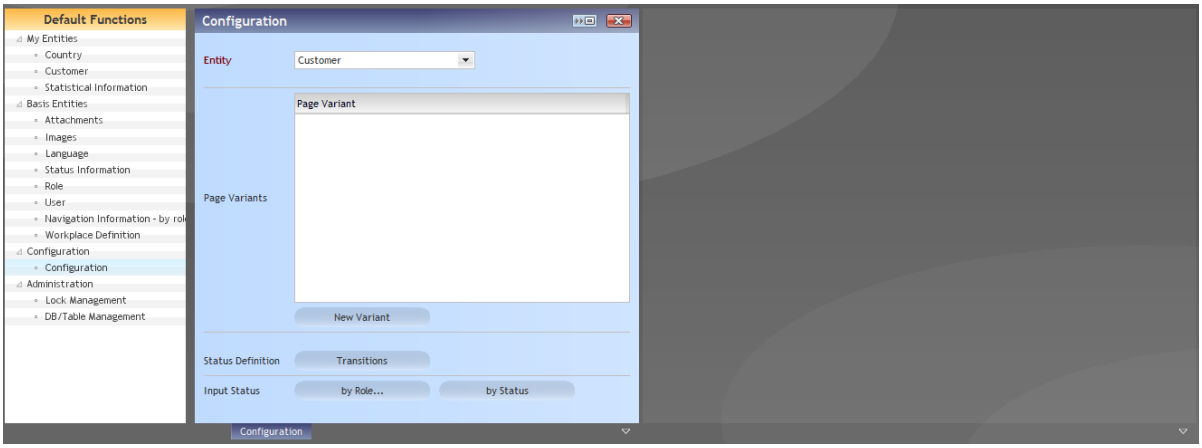
- “OK” - this validates the form data. By default all the link-relations that are defined within the entity model are checked.
- “Save” - this validates the form data, writes the updated content to the database and closes the form.

Entity Configuration

There are certain functions that can be used to configure CC-SDM at runtime. These

functions typically are part of a customization process at customer site. This means: they are not part of the toolset that you use within the Layout Editor, but they are part of a runtime toolset coming with each CC-SDM based application.

You reach these functions by opening menu item “Configuration” from the default workplace:

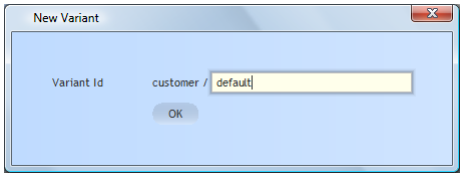


Select one of the entities and then use the buttons below to start the corresponding function.

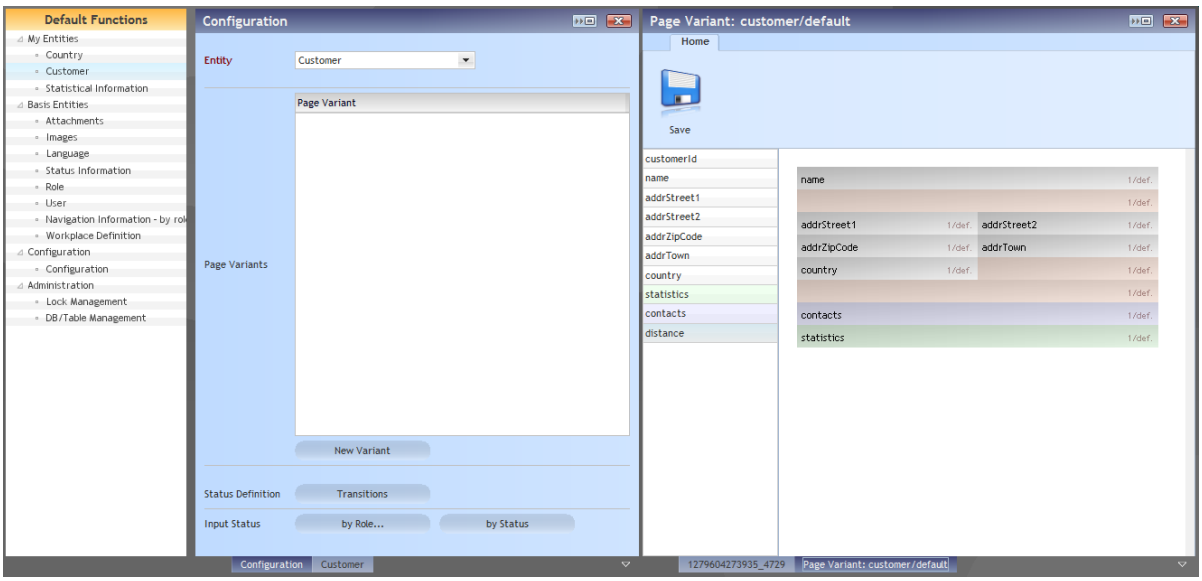
Creating customized Screen Variant

The screen / form of an entity is derived from its data structured – that's what happens by default. But: you can also define own screen variants in which you can arrange these properties that you want to see in a certain sequence.

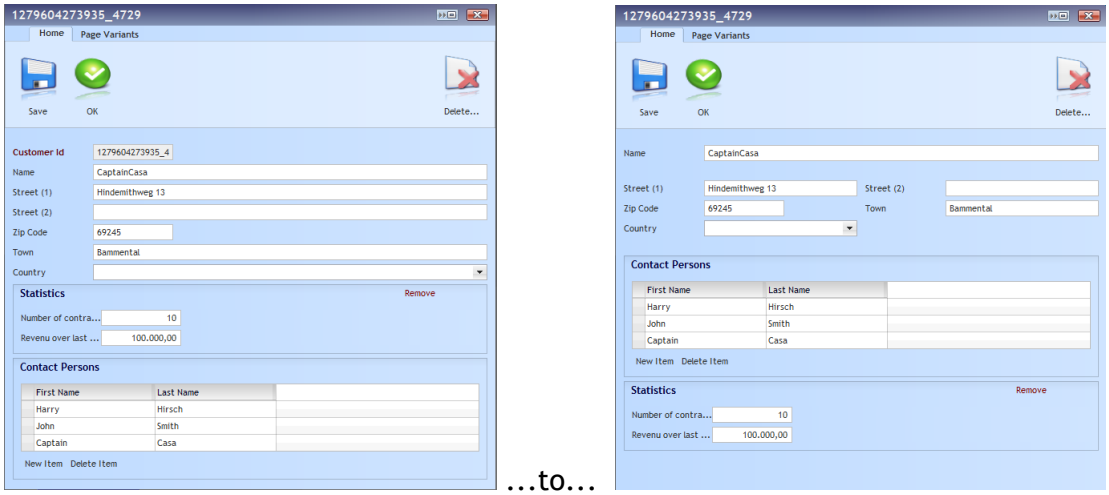
Select the entity in the tool and press the “New Variant” button. Select variant “default” in the window that pops up:



Now you get into an editor, in which you see all the properties of the entity on the left. You can drag & drop the properties into the area on the right and build up a corresponding screen layout:



After saving the layout of the corresponding screen will change:



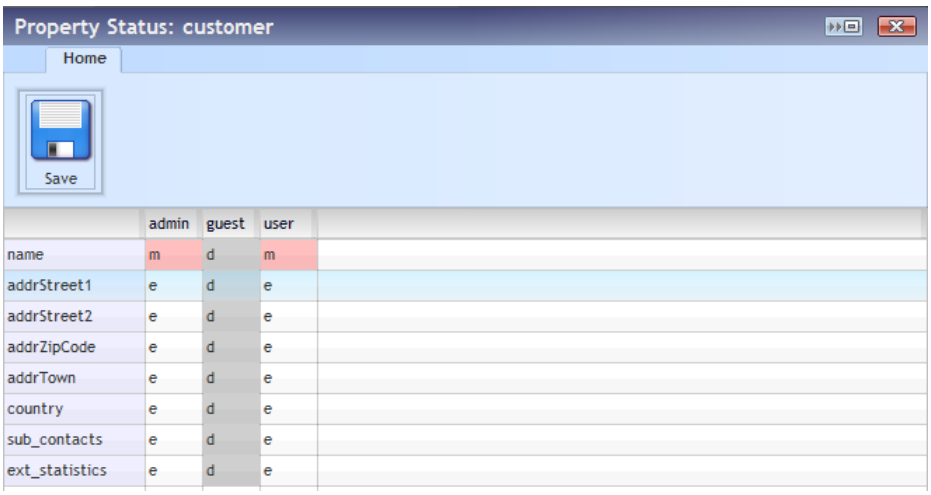
This is quite powerful function: the end user can re-design screens on his/her own without development effort.

Please pay attention: once having defined an own form variant for an entity, you will need to check new properties that you assign to the entity, if they should become part of your form or not. By default, new properties are not included automatically in your variant definitions.

Property-based Definition of Input Status

When clicking the button “Input Status – by Role...” then an editor will open up. The editor shows all the user roles defined in the system. You now can select by property if it is...

- “e” - editable
- “d” - display only
- “m” - mandatory



Status-based Definition of Input Status

...

Basis Configuration

There are a few configurations that you ought to do that are not part of your application but are part of the CC-SDM configuration.

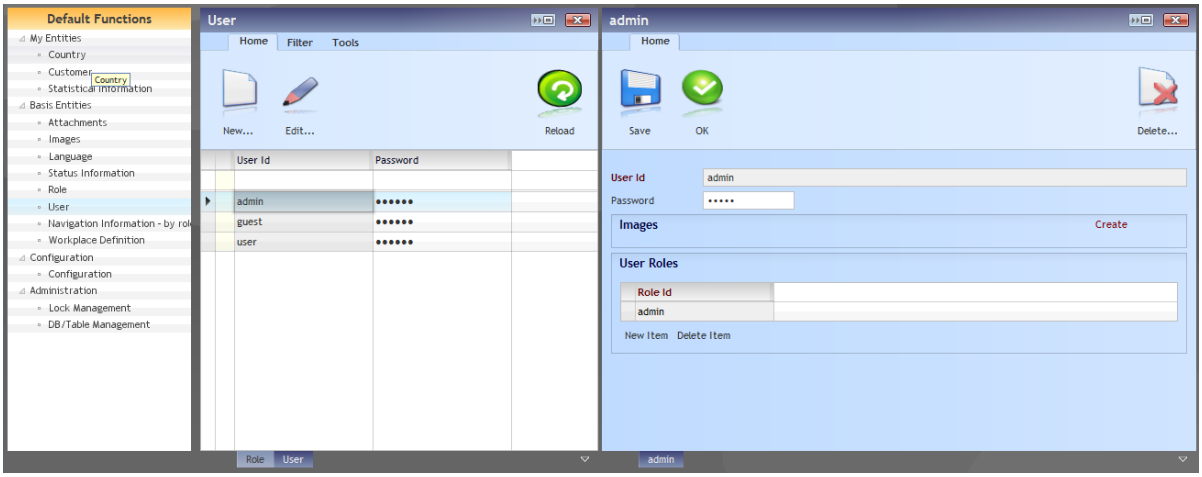
- Definition of roles and users
- Definition of languages

The configurations are part of the default workplace.

Roles and Users

You can set up various roles and users. A role is the abstraction of a certain class of user, e.g. “admin”, “applicationuser”, “guest”. A user is a concrete user account for a names user.

First maintain the roles, then maintain the users and assign the roles to the users. Though from data structure point of view a user may hold more than one role currently, please assign exactly one role to one user currently.

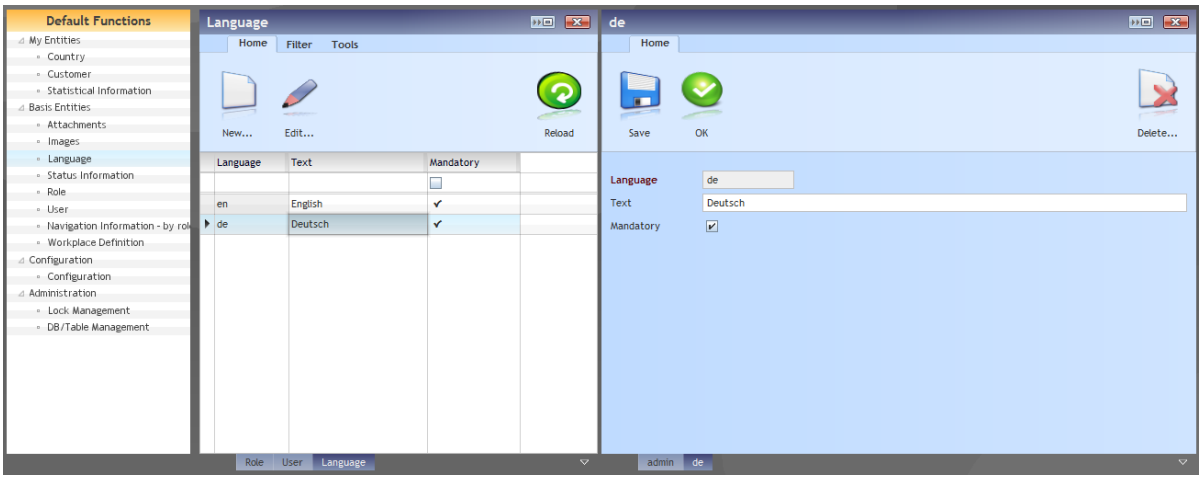


Currently there are the following functions that use the role concept:

- Input status management for entity properties
- Definition of workplace for a role

Definition of Languages

When using the built in, optional literal management for an entity then you need to define the languages that you want to maintain.



For each language you need to define the code (use ISO-codes for simplicity reason...) and the information if it's required to maintain the literal for this language.

Defining own, role-based Workplaces

By default the workplace shows all available entities and some system functions. Sequence and structure of the workplace function tree is technically driven.

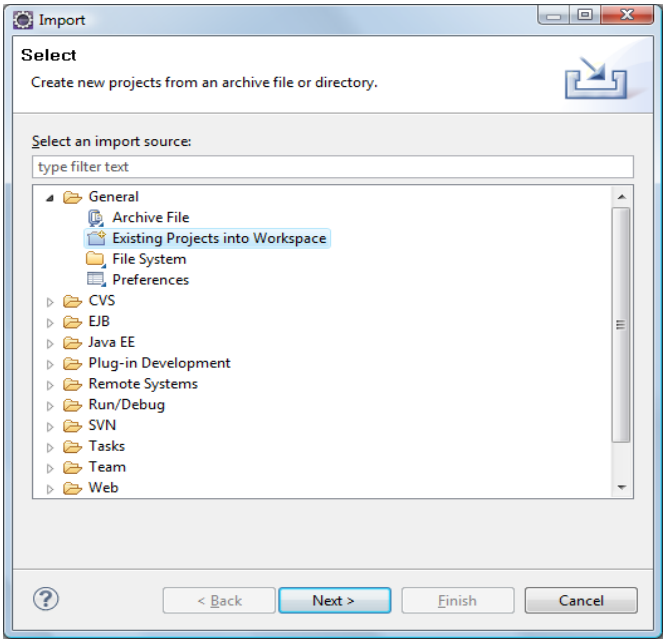
You can set up own workplace definitions easily and assign them to user-roles, so that dependent from their roles user will get access to a different set of functions.

Adding own Logic & Functions

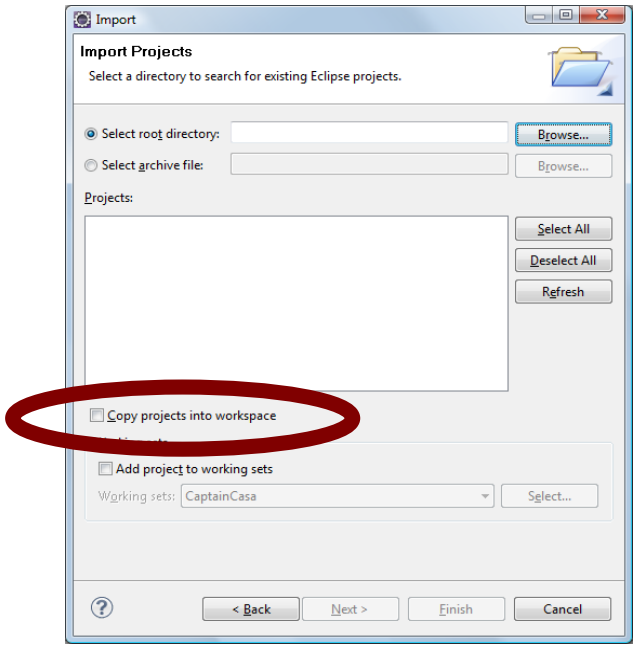
Importing the Project to Eclipse

During project creation you selected a certain directory. For working with the project within Eclipse (or other development environments) simply import the project directory:

Select “File > Import...” from the menu and define “Existing Projects into Workspace”:



In the following dialog select the project directory and pay attention to that the checkbbox “Copy projects into Workspace” is NOT selected.



After importing the project it will be visible in the project list. Its name is

“ec_<projectname>” (“ec” for Enterprise Client).

Generation Results

In the project you can immediately see what is generated for each entity definition.

Data-Access Layer

The data access layer manages the access to the database. For each entity the following classes are generated:

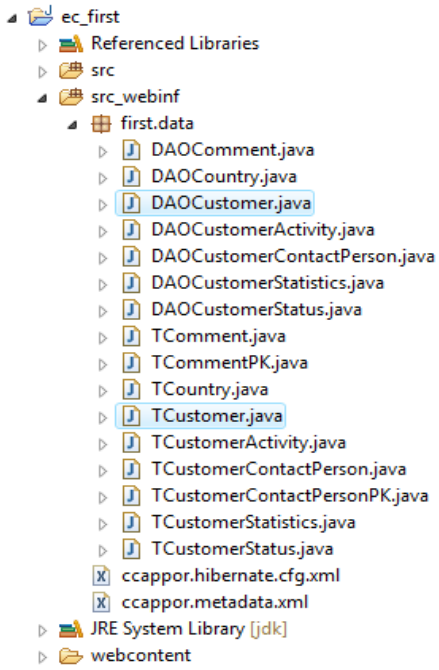
- The “POJO-Class” for the entity (naming convention “T<entity name>”)
- The “DAO-Class” fot the entity (naming cnovention “DAO<entity name>”)

The code is generated into the “src_webinf” directory of the project.

The data access layer internally uses the Hibernate framework for accessing the data from the database.

Example - POJO / “T”-Class

Let's take a look into the example project that was used in the previous chapters. Per entity a “T”-class and a “DAO”-class was generated.



Looking into the “T”-class (“T” for “table”) then you see that this class is the Hibernat-Pojo-Class that both defines the entity itself and defines its representation within the database:

```
package first.data;

import org.eclnt.ccappor.rt.interfaces.*;
import java.io.*;
import java.lang.*;
import java.math.*;
import java.util.*;
import javax.persistence.*; // Annotations
```

```
@Entity
@Table(name="t_customer")

public class TCustomer
  implements Cloneable,
             ICloneable<TCustomer>,
             ITrackable,
             IUniqueKeyProvider,
             Serializable
{
    // ---------------------------------------------------------------------
    // constants for property names
    // ---------------------------------------------------------------------

    public static final String PROP_CUSTOMERID = "customerId";
    public static final String PROP_NAME = "name";
    public static final String PROP_ADDRSTREET1 = "addrStreet1";
    public static final String PROP_ADDRSTREET2 = "addrStreet2";
    public static final String PROP_ADDRZIPCODE = "addrZipCode";
    public static final String PROP_ADDRTOWN = "addrTown";
    public static final String PROP_COUNTRY = "country";
    public static final String PROP_STATUS = "status";

    // ---------------------------------------------------------------------
    // member/ property definitions
    // ---------------------------------------------------------------------

    private transient int m_ccapporTrackIndex = 0;

    private String m_customerId;
    @Id
    @Column(length=25)
    public String getCustomerId() { return m_customerId; }
    public void setCustomerId(String value) { m_customerId = value;
m_ccapporTrackIndex++; }

    private String m_name;
    @Column(length=100)
    public String getName() { return m_name; }
    public void setName(String value) { m_name = value;
m_ccapporTrackIndex++; }

    private String m_addrStreet1;
    @Column(length=100)
    public String getAddrStreet1() { return m_addrStreet1; }
    public void setAddrStreet1(String value) { m_addrStreet1 = value;
m_ccapporTrackIndex++; }

    private String m_addrStreet2;
    @Column(length=100)
    public String getAddrStreet2() { return m_addrStreet2; }
    public void setAddrStreet2(String value) { m_addrStreet2 = value;
m_ccapporTrackIndex++; }

    private String m_addrZipCode;
    @Column(length=10)
    public String getAddrZipCode() { return m_addrZipCode; }
    public void setAddrZipCode(String value) { m_addrZipCode = value;
m_ccapporTrackIndex++; }

    private String m_addrTown;
    @Column(length=100)
    public String getAddrTown() { return m_addrTown; }
    public void setAddrTown(String value) { m_addrTown = value;
m_ccapporTrackIndex++; }

    private String m_country;
    @Column(length=25)
    public String getCountry() { return m_country; }
    public void setCountry(String value) { m_country = value;
m_ccapporTrackIndex++; }
```

```
    private String m_status;
    @Column(length=25)
    public String getStatus() { return m_status; }
    public void setStatus(String value) { m_status = value;
m_ccapporTrackIndex++; }



    private String m_uniqueKey;
    @Column(length=100)
    public String getUniqueKey()
    {
        if (m_uniqueKey != null) return m_uniqueKey;
        else return buildUniqueKey();
    }
    public void setUniqueKey(String value) { m_uniqueKey = value;
m_ccapporTrackIndex++; }


    // ------------------------------------------------------------------
    // cloneable
    // ------------------------------------------------------------------

    public Object clone() throws CloneNotSupportedException { return
super.clone(); }

    public TCustomer createClone()
    {
        try
        {
            return (TCustomer)super.clone();
        }
        catch (CloneNotSupportedException exc)
        {
            throw new Error(exc);
        }
     }

    // ------------------------------------------------------------------
    // trackable
    // ------------------------------------------------------------------

    public int trackIndex()
    {
        return m_ccapporTrackIndex;
    }

    // ------------------------------------------------------------------
    // entity key
    // ------------------------------------------------------------------

    public String buildUniqueKey()
    {
        return "customer:" + m_customerId;
    }

}
```

In addition there are some useful add-ons, e.g. there are constant defintions (“PROP_*”) for each property, there is a clone interface, etc.

Example – DAO Class

While the POJO-class represents one single entity object, the DAO class represents an API to access the object, both for read and for write access.

It's not very interesting to take a look into the generated DAO class, because it internally derives from some other generic class – so that the core logic of the DAO class is not

visible within its code itself.

But: it's very interesting to see the DAO-interface “ICrudDAO” that is implemented by each DAO class:

```
package org.eclnt.ccappor.rt.dao;

import java.util.List;

import org.eclnt.ccappor.rt.interfaces.CrudEqualsQuery;
import org.eclnt.ccappor.rt.interfaces.CrudQuery;


public interface ICrudDAO<POJOCLASS,POJOKEY>
{
    public POJOCLASS createNewInstance();
    public POJOKEY extractKey(POJOCLASS object);
    public void save(POJOCLASS object);
    public void delete(POJOCLASS object);
    public POJOCLASS read(POJOKEY key);
    public List<POJOCLASS> queryAll();
    public List<POJOCLASS> query(String sqlJoin, CrudQuery query);
    public List<POJOCLASS> query(CrudQuery query);
    public List<POJOCLASS> query(CrudEqualsQuery query);
    public void delete(CrudQuery query);
}
```

The interface is a typcial “CRUD”-interface (create, retrieve, update, delete), that is very easy to use.

Logic Layer

The data access layer only provides simple data access for each entity type.

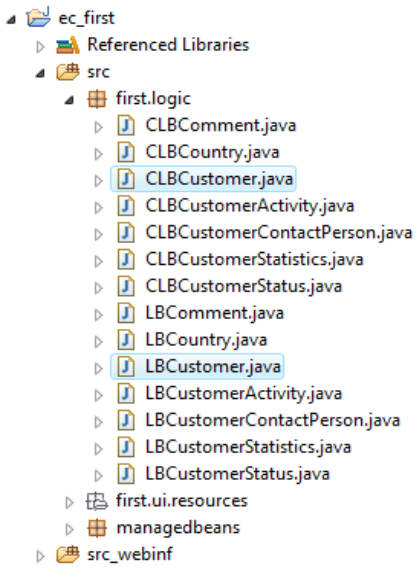
The logical layer is the one to now add:

- rules (e.g. checking relationships, checking mandatories, ...)
- combined view on entities - if you defined an entity to hold sub-aspects or extension-aspects, then these will be managed on this layer
- information that is required by the user interface, e.g.
    - the display type for each property (is it mandatory?)
    - the access to valid values for a certain property

Per entity type there are two classes that are generated:

- The “logic bean” (naming convention “LB_<entity name>”)
- The “CRUD access to logic beans” (naming convention “CLB_<entity name>”)

The classes are generated into the “src” directory of you project:



Logic Bean Class - "LB"-Class

The logic bean class is a wrapper that wraps a certain entity instance. The data of the entity instance is represented by the POJO/"T"-class, now all the logic around the bean is kept in the logic bean class.

The generated class itself is derived from some other classes, so the generated code is not of great interest. But: the interface that is implements "ILogicaBean" is interesting:

```
package org.eclnt.ccappor.rt.logic;

import java.util.List;
import java.util.Set;

import org.eclnt.ccappor.metadata.MDEntity;
import org.eclnt.ccappor.rt.context.ISessionInformation;
import org.eclnt.ccappor.rt.interfaces.CrudEqualsQuery;
import org.eclnt.ccappor.rt.interfaces.CrudQuery;

/**
 * Interface of logic that is arrange around a certain bean in order
 * to serve frontend form needs.
 */
public interface ILogicBean<BEANCLASS,OWNERLBCLASS extends ILogicBean<?,?
>>
{
    public static int PDT_HIDDEN = 0;
    public static int PDT_DISABLED = 1;
    public static int PDT_EDIT = 2;
    public static int PDT_MANDATORY = 3;

    public static int ROLE_SUBASPECT = 0;
    public static int ROLE_EXTENSIONASPECT = 1;

    // bean access
    public BEANCLASS getBean();
    public BEANCLASS getLastSavedVersionOfBean();
    public boolean isNew();
    public MDEntity getMDEntity();

    // information about properties
    public int getDisplayTypeForProperty(String propertyName,
ISessionInformation sessionInformation);
    public int getDisplayTypeForFunction(String functionName,
ISessionInformation sessionInformation);
    public int getDisplayTypeForSubAspect(String subAspectName,
```

```
ISessionInformation sessionInformation);
    public int getDisplayTypeForExtensionAspect(String
extensionAspectName, ISessionInformation sessionInformation);

    // sub bean management
    public Set<String> getSubBeanIds();
    public ILogicBean<?,?> getExtensionBean(String extensionBeanId);
    public LogicBeanList<? extends ILogicBean<?,?>> getSubBeans(String
subBeansId);
    public ILogicBean<?,?> createSubBean(String subBeanId) throws
Exception;
    public void removeSubBean(String subBeanId, ILogicBean<?,?> subBean)
throws Exception;
    public void setOwner(OWNERLBCLASS owner,int roleInOwner,String
nameInOwner);
    public OWNERLBCLASS getOwner();
    public ILogicBean getTopOwner();

    // loading
    public void processAfterCreation() throws Exception;
    public void processAfterLoad() throws Exception;

    // function processing
    public void executeFunction(String functionId) throws Exception;

    // validation
    public void synchronize() throws Exception;

    public void validate() throws Exception;
    public void processBeforeCheck() throws Exception;
    public void check() throws Exception;
    public void processAfterCheck() throws Exception;

    // saving and processing of bean
    public void post() throws Exception;
    public void processBeforeSave() throws Exception;
    public void save() throws Exception;
    public void processAfterSave();

    public Throwable getCurrentException();

    // persistence
    public void delete() throws Exception;

    // entity query interface for contained beans
    public List<Object> queryValidEntities(String property);
    public void queryWithinOwnObjects(List<Object> resultList, Class
entityClass);

    // access to property value check (if available by MDProperty
definition)
    public IPropertyValueCheck getPropertyValueCheckInstance(String
property);

    // listener managemenet
    public void addListener(ILogicBeanListener listener);
    public void removeListener(ILogicBeanListener listener);
}
```

<missing: details to be added>

CRUD Logic Bean Class - “CLB” Class

While the logic bean is a wrapper for a single entity, the “CRUD” logic bean class is the one to retrieve logic bean instances. The interface that is implemented is:

```
package org.eclnt.ccappor.rt.logic;

import java.util.List;
```

```
import org.eclnt.ccappor.rt.interfaces.CrudEqualsQuery;
import org.eclnt.ccappor.rt.interfaces.CrudQuery;

/**
 * Interface to query and to create instances of beans/ logic beans.
 */
public interface ICrudLogicBean<BLCLASS extends ILogicBean<BEANCLASS,?
>,BEANCLASS,BEANKEY>
{
    public BLCLASS createNewInstance();
    public BLCLASS read(BEANKEY key);
    public BLCLASS readAndLock(BEANKEY key) throws Exception;

    public BEANKEY extractKey(BEANCLASS object);

    public List<BEANCLASS> queryAll();
    public List<BEANCLASS> query(String sqlJoin, CrudQuery query);
    public List<BEANCLASS> query(CrudQuery query);
    public List<BEANCLASS> query(CrudEqualsQuery query);
}
```

The interface provides:

- Access to the logic bean class (“createNewInstance, read, readAndLock”)
- Access to the POJO entity bean instances (“query*” methods)

User Interface Layer

There is no generation of code and no generation of JSP-pages on UI layer. The UI layer is completely dynamic.

Accessing Entities

When implementing own user interfaces that are non-generated user interfaces, then you might want to use the generated SDM classes in order to read or update entities.

Reading Entities

Using the generated Classes

Reading data is the most simple scenario.

```
    public void onQueryCustomers(ActionEvent event)
    {
        CLBCustomer crud = new CLBCustomer();
        List<TCustomer> customers = crud.queryAll();
        for (TCustomer customer: customers)
        {
            System.out.println(customer.getCustomerId());
        }
    }
```

For doing more complex queries you can use two special query methods.

```
    public void onQueryCustomers(ActionEvent event)
    {
        CLBCustomer crud = new CLBCustomer();
        List<TCustomer> customers = crud.query
        (
            new CrudQuery
            (
                "name LIKE :p0 or name LIKE :p1",
                new Object[] {"aaa","bbb"}
```

```
            )
        );
        for (TCustomer customer: customers)
        {
            System.out.println(customer.getCustomerId());
        }
    }
```

You can pass any selection SQL string for the corresponding entity. Parameters in the string need to be named with ":p0", ":p1", etc. There is an object array into which you can pass the parameters.

You also can directly define queries for typical simple select variants, in which each attribute may only be listed once and in which the query is done with "="-operator:

```
    public void onQueryCustomers(ActionEvent event)
    {
        CLBCustomer crud = new CLBCustomer();
        List<TCustomer> customers = crud.query
        (
            new CrudEqualsQuery
            (
                new String[] {"name"},
                new Object[] {"aaa"}
            )
        );
        for (TCustomer customer: customers)
        {
            System.out.println(customer.getCustomerId());
        }
    }
```

Directly accessing Hibernate

You can any time directly access Hibernate and use all the other Hibernate by retrieving a Hibernate session and working with it:

```
    public void onQueryViaHibernate(ActionEvent event)
    {
        Session hibernateSession =
ThreadContext.currentInstance().getHibernateSession();
        // ...
        // ...
        // ...
    }
```

Please note:

- A hibernate session is only valid within the current request / response processing of the user interface
- There is one hibernate session that is assigned to the request / response cycle (thread)

Updating Entities

Of course it's possible to directly update entities by accessing the data access layer – but of course you should not! Updating entities should always go through the logical layer!

Updating an entity needs to be done with some transactional scope in mind. Someone needs to open up a transaction and to commit / rollback depending on successful operations.

In order to simplify the processing of transactions, the SDM layer provides a quite simple to use mechanism to do so. You need to open up a so called "Activity" and then execute an operation in the context of this activity. The activity automatically embeds the operation into a transactional scope.

Example:

```
    public void onCreateCustomer(ActionEvent event)
    {
        try
        {
            Activity activity = new Activity();
            activity.executeOperation(new IOperation()
            {
                public void run() throws Exception
                {
                    CLBCustomer crud = new CLBCustomer();
                    LBCustomer lbCustomer = crud.createNewInstance();
                    lbCustomer.getBean().setName("New customer");
                    lbCustomer.getBean().setAddrStreet1("Street new");
                    lbCustomer.save();
                    Statusbar.outputSuccess("Customer was created " +
lbCustomer.getBean().getCustomerId());
                }
            });
        }
        catch (Throwable t)
        {
            Statusbar.outputError("Problem occurred: " + t.toString());
        }
    }
```

In the following example all customers will be updated:

```
    public void onUpdateCustomers(ActionEvent event)
    {
        try
        {
            Activity activity = new Activity();
            activity.executeOperation(new IOperation()
            {
                public void run() throws Exception
                {
                    CLBCustomer crud = new CLBCustomer();
                    List<TCustomer> customers = crud.queryAll();
                    for (TCustomer customer: customers)
                    {
                        LBCustomer lbCustomer =
crud.readAndLock(customer.getCustomerId());
                        lbCustomer.getBean().setAddrStreet2("UNKNOWN");
                        lbCustomer.save();
                    }
                    Statusbar.outputSuccess("Customers were updated");
                }
            });
        }
        catch (Throwable t)
        {
            Statusbar.outputError("Problem occurred: " + t.toString());
        }
    }
```

Thread Context - Multi Tenancy, User Management

When logging on to SDM then you by default see a logon screen in which you need to define the system (tenant) you are working and the user name. Both tenant and user are assigned to the server side (http-) session.

During request/response processing each thread automatically gets assigned its context information.

You may set the context information in the following way:

```
    public void onSetContext(ActionEvent event)
```

```
    {
        ThreadContext.currentInstance().assignNewSessionInformation
        (
            new SessionInformation
            (
                "cctest",
                "username",
                "en"
            )
        );
    }
```

Default Behaviour

When no explicit context information is passed, then the tenant that is used is “cctest”.

Adding own Logic to Entity Processing

…

Extending / Changing the Workplace

…

Connecting to your own Database

By default CaptainCasa SDM ships with a lightweight database “Hypersonic HSQL”. This database does not require some explicit setup-steps but directly runs within the Java process of the web application. The database is “optimal” for being used in no-installation-effort scenarios, in which you do not want to first advice the customer that a database needs to be installed and configured.

The embedding of an own database needs to be done by implementing a certain extension interface, and by making the implementation access-able for the SDM logic.

Interface HibernateSessionFactory.IExtension

The Hibernate session factory that is internally used within SDM (class “HibernateSessionFactory”) has an extension mechanism: requests for new database connections are passed to the interface IExtension:

```
public interface IExtension
{
    public AnnotationConfiguration getApplicationConfiguration
                                    (String tenant);
}
```

You need to implement the interface and pass back the Hibernate configuration for a given tenant (this is the system-id that is defined by the user when logging on).

In order to give you an example, please have a look onto the default implementation that is used:

```
private static class DefaultExtensionHSQL implements IExtension
{
    public AnnotationConfiguration getApplicationConfiguration
                                    (String tenant)
    {
        AnnotationConfiguration ac = null;
        String webappDir = HttpSessionAccess.getServletTempDirectory();
        webappDir = ValueManager.encodeIntoValidFileName(webappDir,true);
```

```
        String dbFileName = webappDir+"/ccappordata/DATA"+tenant;
        FileManager.ensureDirectoryForFileExists(dbFileName);
        String dbURL = "jdbc:hsqldb:file:"+dbFileName;
        ac = new AnnotationConfiguration()
                .setProperty(Environment.CURRENT_SESSION_CONTEXT_CLASS,"
thread")
                .setProperty(Environment.POOL_SIZE,"1")
                .setProperty(Environment.CACHE_PROVIDER,"org.hibernate.c
ache.NoCacheProvider")
                .setProperty(Environment.DIALECT,"org.hibernate.dialect.
HSQLDialect")
                .setProperty(Environment.SHOW_SQL,"true")
                .setProperty(Environment.HBM2DDL_AUTO,"update")
                .setProperty(Environment.DRIVER,"org.hsqldb.jdbcDriver")
                .setProperty(Environment.URL,dbURL)
                .setProperty(Environment.USER,"SA")
                .setProperty(Environment.PASS,"");
            return ac;
    }
}
```

You see that a different database URL is built for each tenant, so that as result each tenant is managed within an own database instance.

Register your IExtension Implementation

Once having finished your implementation of IExtension, you need to register your extension, so that the SDM runtime does not use its default implementation but uses your implementation.

The common way for CaptainCasa SDM to look for implementations of certain interfaces is to check the central class “InterfaceFactory”. The following code is taken from the class HibernateSessionFactory and is exactly the place where the extension is loaded:

```
private static IExtension getExtension()
{
    if (s_extension != null)
        return s_extension;
    s_extension =  (IExtension)InterfaceFactory
                                .getInterface(IExtension.class);
    if (s_extension == null)
    {
        s_extension = new DefaultExtensionHSQL();
    }
    return s_extension;
}
```

You see: internally, the InterfaceFactory is checked, if an instance of HibernateSessionFactory.IExtension is registered. If yes, then this instance is used – if no, then a default instance is built up.

So, what you have to do is: you need to create an instance of your IExtension-implementation and register it in the InterfaceFactory. Example:

```
// creation of your implementation
HibernateSessionManager.IExtension ext = new MyExtension();
// registration
InterfaceFactory.addInterface(HibernateSessionManger.IExtension.class,
                              ext);
```

Now, the last question remains: when is a good point of time of executing this registration code? - The answer: at start up time of SDM. So, the next section tells you how to become a member of the SDM start up.

Becoming a Member of the SDM Start up

When SDM starts, it checks the very central application definition that you make as part of the SDM configuration. Please check the chapter “One time definitions” at the very beginning of this documentation).

This definition contains a definition of a so called “Initialization Class”. This is a name of a class, which is instanciated by the SDM runtime during its start up. The instanciation is done using a constructor without parameters. Within the constructor you can execute all initialization steps, such as registering the HibernateSessionManager.IExtension-interface instance.

Example:

- In the application definition you define the class name “xyz.StartUp”
- You define the following class:

```
package xyz;

public class StartUp
{
    public StartUp()
    {
       // all you initialization... e.g.:
       HibernateSessionManager.IExtension ext = new MyExtension();
       InterfaceFactory.addInterface(HibernateSessionManger
                                          .IExtension.class,
                                     ext);
    }
}
```

...that's it!

CaptainCasa GmbH

Hindemithweg 13
D-69245 Bammental

+49 6223 484147

http://www.CaptainCasa.com
info@CaptainCasa.com