# Simio API Note: Data Use with DirectConnect

Creation (Nov 2019 Dhouck) Last Update (Nov 2019)

Contents

[Simio API Note: Data Use with DirectConnect 1](#_Toc25228073)

[DesignContext AddIn 4](#_Toc25228074)

[AccessDesignContext Process Step 5](#_Toc25228075)

[The Design-Time Singleton Pattern 7](#_Toc25228076)

[AccessDesign Example Project 9](#_Toc25228077)

DirectConnect is a technique that is used for custom databinding, primarily for RPS implementation. It was built and used before the Unified DataBinding was conceived.

It works by using a combination of techniques, technology, and convention.

Specifically, it is combination of specially configured Simio Project along with .NET User Extensions:

* There are configuration Tables such as Table Export Config that are used by a
* UserExtension, which taps into events such as OnModelLoaded that
* Employ Simio APIs to import and export data from data Tables.

The class DirectConnectHelperAddIn implements IModelHelperAddIn and has logic attached to the ModelSaved event. During this event:

* The method SaveDataToSql is called to display the SaveToDatabaseWindows, where options of saving tables and/or logs are presented to the user.
* When selected the SaveSimioTablesToDB and/or SaveSimioLogsToDB are called.

UserExtension DirectConnect (.Net DLL)

Creates AddIns Properties for Model

ConnectionString

A screenshot of a cell phone

Description automatically generated

Simio Tables (Created Manually)

1. Table Export Config
2. Log Export Config
3. Table Export Exclude For Update

It relies on some specific tables. For example Table Export Config:

As it start, it calls the DefineSchema method to define three properties:

1. Connection String
2. Connection Timeout (in seconds)
3. DateTimeFormat

It is built to react to these events:

* Model Saved
* Tables Importing, and
* TablesImported

While the Simio Engine is in run-time, there is a live design-time object. In other words, it is not destroyed during run-time.

So, there is a work-around, and that is what we’ll describe here.

*Note: It was pointed out to me that a ‘correct’ solution would be to add a readonly Interface, which would be a change to the API. So, send in your letters to request such a thing to Simio support if you wish.*

There are two parts to this work-around:

1. A lightweight Add-In that squirrels away a copy of the IDesignContext object in a singleton (for more info about the Singleton pattern, please refer to Wikipedia).
2. A lightweight User Extension for a ProcessStep which demonstrates accessing this IDesignContext.

The bad news is that you must remember to execute the Add-In after you load your model and before running the model. But such is the nature of work-arounds.

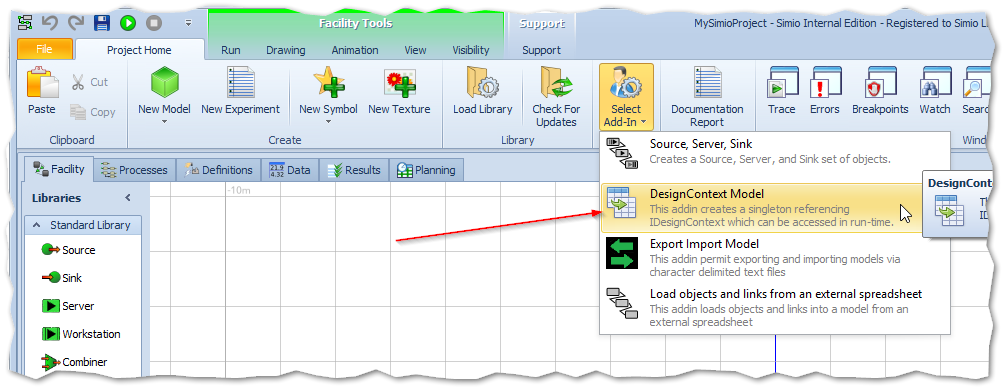
We’ll take this in three sections. One for the Add-In, the second for the User Defined Process Step, and thirdly a Simio Project that employs the AddIn and User Extension Step.

An important note:

*Note: This work-around provide you with a reference to the IDesignTime context at run-time. It is very important that you treat this as a read-only access. Writing to the IDesignTime object at run-time would be a very bad thing.*

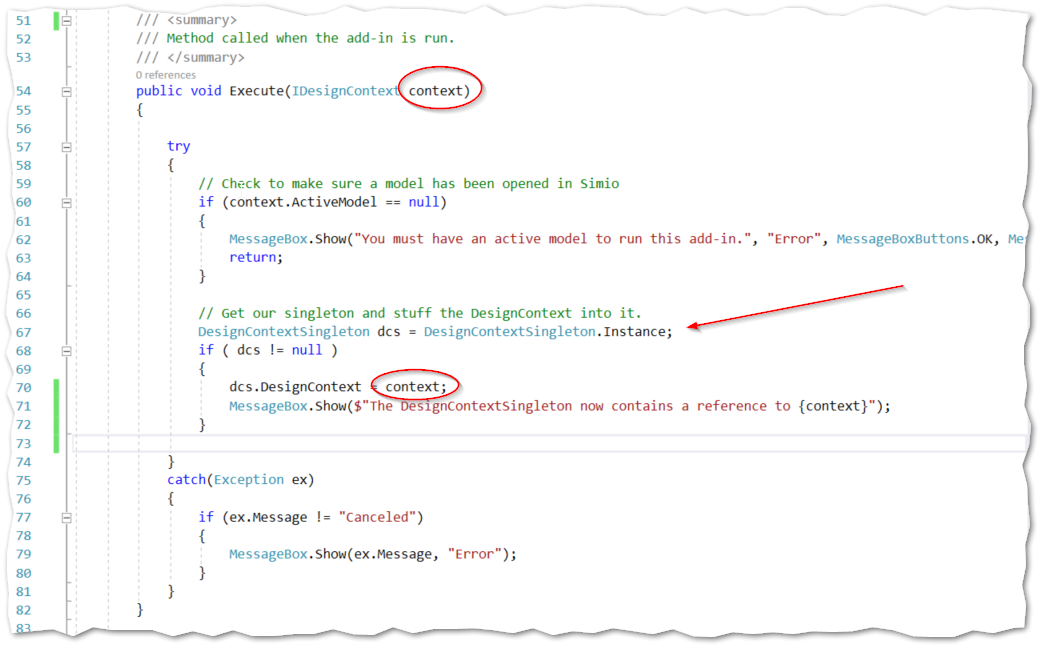
## DesignContext AddIn

When we are done, the Add-In will appear like this:



Selecting it will bring up a message that a singleton object has been created that references the DesignContext.

The construction of this code is trivial. In the Execute method it accesses the Singleton, and the set the IDesignContext that was passed in as the argument:



## AccessDesignContext Process Step

The code for the Process Step is in the project AccessDesignContext.

The schema for the step includes a RepeatGroup (which is included only as an example and not used) and a State Property (which is just used to display something on the running Model)..



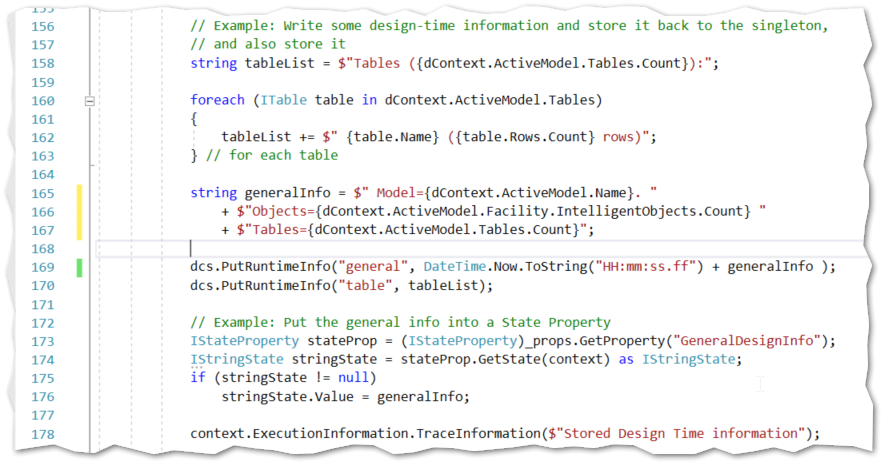
The real meat is the Execute, which:

1. Get the Singleton context
2. Grabs some info from the design context.

Getting the context. Note the error if we forgot to select our AddIn.



Storing some Design-Time summaries back into the Singleton, and also stuffing it into our State Property “GeneralDesignInfo”.



### The Design-Time Singleton Pattern

The Singleton code is very simple, but I complicated it with the ability to store other values at run-time. The reasoning is that we may want to use one type of AccessDesignStep to do a one-time collection of information useful to us, and others to use this information. It also demonstrates the power of the Singleton pattern.

The basic Singleton portion (the Dictionary is not a part of it):



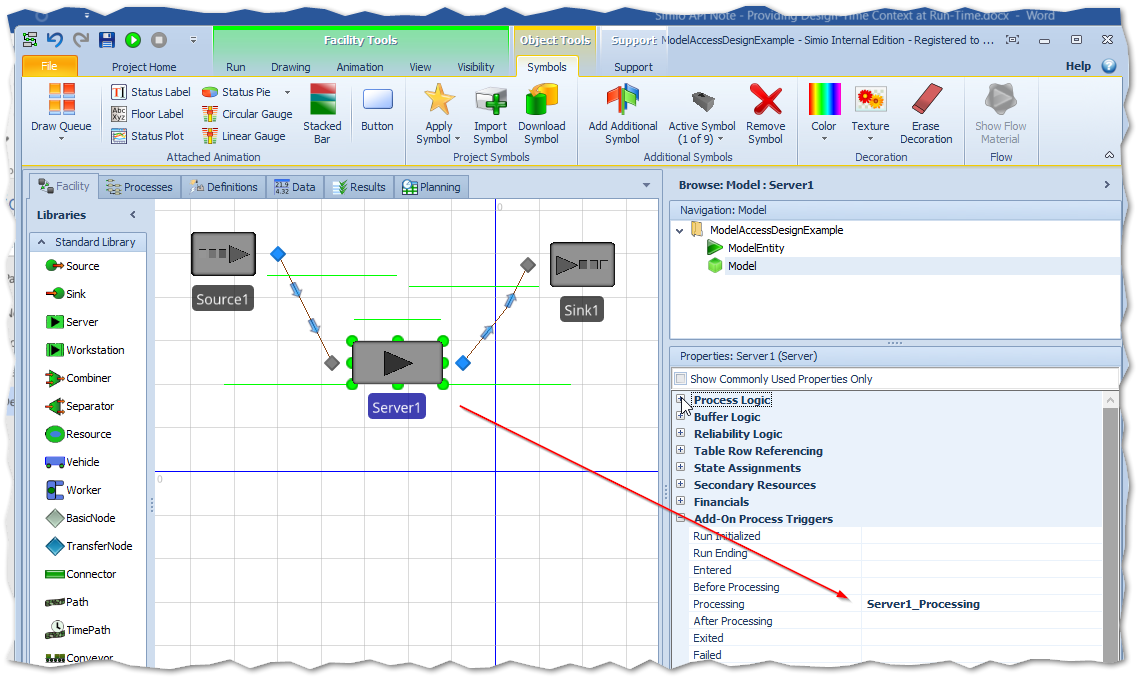
The dictionary storage part:



## AccessDesign Example Project

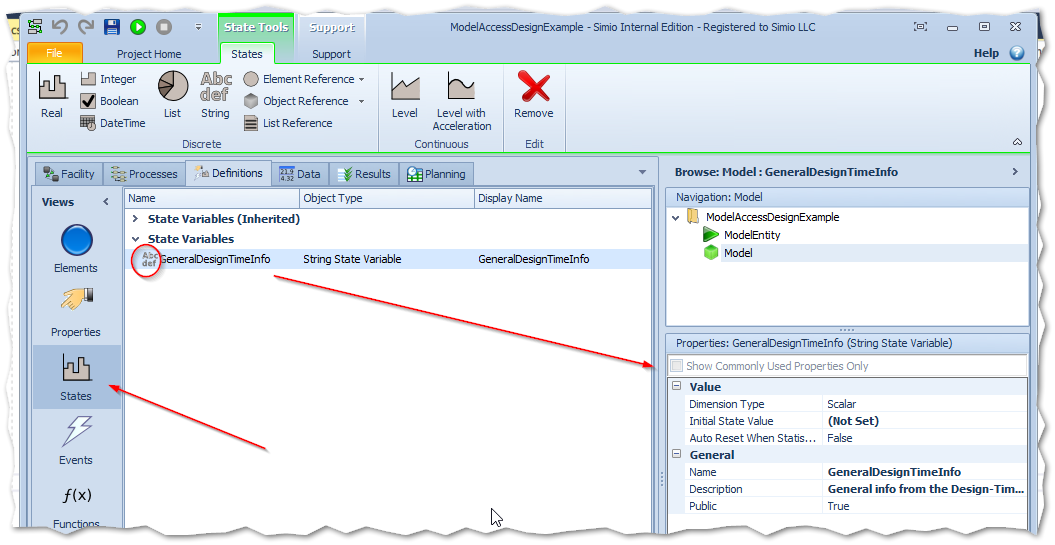
The sample project was created with a simple Source -> Server -> Sink configuration.

A process step was added to the Server and was arbitrarily added to its “Processing” Trigger.

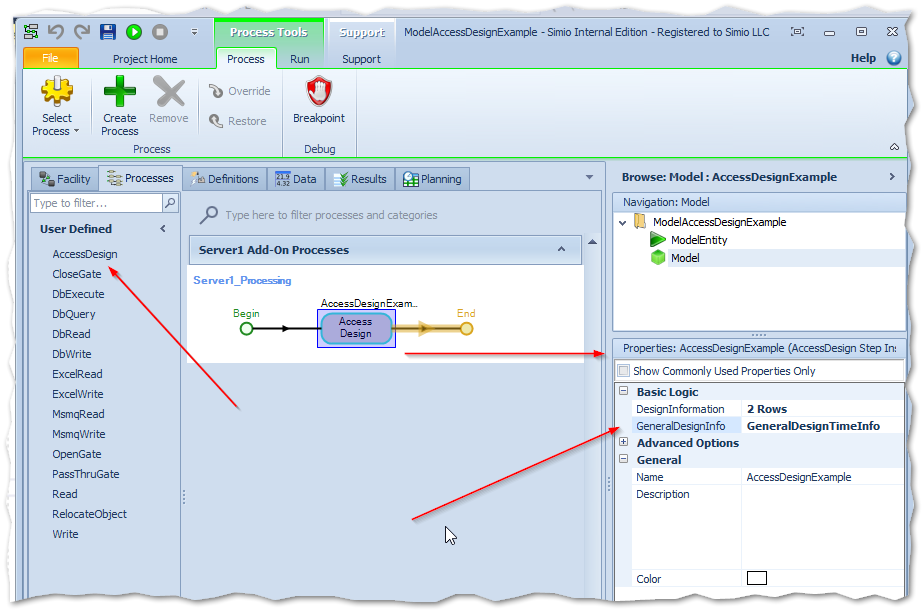


The Process step AccessDesign was added using the “User Defined” collection.

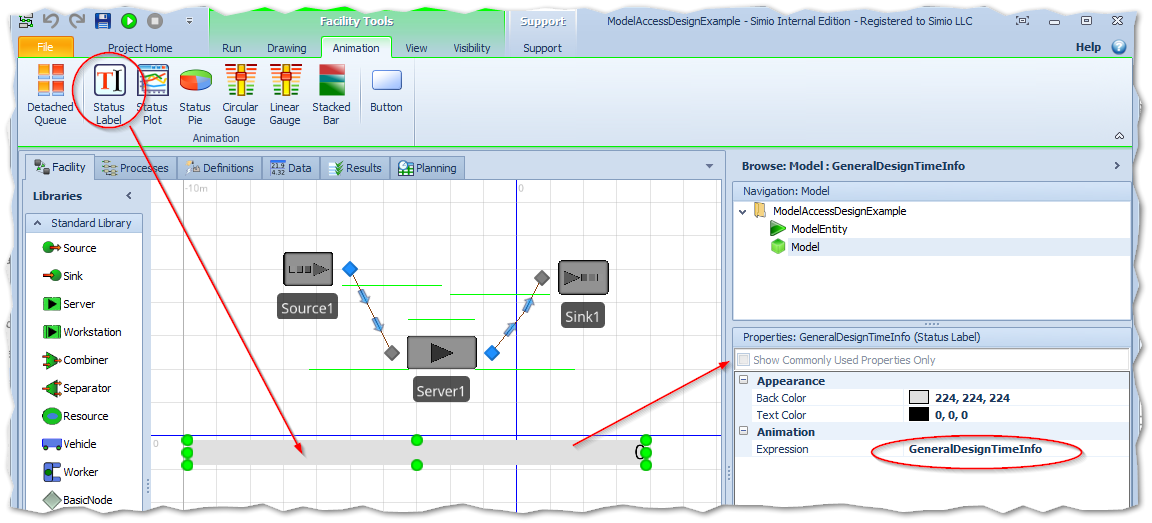
Our AccessDesign Process Step includes a Property that is a State variable called GeneralDesignInfo, just so we could display something. So we’ll have to wire it up to a State variable of type String and is called GeneralDesignTimeInfo in our example project.



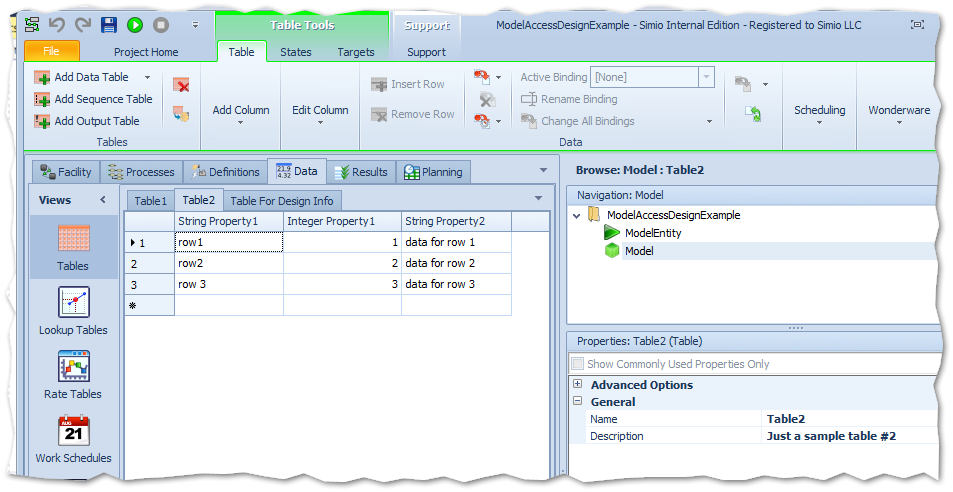
Now we’ll reference this new State variable in our GeneralDesignInfo property:



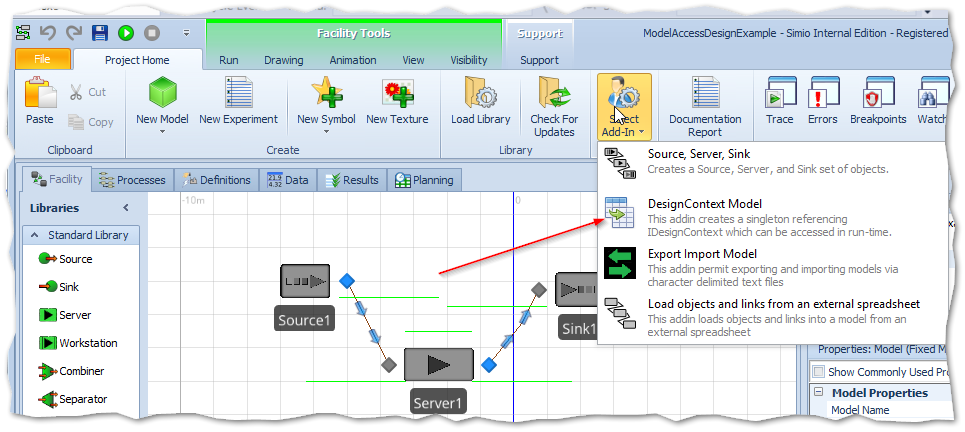
We’re almost ready to run, but let’s add a label to display that GeneralDesignInfo property, …



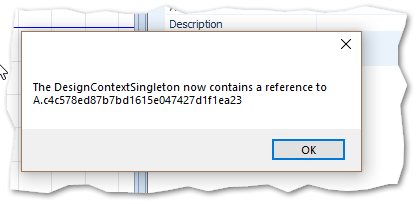
… and add some tables to our project so that we can convince ourselves that we are reporting design-time information correctly.



And now run the Project and make sure that you first select the Add-In:



Which causes a popup that will look something like this:



… and press “Run”:

