dasz.at
Benutzbare Technologie

Kistl Guide

The Hitchhiker's guide to the Kistl galaxy

Contents

Chapter 1

Introduction

Kistl is a programming 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.

Chapter 2

Programming

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

2.1 Objects

2.2 Modules

2.3 Enhancing Kistl's inner workings

2.3.1 Database Providers

2.4 Graphical User Interface

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

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

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

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.

2.4.2 Plumbing

The three layers are connected through two sets of descriptors. The *PresentableModelDescriptor*s contain information about the available View Models and their preferred way of being displayed. The *ViewDescriptor*s link *PresentableModelDescriptor*s with the controls capable of displaying them.

Presentable Model Descriptors

Currently there exist three major types of View Models.

***DataObjectModel*s** represent a complete data object; provide standardised access to properties; provide non-standard Views with additional functionality; selected via *ObjectClass.DefaultPresentableModelDescriptor*

***ValueModel*s** represent a specific piece of data; representations of Properties are selected via *Property.ValueModelDescriptor*; method results are currently created directly via *Factory.CreateSpecificModel*

other Presentables represent objects in the View which do not have persistent representations, like dialogs or wizards; always created by calling *Factory.CreateSpecificModel*

View Descriptors

These descriptors list the available Views by Toolkit and which subset of Presentables they are able to work with.

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

Control Kinds can also be used to configure the actual control. This possibility should be used sparingly as a control should instead seek to infer its configuration from the underlying Presentable. For example, an integer slider control should lookup the minimal and maximal allowed values in the underlying *IntegerRangeConstraint* while an ExtendedYesNoCheckbox has no other place to retrieve the new label.

2.5 Core Kistl Development Environment

2.5.1 Preparing a clean local build

First, it is necessary to have a clean build environment. Use *subst* to create a drive *P:* where your subversion checkout resides in a directory called *Kistl*. Be sure to delete the contents of your *CodeGenPath*[2] as well as any old artifacts from your working directory[3]. Clean your database by calling the *!KillDatabase.cmd* in the project's root.

Now build the solution. This will create the necessary artifacts to bootstrap the database.

The *!CreateDatabase.cmd* will now use the freshly built server executable to update the (empty) schema in the database to the current *Database.xml*. Now all tables are ready to be filled with data.

Finally you can run *!DeployAll.cmd* to populate the database and generate the frozen and client object assemblies.

Now the environment is ready for programming.

[2] Typically *C:\temp\KistlCodeGen*

[3] Use e.g. TortoiseSVN's "Show ignored files" option to find cruft

2.5.2 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.

List of Figures