ZBox Guide

The Hitchhiker's guide to the ZBox galaxy

Contents

Chapter 1

Introduction

ZBox is a application framework to provide the complete process from defining data structures, designing data access and transfer objects, designing servers and GUIs and the necessary parts to make everything work together.

1.1 ZBox Basic

ZBox Basic provides the folowing services:

Module editor All meta informations are defined by the module editor

Database Provider The Database provider manages database access. Currently SQL Server 2008 & PosgreSQL are supported.

Chapter 2

Programming

This chapter describes the various ways and pieces the ZBox system is programmed and customized.

2.1 Objects

2.1.1 ObjectClass

ZBox knows four kinds of Types, all derived from *DataType*:

ObjectClass Type for ZBox Object Classes

Interface Type for ZBox Interfaces

Enumeration Type for ZBox Enumerations

CompoundObject Type for ZBox Object Classes

DataType

This is the abstract base class for all ZBox Types. It provides the necessary infrastructure to describe a Type.

```
public interface DataType
{
    string Name;
    string Description;

    IList<Property> Properties;
    ICollection<Method> Methods;
    ICollection<MethodInvocation> MethodInvocations;

    Icon DefaultIcon;
    ICollection<InstanceConstraint> Constraints;
}
```

Name The name of the *DataType*. Note that this name has to be a valid C# name. A *Constraint* protects this.

Description Each Type should have a description. This description is used for documentation purposes.

Properties Each Type can have Properties (except *Enumeration*)

Methods Each Type can have Methods (except *Enumeration*)

MethodInvocations Methods on a *DataType* are invoked by an Method-Invokation. It does not matter on which level of type hierarchy the invokation is defined.

DefaultIcon Each *DataType* can have a default icon

Constraints A *Constraint* checks the validity of an instance. If any constraint throws an error nothing will be committed.

ObjectClass

ObjectClass is *the* defining class for ZBox Objects.

```
public interface ObjectClass : DataType
{
    ObjectClass BaseObjectClass;
    ICollection<Interface> ImplementsInterfaces;

    string TableName;

    bool IsAbstract;
    bool IsFrozenObject;
    bool IsSimpleObject;

    ViewModelDescriptor DefaultViewModelDescriptor;
    ControlKind RequestedKind;

    ICollection<AccessControl> AccessControlList;
}
```

2.1.2 Relation

A *Relation* defines the relationship between two Objects. Every Object can have zero or more *Relations*.

An example of a *Relation* is the relation between *Project* and *Tasks*. One *Project* can have zero or more *Tasks*. One *Tasks* must have a *Project*.

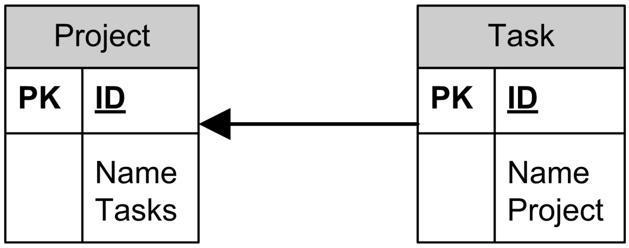


Figure 2.1: Example for a *Relation*

Modeling a relation

A Relation can be defined by creating an object of type *Relation* and two *RelationEnd* objects. This can be done by

- creating an Relation Object.
- invoking the *Create Relation* method on an *ObjectClass* instance.

RelationEnd objects will be created automatically.

Relations are edited in the *Relation Editor*. The *Relation Editor* is a custom *FullObjectView* created by us.

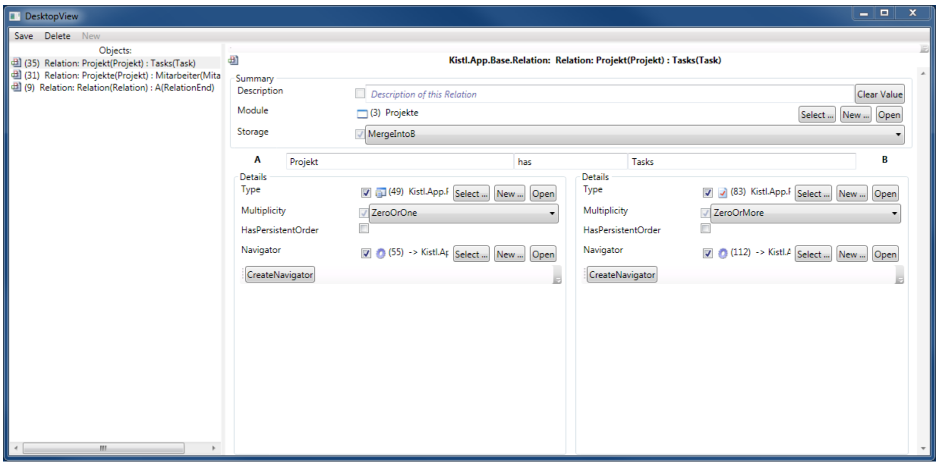


Figure 2.2: Example for editing a *Relation*

Attributes of a relation

A *Relation* has these attributes:

Description	A text property used to describe the current relation

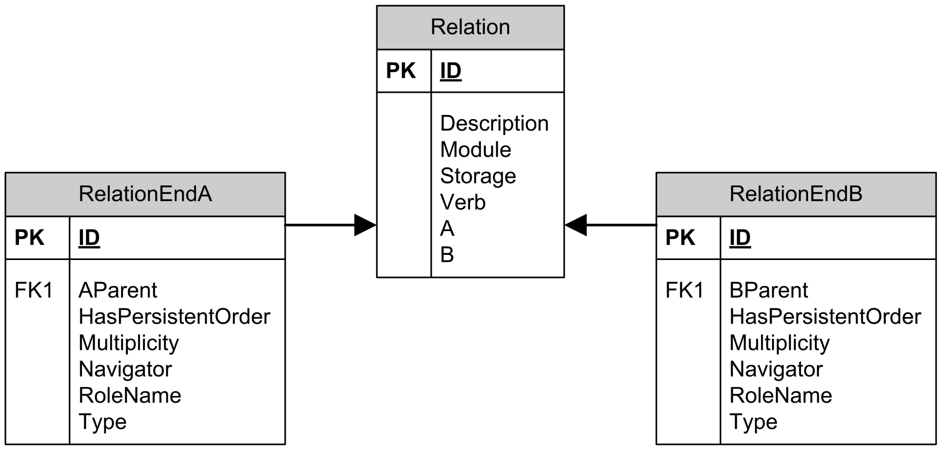


Figure 2.3: Attributes of a relation

Module The *Module* which is introducing the current relation

Storage The *StorageType* of the current relation

Verb A verb used to name the current relation. The verb is used in conjunction with the role names of the *RelationEnd* objects to model a unique relation name. This relation name will be used e.g. for the database FK Contraint name.

A The *RelationEnd* A of the current relation

B The *RelationEnd* B of the current relation

A *RelationEnd* has these attributes:

AParent *Relation* object if this *RelationEnd* is the A-Side of the current relation. Otherwise *NULL*

BParent *Relation* object if this *RelationEnd* is the B-Side of the current relation. Otherwise *NULL*

HasPersistentOrder Specifies that the list is ordered. Applies only to lists

Multiplicity The *Multiplicity* of the current *RelationEnd*

Navigator An optional *Navigator*

RoleName Name of the role of the current *RelationEnd*

Type *ObjectClass* to which the current *RelationEnd* points

There are four *StorageTypes* defined:

MergeIntoA The relation information is stored with the A-side database table

MergeIntoB The relation information is stored with the B-side database table

Replicate The relation information is stored on both sides of the relations database tables

Separate The relation information is stored in a separate database table

There are three *Multiplicities* defined:

ZeroOrOne Optional Element (zero or one)

One Required Element (exactly one)

ZeroOrMore Optional List Element (zero or more)

1:n Relation

A *Project* can have zero or more *Tasks*. A *Task* may have one *Project*.

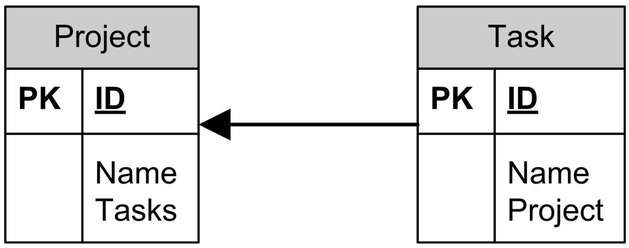


Figure 2.4: Project/Tasks relation

The *Relation* object would be:

Storage = MergeIntoB

Verb = has

The *RelationEnd* A object would be:

AParent = *Relation*

BParent = *NULL*

HasPersistentOrder = false

Multiplicity = ZeroOrOne. If a *Task* must have a *Project* then One.

Navigator = *Navigator* to Tasks. The result would be a collection of *Tasks* (ICollection<Task>)

RoleName = Project

Type = *Task* instance of type *ObjectClass*

The *RelationEnd* B object would be:

AParent = *NULL*

BParent = *Relation*

HasPersistentOrder = false

Multiplicity = ZeroOrMore

Navigator = *Navigator* to the parent *Project*. The result would be a reference to a *Project*

RoleName = Tasks

Type = *Project* instance of type *ObjectClass*

n:m Relation

A *Project* can have zero or more *ProjectMembers*. A *ProjectMember* can be assinged to zero or more *Projects*.

The *Relation* object would be:

Storage = Seperate

Verb = has

The *RelationEnd* A object would be:

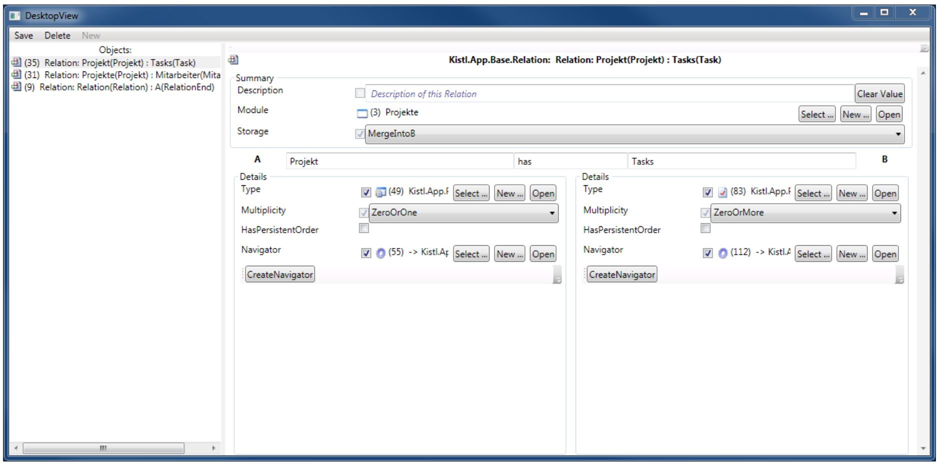


Figure 2.5: Editing the *Project/Tasks* relation

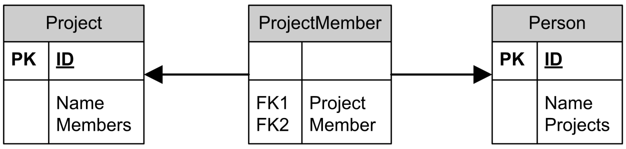


Figure 2.6: Project/Member relation

AParent = *Relation*

BParent = *NULL*

HasPersistentOrder = true

Multiplicity = ZeroOrMore.

Navigator = *Navigator* to Persons. The result would be a list of *Persons* (IList<Person>)

RoleName = Projects

Type = *Person* instance of type *ObjectClass*

The *RelationEnd* B object would be:

AParent = *NULL*

BParent = *Relation*

HasPersistentOrder = true

Multiplicity = ZeroOrMore

Navigator = *Navigator* to the assigned *Projects*. The result would be a list of *Projects* (IList<Project>)

RoleName = Member

Type = *Project* instance of type *ObjectClass*

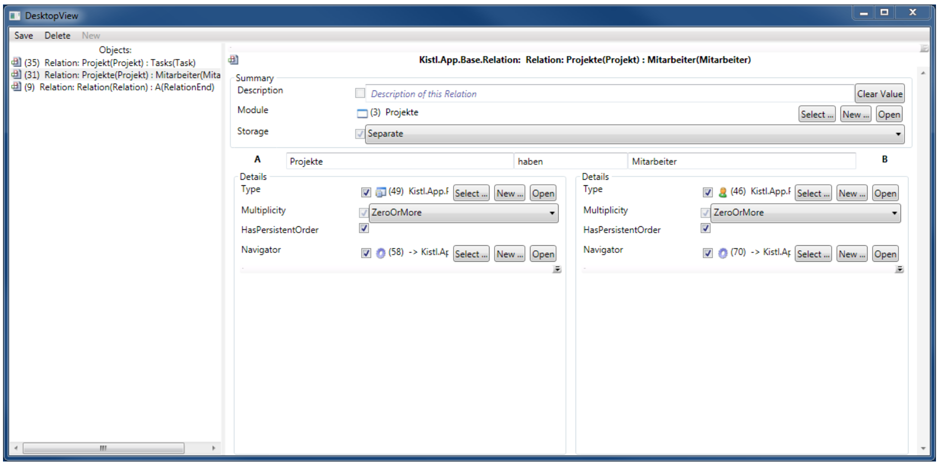


Figure 2.7: Editing the *Project/Member* relation

1:1 Relation

A *Relation* must have a *RelationEnd* A. A *RelationEnd* may have a AParent *Relation* if it's a A *ReleationEnd*.

The *Relation* object would be:

Storage = MergeIntoA

Verb = hasA

The *RelationEnd* A object would be:

AParent = *Relation*

BParent = *NULL*

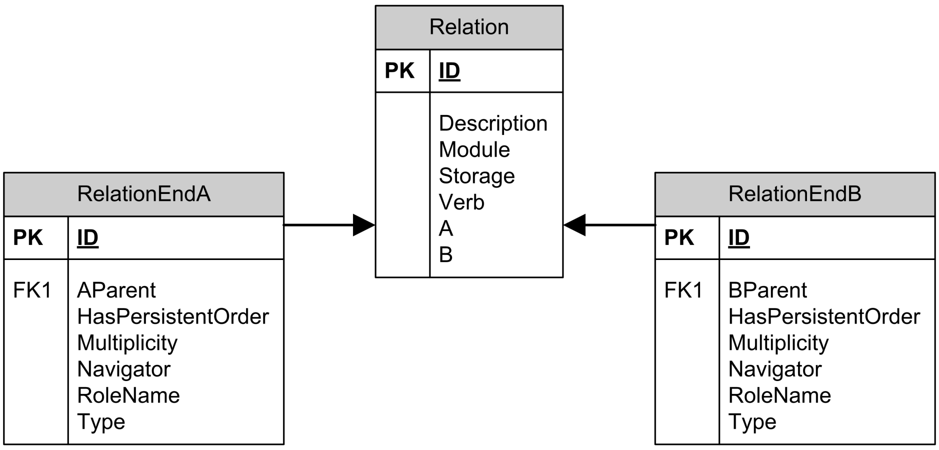


Figure 2.8: Relation/RelationEnd relation

HasPersistentOrder = false

Multiplicity = ZeroOrOne

Navigator = *Navigator* to *RelationEnd*. The result would be a reference to a *RelationEnd*

RoleName = Relation

Type = *RelationEnd* instance of type *ObjectClass*

The *RelationEnd* B object would be:

AParent = *NULL*

BParent = *Relation*

HasPersistentOrder = true

Multiplicity = One

Navigator = *Navigator* to the assigned *Relation*. The result would be a reference to a *Relation*

RoleName = A

Type = *Relation* instance of type *ObjectClass*

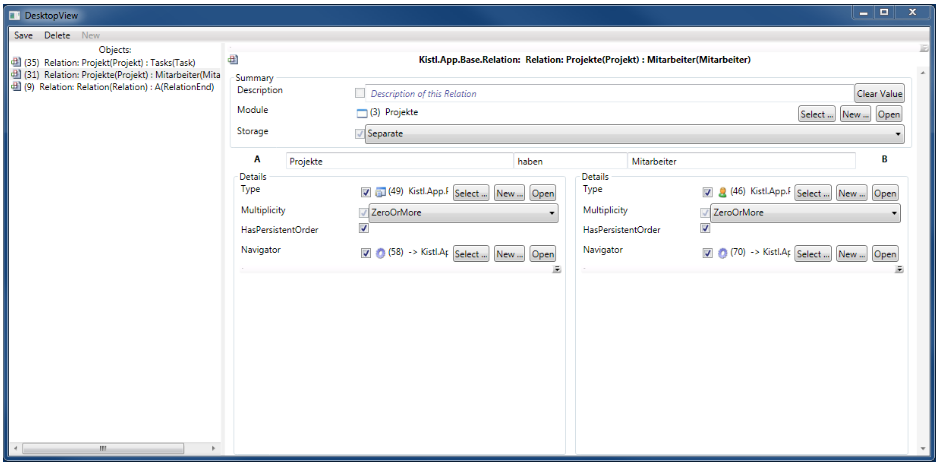


Figure 2.9: Editing the *Relation/RelationEnd* relation

Multiplicity and StorageType summary

1:n Storage = MergeIntoB	
A	**B**
A.Nav is a collection Multiplicity = ZeroOrOne	B.Nav is nullable Multiplicity = ZeroOrMore
A.Nav is a collection Multiplicity = One	B.Nav is not nullable Multiplicity = ZeroOrMore

n:1 Storage = MergeIntoA	
A	**B**
A.Nav is nullable Multiplicity = ZeroOrMore	B.Nav is a collection Multiplicity = ZeroOrOne
A.Nav is not nullable Multiplicity = ZeroOrMore	B.Nav is a collection Multiplicity = One

n:m Storage = Seperate	
A	**B**
A.Nav is a collection Multiplicity = ZeroOrMore	B.Nav is a collection Multiplicity = ZeroOrMore

1:1 Storage = MergeIntoA Storage = MergeIntoB Storage = Replicate (Not supported yet)	
A	**B**
A.Nav is nullable Multiplicity = ZeroOrOne	B.Nav is nullable Multiplicity = ZeroOrOne
A.Nav is nullable Multiplicity = One	B.Nav is not nullable Multiplicity = ZeroOrOne
A.Nav is not nullable Multiplicity = ZeroOrOne	B.Nav is nullable Multiplicity = One
A.Nav is not nullable Multiplicity = One	B.Nav is not nullable Multiplicity = One

2.1.3 Additional Metadata

The object model is intended to be very rich and provide the various subsystems with meta data directly from the *ObjectClass*.

This section describes the various pieces of this meta data.

New related objects

A *CreateRelatedUseCase* describes the use case of creating a new object related to the "current" instance. One such use case would be e. g. "create a new *Relation* from the current ObjectClass."

Such use cases are described with *CreateRelatedUseCase* objects:

```
interface CreateRelatedUseCase
{
        string Label;
        Method Action;
        Relation AffectedRelation; // optional
}
```

The *Action* will be called when the user requests an execution of this use case. This method doesn't take any parameters and returns the newly created object. The infrastructure on the client will cause the returned object to be displayed to the user. The business logic should already have filled out the property values according to the use case. The name of the method should start with "Create".

If the optional *AffectedRelation* is specified, one of its ends it must match the *ObjectClass* of the *Method*. This relation can then be used to identify controls in the UI where the action can be placed.

2.2 Modules

2.3 Compound Objects

Lets say there is a *PhoneNumber* Compound Object.

```
class PhoneNumber
{
        string CountryCode;
        string AreaCode;
        string Number;
        string Extension;
}
```

A Person has two phone numbers:

```
class Person
{
        string Name;
        ...
        PhoneNumber  Tel;
        PhoneNumber? Fax;
}
```

Tel is not nullabe, *Fax* is nullable.

2.3.1 Accessing Compound Objects

- If a compound object property is not nullable then the content of the property is always a valid reference.
- If a compound object property is nullable then the content of the property may be null.
- When a compound object property is set the given compound object will be copied.

```
Person p;
string number;

number = p.Tel.Number;
number = p.Fax.Number; // throws ...
   NullReferenceException if Fax is null

p.Tel.Number = "12345678";
```

```
p.Fax.Number = "12345678"; // throws ...
    NullReferenceException if Fax is null

PhoneNumber n;

n = p.Tel; // returnes a reference of the compound ...
    object
n.Number = "87654321"; // changes p.Tel.Number
n = p.Fax; // may be null
n.Number = "87654321"; // changes p.Fax.Number or ...
    throws NullReferenceException if Fax is null

p.Fax = p.Tel; // creates a copy of p.Tel
p.Fax.Number = "87654321"; // changes p.Fax.Number ...
    but does not change p.Tel.Number

p.Tel = null; // throws a ArgumentNullException
p.Fax = null; // sets Fax to null

n = ctx.CreateStruct<PhoneNumber>(); // creates a new...
     PhoneNumber Struct;
n.Number = "12345678";
p.Tel = n; // creates a copy of n
p.Tel.Number = "18273645"; // changes p.Tel.Number ...
    but does not change n.Number
n.Number = "87654321"; // changes n.Number but does ...
    not change p.Tel.Number
```

2.4 Enhancing Kistl's inner workings

2.4.1 Database Providers

2.5 Graphical User Interface

Like other subsystems, the GUI core is designed to be platform independent. Therefore only the *outermost* shell contains toolkit specific code.

2.5.1 Architecture

The GUI is modeled after the Model-View-ViewModel architecture. The *Model* represents the underlying data structures and business logic. It is provided by the generated classes from the actual datamodel.

View Models provide display specific functionality like formatting, transient state holding and implementing the user's possible actions. They always inherit from *Kistl.Client.Presentables.ViewModel*. Common implementations reside in the *Kistl.Client.Presentables* namespace.

Control Kinds are representing a way how *ViewModels* would like to be displayed. For example:

- as a TextBox
- as a DropDownList
- as a CheckBox
- as a RadioButtonList

Control Kinds are simply a sort of enumeration items. They do not provide any services. *ControlKinds* can be put into a hierarchy. That enables the infrastructure to choose another view if a certian *ControlKind* is not implemented in a Toolkit.

Finally, *Views* (editors and displays) are the actual components taking care of showing content to the user and converting the users keypresses and clicks into calls on the view models interface. Views are toolkit[1] specific and reside in the toolkit's respective assembly.

This architecture decouples the actual functionality of the Model and the View Model completly from the inner workings of a toolkit and thereby maximise the reuse of code between different clients.

2.5.2 Plumbing

The three layers are connected through two sets of descriptors and *ControlKinds*. The *ViewModelDescriptor*s contain information about the available View Models and their preferred way of being displayed.

```
public interface ViewModelDescriptor
{
        Kistl.App.GUI.ControlKind DefaultKind;
        Kistl.App.GUI.ControlKind DefaultDisplayKind;
        Kistl.App.GUI.ControlKind ...
            DefaultGridCellDisplayKind;
        Kistl.App.GUI.ControlKind DefaultGridCellKind...
            ;

        string Description;
```

[1]Toolkits are GUI libraries like WPF, GTK# or Windows Forms but can also be implemented by more complex providers such as ASP.NET.

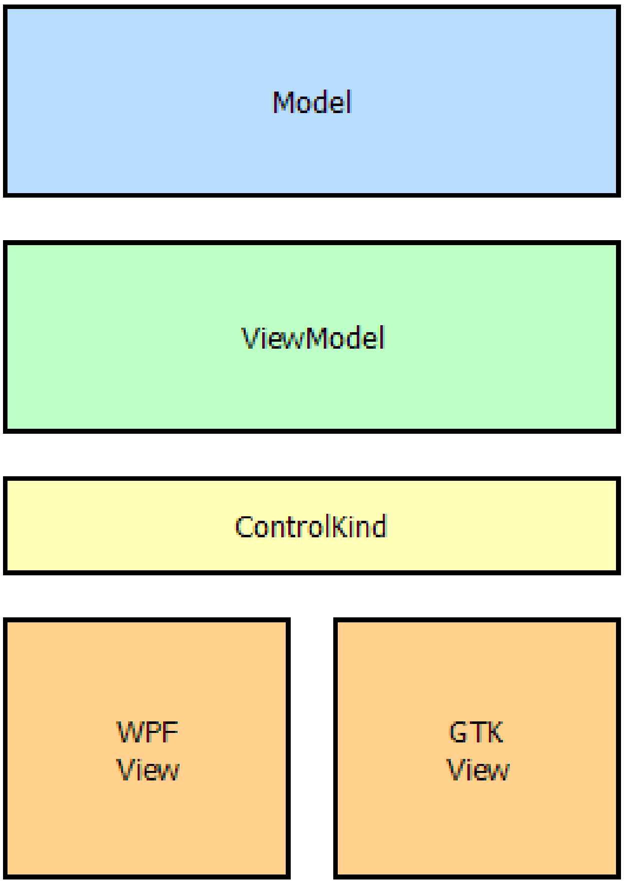


Figure 2.10: Relation of Models, ViewModels, ControlKinds and Views

```
        Kistl.App.Base.TypeRef ViewModelRef;
}
```

*ViewDescriptor*s contain information about controls which are capable of displaying certian *ControlKinds*.

```
public interface ViewModelDescriptor
{
        Kistl.App.GUI.ControlKind ControlKind;
        Kistl.App.Base.TypeRef ControlRef;
        Kistl.App.GUI.Toolkit Toolkit;
}
```

Some implemented ViewModels

DataObjectViewModel represent a complete data object; provide standardised access to properties; provide non-standard ViewModels with additional functionality; selected via *ObjectClass.DefaultViewModelDescriptor*

***BaseValueViewModel*s** represent a specific piece of simple data (strings, DateTimes); Data can be the a Propertiy, MethodResults or simply a Value; Properties and Parameter of Methods selects their *ViewModel* via their *ValueModelDescriptor* Property;

ActionViewModel represent a Method which can be invoked in the UI.

ObjectEditor.WorkspaceViewModel This ViewModel implements all featurs of and Object Editor. This includes Cancel, Save or selecting the current item.

Control Kind

The *ControlKind* specifies the toolkit-independent kind or type of control that should display a given Presentable. While the View specifies the Control Kind it implements the Presentable requests a specific Kind to be displayed via the *PresentableModelDescriptor.DefaultControlKind* value.

In special situations this default value can be overridden. For example, the metadata of a property contains a *RequestedControlKind* which is used instead of the *DefaultControlKind* when present. If there is no View matching the requested Kind, the infrastructure may either fall back to the default control kind, or use a similar control kind from higher up in the hierarchy.

Typical kinds of controls:

WorkspaceWindow the top-level control within which all user interaction happens

SelectionTaskDialog a dialog letting the user select something from a longer list of items

ObjectView display the modeled object in full

ObjectListEntry display the modeled object as item in a list

TextEntry lets the user edit a property as text

IntegerSlider lets the user edit a number with a slider

YesNoCheckbox a simple yes/no checkbox

YesNoOtherText radio buttons allowing one to select either "yes", "no" or a TextEntry field

ExtendedYesNoCheckbox a checkbox with additional text as label

The kind of a control is identified by the *ControlKind*'s class. The hierarchy between different kinds of controls is modeled with inheritance.

Views

Their descriptors list the available Views by Toolkit and which ControlKind they represent. *View Descriptors* can define which *ViewModel*s (or Interfaces to ViewModels) are supported.

2.5.3 Resolving ViewModels

ViewModels are resolved by the *IViewModelFactory*

```
public interface IViewModelFactory
{
    void ShowModel(ViewModel mdl, bool activate);
    void ShowModel(ViewModel mdl, Kistl.App.GUI....
       ControlKind kind, bool activate);

    void CreateTimer(TimeSpan tickLength, Action ...
       action);
    string GetSourceFileNameFromUser(params string[] ...
       filter);
    string GetDestinationFileNameFromUser(string ...
       filename, params string[] filter);
    Toolkit Toolkit { get; }

    // Create Models
    TModelFactory CreateViewModel<TModelFactory>() ...
       where TModelFactory : class;
    TModelFactory CreateViewModel<TModelFactory>(...
       Kistl.API.IDataObject obj) where TModelFactory...
        : class;
    TModelFactory CreateViewModel<TModelFactory>(...
       Kistl.API.ICompoundObject obj) where ...
       TModelFactory : class;
    TModelFactory CreateViewModel<TModelFactory>(...
       Kistl.App.Base.Property p) where TModelFactory...
        : class;
    TModelFactory CreateViewModel<TModelFactory>(...
       Kistl.App.Base.BaseParameter p) where ...
       TModelFactory : class;
    TModelFactory CreateViewModel<TModelFactory>(...
       Kistl.App.Base.Method m) where TModelFactory :...
        class;
    TModelFactory CreateViewModel<TModelFactory>(...
       System.Type t) where TModelFactory : class;

    // IMultipleInstancesManager
```

```
    void OnIMultipleInstancesManagerCreated(Kistl.API...
        .IKistlContext ctx, IMultipleInstancesManager ...
        workspace);
    void OnIMultipleInstancesManagerDisposed(Kistl....
        API.IKistlContext ctx, ...
        IMultipleInstancesManager workspace);
}
```

*ViewModel*s can be created directly if the requested *ViewModel* is known. Some *ObjectClasses* (ObjectClass, Property, Method, Parameter, etc.) can declare a more specific *ViewModel*. Use a more specific *CreateViewModel* overload in such a case.

The most obvious example is *Property*. There is a need for different *ViewModel*s for a *StringProperty* vs. *ObjectReferenceProperty*. Each *ViewModel* for displaying Properties derives from a very basic *BaseValueViewModel*.

IntProperty is displayed by a NullableStructValueViewModel of type int

BoolProperty is displayed by a NullableStructValueViewModel of type bool

DecimalProperty is displayed by a NullableStructValueViewModel of type decimal

StringProperty is displayed by a ClassValueViewModel of type string or MultiLineStringValueViewModel

DateTimeProperty is displayed by a NullableDateTimePropertyViewModel

ObjectReferenceProperty is displayed by a ObjectReferenceViewModel, ObjectCollectionViewModel or ObjectListViewModel

Create a specific *ViewModel* by calling:

```
        mdlFactory.CreateViewModel<WorkspaceViewModel...
            .Factory>().Invoke(ctx);
```

Create a *ViewModel* for a *Property* by calling:

```
        ViewModelFactory.CreateViewModel<...
            BaseValueViewModel.Factory>(prop).Invoke(...
            DataContext,
prop.GetValueModel(Object));
```

Create a *ViewModel* for a *IDataObject* by calling:

```
ViewModelFactory.CreateViewModel<...
    DataObjectViewModel.Factory>(obj).Invoke(...
    DataContext, obj);
```

The *ViewModelFactory* will look up the *IDataObjects* type and tries to find it's *ObjectClass*. Then it looks up the *ViewModelDescriptor* and creates the *ViewModel*.

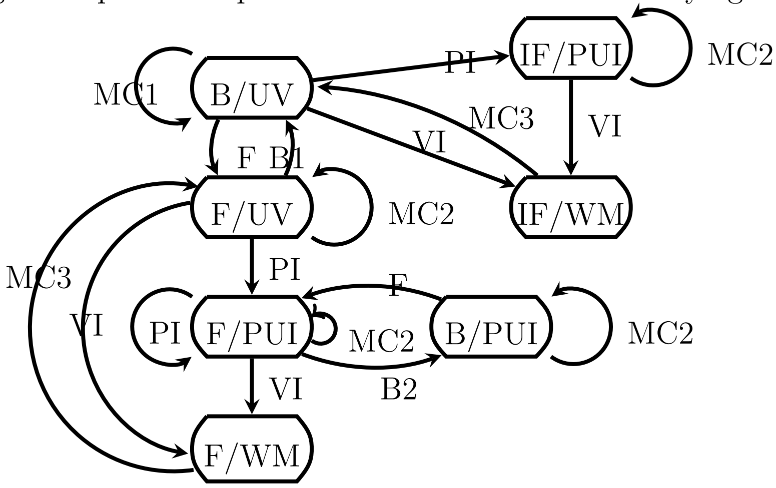
2.5.4 Implementation

Implementing a ViewModel

Implementing a WPF View

2.5.5 States of an Input Control

Input controls are Views receiving input from the user. The following state diagram explains the possible basic states of the underlying *InputViewModel*:



An InputViewModel has two distinct dimensions along which it modifies its behaviour. One is whether or not it currently has the focus[2]. The other dimension is whether or not there are outstanding (partial) modifications to the Value.

B The ViewModel is *blurred*. That is, it does not have the focus. User input will not reach the control.

F The ViewModel is *focused*. That is, it has the focus. The control will be notified of the user's next input.

IF The ViewModel is *implicitely focused*. This states is reached by signalling user input without first signalling a focusing event. It is a convenient shortcut for Toolkits which do not track the concept of focus.

[2]usually the keyboard focus

UV The View should display the *unmodified Value*. This is the common state when no input was received or all input was processed successfully.

PUI The View should display a *partial user input*. If the user cannot input a valid value atomically, the ViewModel will enter this state to keep track of the partial value to add more input from the user until a valid value is formed.

WM The ViewModel is currently *writing to the Model*. Change notifications from the Model in this state are generated by the writing ViewModel and thus must be ignored locally.

There are several events that determine the actual state of the InputViewModel.

F The ViewModel receives *Focus*.

B1 The ViewModel looses *Focus*. Since this signals the user loosing interest in further editing this Value, a change in the FormattedValue is signalled[3] to update the View to the canonical representation of the Value.

B2 The ViewModel looses *Focus*. Since the user has still a pending partial input, the ViewModel expects the user to return and continue editing.

MC1 The Model is *changed* by a third party. This might be due to changes from the business logic or via other ViewModels. In any case the View should be signalled to update its display to the new Value.

MC2 The Model is *changed* by a third party. The user is currently editing the Value, so changes there should be ignored. In most cases this constitutes a programming error, as no code should try to change values while the user is editing them.

MC3 The Model is *changed* by the ViewModel. The View should be signalled to update its display to the new Value. Since this is in reaction to user input, this might not display a canonicalized value

PI The user enters *partial input*. The ViewModel will store this until the user completes or aborts[4] her edit.

VI The user enters *complete input*. The ViewModel accepts the input as valid and will proceed to write it back to the Model.

[3]If the user has not modified the Value, this signalling could be optimized away. Since the canonical representation should always be on display unless the user is currently editing this is not deemed an issue at this time.

[4]Canceling is currently not supported

2.5.6 Asynchronous Loading

Not yet implemented.

To facilitate low-latency user interfaces, the ViewModels should implement a thin proxy layer to delegate all potential blocking operations onto a worker thread. To keep programming this layer easy, there are a few helper classes and a few constraints on the available interface mechanisms as well as a consistent contract over all compliant ViewModels.

There are only three ways to communicate over the *thread gap*:

1. accessing a property
2. calling a *void* function (with not *out* or *ref* parameters)
3. having an *EventHandler* called

All compliant ViewModels provide a *IsLoading* property that signifies whether any background processing is active. While this property is true, any value read from a property may be stale and/or about to be replaced. Changes to the visible value of a property are always reported via the *PropertyChanged* event from the *INotifyPropertyChanged* interface. This should suffice for enabling binding frameworks to show current values to the user: When reading a value from a property, a cached value is returned immediately and optionally a refresh is triggered, which in turn may cause a PropertyChanged event a little bit later.

In the case of time-dependent values, the ViewModel has to take care to establish a periodic refresh timer[5] which triggers PropertyChanged events when new values arrive.

Similarily, methods called on the ViewModel do not actually do their work immediately, but delegate to the background worker thread. Results either show up automatically through changed properties and the PropertyChanged event or via specialized events.

Thread Safety

The ViewModel is designed to be accessed from a single UI thread. Due to the low latency of the public interface, this should pose no problem. The ViewModel internally takes care of all synchronization with the worker thread. Due to the asynchronicity of the underlying data it is quite possible that the values of properties change while a method on the UI thread is currently executing.

[5]Todo: such a timer should be provided by the infrastructure for platform dependent refreshing

Automatic Generation

Due to the restricted set of operations allowed, the proxy can and should be automatically generated, freeing the ViewModel from the intricacies of synchronizing and delegating across thread boundaries. Special needs like callback parameters and time-dependent values have to be communicated via special Attributes.

2.6 Core Kistl Development Environment

2.6.1 Preparing a clean local build

First, it is necessary to have a clean build environment. Use *subst* to create a drive *P:* where your checkout resides in a directory called *Kistl*.

The *!FullReset.cmd* will bring the database and the bootstrapping code up to the current *Database.xml*'s content.

Now the environment is ready for programming.

2.6.2 Merging local and remote changes

When the subversion repository has changed the *Database.xml* while local changes were made to the schema, it is necessary to merge them before comitting.

After fetching and merging the update from the subversion repository, the local *Database.xml* has changes which are not yet in the database. Running *!DeployAll.cmd* updadates the SQL-schema and produces a new set of generated assemblies in the *CodeGenPath*. After testing that the merge was successful, use *GetCodeGen.cmd* to update the working directory with the newly generated bootstrapping code.

Now the working directory is ready for check in.

Chapter 3

Management

3.1 Access Rights

Managing access to objects has to be fine-grained, flexible, fast and robust. To achieve these goals, the Kistl combines an expressive set of rules and generated rights tables.

3.1.1 Roles

To decouple users from the specifics of access management, rights are only conferred to roles, which in turn are assumed by users through their membership in groups and relations.

As an example, consider the project. The project's manager will always have special access rights on the project, regardless of who actually fills this role. Formulating access rights relative to such roles makes them more robust against changes in the people's responsibilities, since when the project manager changes all his rights automatically follow.

Groups

The easiest kind of role is the *Group*. Applicability of this role is only defined through membership in the group and is independent of any other relationships.

Instance-specific roles

Instance-specific roles are conferred through specified relationships with business objects. One example already mentioned is the project's *Manager*. Another the *assigned employees* for a project.

Transitive roles

In some cases the role is not directly associated with the business object under consideration. Instead the connection spans over one or more navigational properties. An example would be access to the set of *Tasks* contained in a project, which is granted by virtue of being an assigned employee of the project.

Nested roles

Finally, roles can be members of groups or roles. This allows for the definition of groups like *all project managers*, which for example might be granted rights on a special set of documents pertaining to management procedures.

3.1.2 Rules

Rules are the way to specify how rights are assigned to roles. There are global, *ObjectClass*-specific and instance-specific rules.

Global rules can be used for granting blanket administrative access or for privileged transfer or analysis processes.

ObjectClass-specific rules are useful for defining class-specific administrators, journalling classes (insert only) and other special cases.

Finally, instance-specific rules allow the most flexible and fine-grained access control, by defining cascading rights through various mechanisms. All such rules specify the set of roles for which the rule is applicable, the set of instances which are affected by this rule and the set of granted rights.

Instance-specific rules

First, an example: all employees assigned to a project are allowed to edit the associated tasks of the project. As an instance-specific rule, this would read: "*Project*: Grant READ,WRITE on *Tasks* to *Employees*."

The set of roles can be specified by a navigator from the current instance to a single Identity or a set of Identities, by a constant set of groups or as a set of necessary access rights to the current instance.

The set of affected instances can be specified by a direct navigator from the current instance, or by using the instance itself.

The rules themselves are defined on the ObjectClass and are then evaluated for each instance.

3.1.3 Implementation

The goal of the implementation is minimal overhead when reading data from the store while providing maintainable and discoverable structures in the underlying database. To achieve this goal, the rules and roles are recursively evaluated until only bare identity-instance pairs with the appropriate access rights remain. This data is subsequently stored in an auxilliary table to each data table. When selecting data from the primary table, an inner join with the access table only returns those instances that have any granted rights at all. Additional filtering can either take place on the SQL level or by passing the access flags through a read-only property to the business logic.

The access tables are implemented as materialized views of table-valued functions. See [1] for implementation details. Due to the structured source of the data all necessary functions, update functions and triggers can be generated together with the schema.

Depending on the specific implementation needs the recursive evaluation can take place when instances are changed, when the user first tries to access the instance, or off-line as a maintenance task.

3.1.4 Extended Example

Here is an extended example, containing two *Projects* and a few people working on them. Every Project has some *Tasks* and working time is recorded in *TimeRecords*.

Furthermore, there are the following rules:

- global: Admins may ALL on ALL
- Project: Manager may READ,WRITE this
- Project: Manager may CREATE,DELETE,READ,WRITE Tasks
- Project: Manager may READ this
- Project: Employees may READ,WRITE Tasks
- Task: READERS may READ TimeRecords
- Task: WRITERS may CREATE TimeRecords
- TimeRecord: Owner may READ,WRITE this

Figure 3.1 shows how the people are distributed on the projects: Alice and Dorothy are managers, Bob is working on both projects, while Charly, Erich and Franz are only working on one of the projects. Gustav is only an administrator.

This leads to the following ultimate access rights for Alice, Bob, Erich and Gustav:

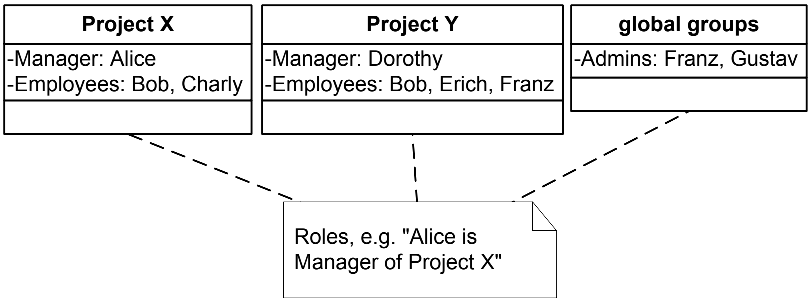


Figure 3.1: Project roles

Person	Objects	Rights
Alice	Project X	READ, WRITE
	Tasks of Project X	CREATE, DELETE, READ, WRITE
	TimeRecords of Project X	CREATE, READ
	TimeRecords with Owner==Alice	READ, WRITE
Bob	Project X and Y	READ, WRITE
	Tasks of Project X and Y	READ, WRITE
	TimeRecords of Project X and Y	CREATE, READ
	TimeRecords with Owner==Bob	READ, WRITE
Erich	Project Y	READ, WRITE
	Tasks of Project Y	READ, WRITE
	TimeRecords of Project Y	CREATE, READ
	TimeRecords with Owner==Erich	READ, WRITE
Gustav	ALL	ALL

3.2 Packaging

This chapter describes the way how modules (packages) are transferred between Kistl installations.

3.2.1 Content of a package

A package contains schema information, meta data, object instances and code. See 3.2 for details. A package can contain one or more modules. A package is a zip file that contains one or more XML files containing meta information and/or data and binary files like icons or assemblies.

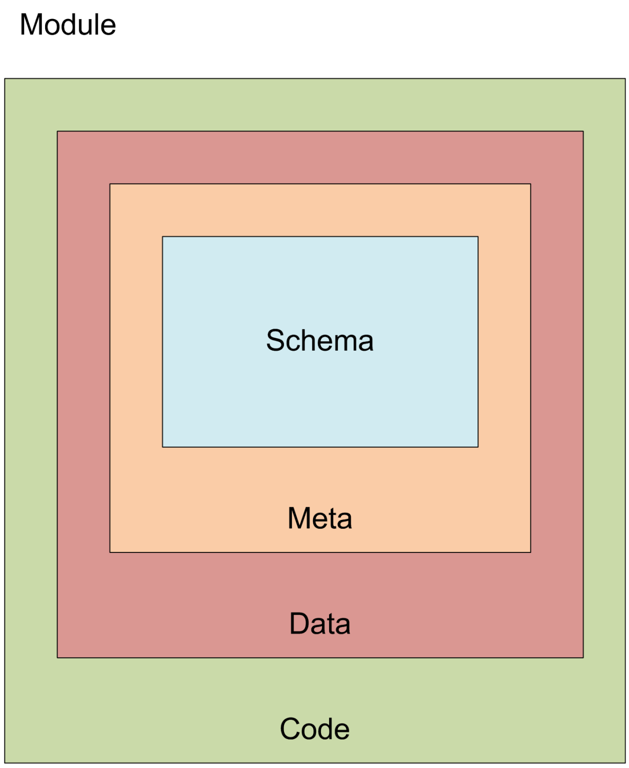


Figure 3.2: Content of a package

Schema

Schema contains all information needed by the SchemaManager **??** to create or update the database. Schema objects are:

DataType All object classes and structs except enumerations and interfaces. *To be implemented!*

Property All properties

Relation All relations

RelationEnd All relation ends

DefaultPropertyValue All known default property values (IntDefaultValue, . . .)

Constraint All known constraints (NotNullable, . . .)

Schema is a subset of meta data. See 3.2.1. Currently there is no way to extract schema information without meta information.

Meta

Meta data contains all information needed to describe a Module. That are *ObjectClasses*, *TypeRefs*, *Module* informations, *icons*, *ViewDescriptors* and so on. Meta data is a superset of schema data 3.2.1. These objects are:

Module The module object

ObjectClass_implements_Interface_RelationEntry All object class interface relations

Method All Methods

BaseParameter All parameter of a method

MethodInvocation All method invocations

PropertyInvocation All property invocations (getter/setter)

Assembly All Assemblies (meta information only, not code)

TypeRef All type references

TypeRef_hasGenericArguments_TypeRef_RelationEntry All generic arguments of type references.

Icon All Icons (descriptor only, no binary data)

PresentableModelDescriptor All presentable model descriptors

ViewDescriptor All view descriptors

DAVID: please add more GUI objects here. And please do it also in *PackagingHelper.GetMetaObjects(...)*

Additionally all unknown DefaultPropertyValues and Constraints belongs to meta data. Meta data can be extracted with the publish command 3.2.2.

Data

A package can also contain additional data. Only object classes that implements *IExportable* will be exported. N:M relations between object classes that implements *IExportable* are also exported. Currently all objects of a specific module are exported. Data can be extracted with the export command 3.2.2.

Code

Finally a package consists of code in form of assemblies. These assemblies are referenced by *Assembly* objects.

3.2.2 Processes

Export

Exporting is the process of saving objects in XML files. Only object classes that implements *IExportable* will be exported. N:M relations between object classes that implements *IExportable* are also exported. Currently all objects of a specific module are exported.

Command line for exporting objects:

```
Kistl.Server <configfile.xml> -export <destfile.xml> ...
    <namespace> [<namespace> ...]
```

The namespace is used to identify a module. *TODO: Work in progress. This is not the best solution!*

This example will export all objects of the project management module:

```
Kistl.Server -export Export.xml Kistl.App.Projekte
```

This example will export *all* meta data of *all* modules:

```
Kistl.Server -export Export.xml Kistl.App.Base Kistl....
    App.GUI
```

This example will export the whole database:

```
Kistl.Server -export Export.xml *
```

Import

Importing is the inverse process of exporting 3.2.2. Objects are imported by the following rules:

1. If an imported object already exists in the target database then the object will be overridden
2. New objects are added
3. No object is deleted if it's not contained in the package

Command line for importing objects:

```
Kistl.Server <configfile.xml> -import <sourcefile.xml...
  >
```

Publish

Publishing is a special case of exporting 3.2.2. Only meta data 3.2.1 of a given module will be exported. Additionally only properties of the Kistl.App.Base and Kistl.App.GUI module are published.

Command line for publishing modules:

```
Kistl.Server <configfile.xml> -publish <destfile.xml>...
   <namespace> [<namespace> ...]
```

The namespace is used to identify a module. *TODO: Work in progress. This is not the best solution!*

This example will publish the project management module:

```
Kistl.Server -publish Meta.xml Kistl.App.Projekte
```

This example will publish all modules:

```
Kistl.Server -publish Meta.xml *
```

Deploy

Deployment is the inverse process of publishing 3.2.2. It also has different rules (see importing 3.2.2). These Rules are:

1. If an imported object already exists in the target database then the object will be overridden
2. Only properties of the the Kistl.App.Base and Kistl.App.GUI module are overriden.
3. New objects are added
4. Any object that is not contained in the packed will be deleted

Command line for importing objects:

```
Kistl.Server <configfile.xml> -deploy <sourcefile.xml...
  >
```

The database schema is not updated. Also no code is generated. This has to be done in an extra step.

3.3 Deployment

This section describes the possible deployment strategies.

■ Note: this section is a subject to change ■

3.3.1 Continuous Integration Server

The *Continuous Integration Server* does a publish in a directory structure. This directory structure is transformed by a *fetch* script into the designated directory structure.

It's not recommended to use the *Continuous Integration Servers* structure directly. The next section will discuss the fetching process and the designated directory structure.

3.3.2 Fetching and Destination Directory Structure

The fetching script is responsible for

- Creating the directory structure
- Fetching all Assemblies and putting them in the right destination directory
- Copying all app configuration files to the right directories

The directory structure on the deployment server should look like this:

- AppConfigs
- bin
  - Bootstrapper
  - Client
    - Client
      - App.ZBox
      - Core
      - Core.Generated
      - WPF
      - WinForms
    - Common
      - App.ZBox
      - Core

      - Core.Generated
    - Kistl.Client.WPF.exe
    - Kistl.Client.WPF.exe.config
    - Kistl.Client.Forms.exe
    - Kistl.Client.Forms.exe.config
  - Server
    - Common
      - App.ZBox
      - Core
      - Core.Generated
    - Server
      - App.ZBox
      - Core
      - Core.Generated
      - EF
      - EF.Generated
      - NH
      - NH.Generated
    - Kistl.Server.Service.exe
    - Kistl.Server.Service.exe.config
- Configs
- DocumentStore
- inetpub
  - App_Data
  - App_GlobalResources
  - App_Themes
  - bin
  - Bootstrapper
  - Common
    - App.ZBox
    - Core
    - Core.Generated
  - Server
    - App.ZBox
    - Core

    * Core.Generated
    * EF
    * EF.Generated
    * NH
    * NH.Generated

- logs
- Packages
- deploy.ps1
- fetch.ps1

Root Directory

deploy.ps1 The deployment script, responsible for upgrading the servers database

fetch.ps1 The fetch script, responsible for creating the directory structure and fetching all assemblies and configuration templates

AppConfigs

This directory contains all app configuration files needed by the executing assemblies. E.g. in the *Kistl.Server.Service.exe.config* all WCF Service stuff can be configured. In *Kistl.Client.WPF.exe.config* all WCF Proxy stuff can be configured.

The fetch script will copy those configuration files into the desired directories, right beside their executalables.

So this is the place where all app specific configuration has to be defined.

Besides WCF stuff, those configuration can be also found in those files:

- log4net
- WCF
- Assembly Bindings
- Database provider registration

■ Note: The web.config is not located here, but this may change in the future ■

Configs

This directory contains all ZBox configuration files. They are located by the executalable by probing

- the given command line parameter
- then the *zenv* environment variable plus executalable name
- then each directory up to the *Configs* directory, still with the executalable name
- at the end by looking for *DefaultConfig.xml* in the configs directory

bin

The bin directory contains all assemblies used by ZBox, divided by theire use (Bootstrapper, Client or Server). The *Client* and *Server* directories contains sub directories to split Client/Server and Common parts. Those directories have for each Application (=ZBox Module) and Modules (not ZBox Modules!) a individual sub directory.

The naming of Application (=ZBox Module) should be:

App.<ModuleName>

Core contains all core assemblies like Kistl.API and all other assemblies that are referenced by default (log4net e.g.).

Core.Generated contains all genereated code. Code Generation is done by the *Continuous Integration Server* so those assemblies will be copied by the fetch script.

■ Note: This is a topic of discussion ■

Bootstrapper contains only one executalable. The Boostrapper itself which is downloading the whole client application with its configuration from an HTTP Server via REST.

3.3.3 Deployment on a Linux Server

On Linux the following packages are needed:

- Mono 2.10 - master
- Apache 2
- mod_mono
- many other…

Apache configuration

- Install Apache
- install mod_mono
- turn off KeepAlive - the .NET HTTP Client won't talk to Apache correctly
- enable/configure your security model
- ensure that the *TrustedBasicAuthenticationModule* is enabled.

The Apache Server is responsible for authentication. The ZToolbox server trusts Apache and uses the passed identity (through the HTTP Header).

3.3.4 Deployment on a Windows Server

Bibliography

[1] Dan Chak, Materialized Views that Really Work, 2008, The PostgreSQL Conference, http://www.pgcon.org/2008/schedule/events/69.en.html

List of Figures