SQL Yoga To-Do Application

Introduction

Introduction: What We Are Going to Create

SQL Yoga extends the simplicity of Revolution to databases by allowing you to treat your database like an object. Stop wrestling with SQL and see how easy database integration can be:

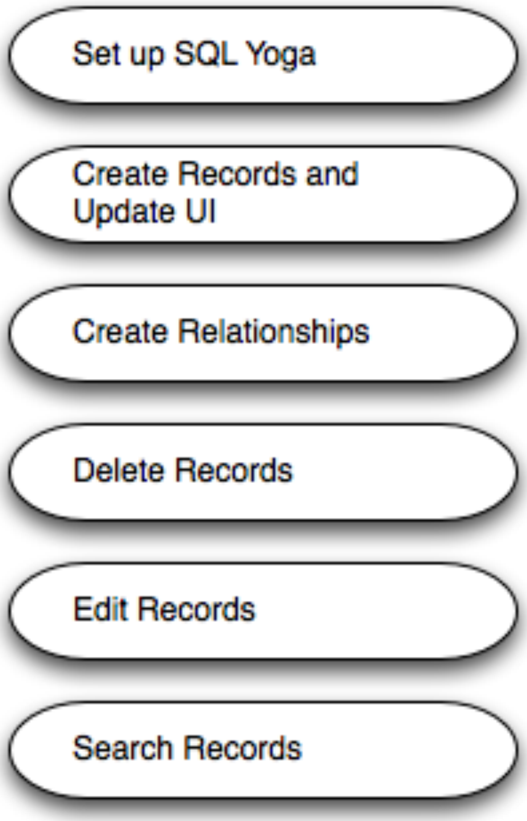
• Set properties rather than writing SQL
• Define searches using english words rather than arcane wildcard symbols
• Manipulate arrays, not cursors
• Easily generate searches from complex search UIs
• Tap into database table relationships to simplify code

To get the latest information, learning materials and SQL Yoga library visit the SQL Yoga web page:

http://www.bluemangolearning.com/revolution/software/libraries/sql-yoga/

Screencast Introduction

The Steps You Will Go Through



Before You Get Started

This manual will walk you through how to create a To-Do application using SQL Yoga. As we walk through creating the application you will be shown how to use SQL Yoga to get things done.

We will use the GLX Application Framework as the foundation for the application. Before you continue you should download the To-Do Application tutorial files.

IMPORTANT: SQL Yoga Is Built On Top of RevDB And the Valentina (V4REV) External

SQL Yoga is built on top of RevDB and the Valentina external (V4REV). In order to use SQL Yoga you must have a version of Revolution that supports database access or own a copy of the V4REV database external.

This sample is built using SQLite.

Download To-Do Sample Application

You can download the To-Do Application tutorial files here:

http://www.bluemangolearning.com/download/revolution/sql_yoga/sql_yoga_todo_app.zip

Start a Fresh Revolution Session

Start up Revolution and don't open any other stacks. This ensures that you don't have any stack name conflicts or conflicts caused by previous SQL Yoga tests while working on the tutorial.

Have A SQLite Database Editor At The Ready

When developing a database application it can be handy to have an application available for managing the database you are working on. The are numerous SQLite tools available. Here are links to two:

SQLite Manager for FireFox (free)
https://addons.mozilla.org/en-US/firefox/addon/5817

SQLiteManager by sqlabs (paid)
http://www.sqlabs.com/sqlitemanager.php

Optional: Install GLX Application Framework Plugin

If you want to look into how the sample application was configured using the framework then the GLX Application Framework plugin is required. This is optional and is not required to work through the tutorial. If you have not already installed the GLX Application Framework plugin you can get it at the following url:

http://www.bluemangolearning.com/download/revolution/glxapp_framework/glx_application_framework.zip

After downloading the framework distribution follow these instruction to install the GLX Application Framework plugin:

http://revolution.screenstepslive.com/spaces/revolution/manuals/glxapp/lessons/5489-Installing-the-Plugin

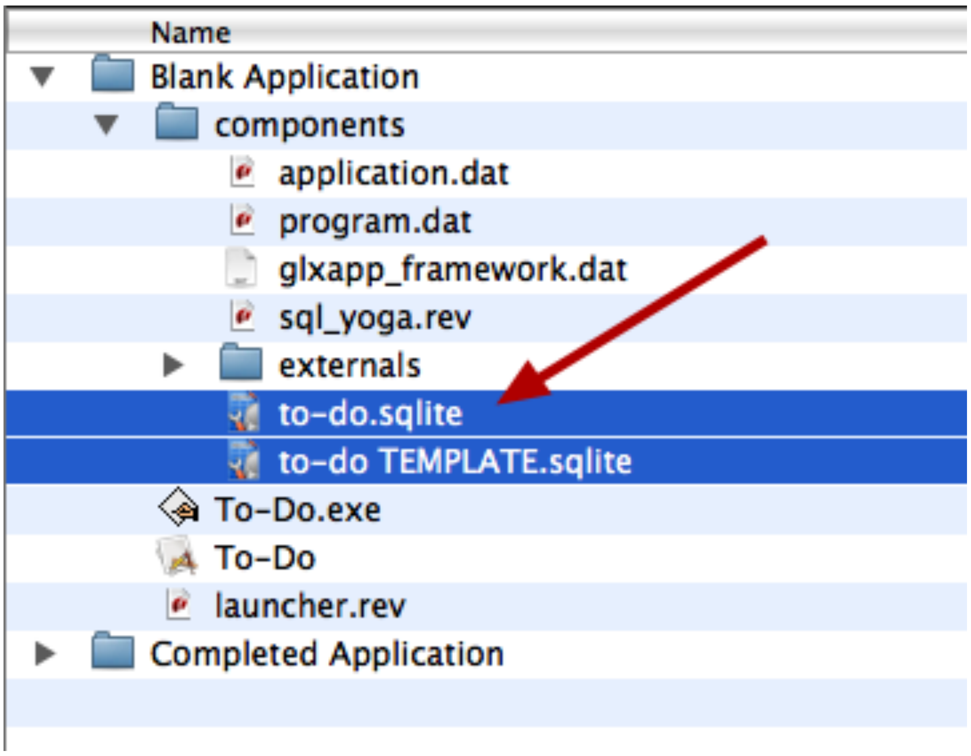
The To-Do Application Database Schema

The database that the To-Do application uses a SQLite database for storing information. The database that has four tables:

1) projects
2) todo_items
3) people
4) people_todo

Projects can have many todo_items and any number of people can be linked with any number of todo_items through the people_todo table.

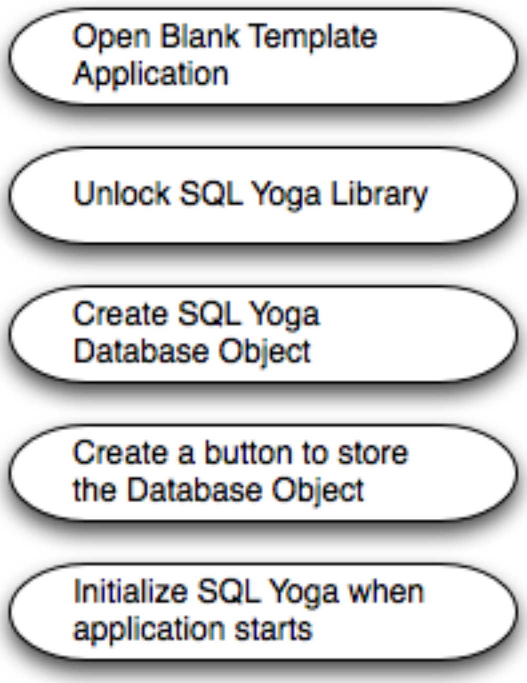
You will find the **to-do.sqlite** database file in the ./components/ folder in the tutorials distribution. The **to-do TEMPLATE.sqlite** file serves as a backup if you need an empty database to start with.



Getting Started

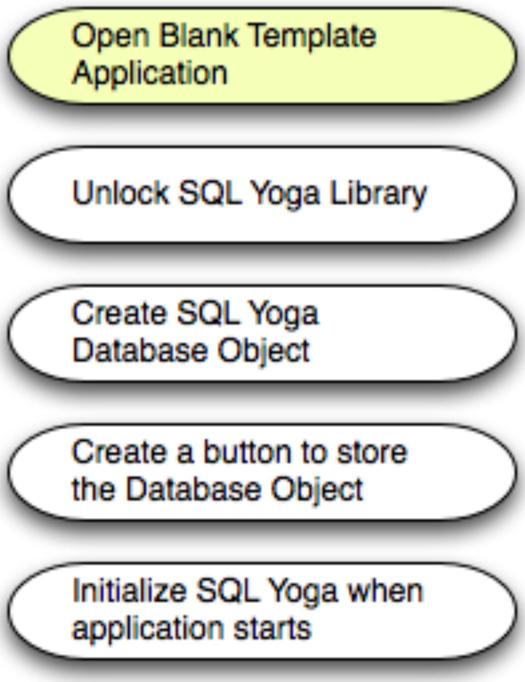
Chapter Overview

Here are the major subjects we are going to cover in this chapter.


Open Blank Template Application
Unlock SQL Yoga Library
Create SQL Yoga Database Object
Create a button to store the Database Object
Initialize SQL Yoga when application starts

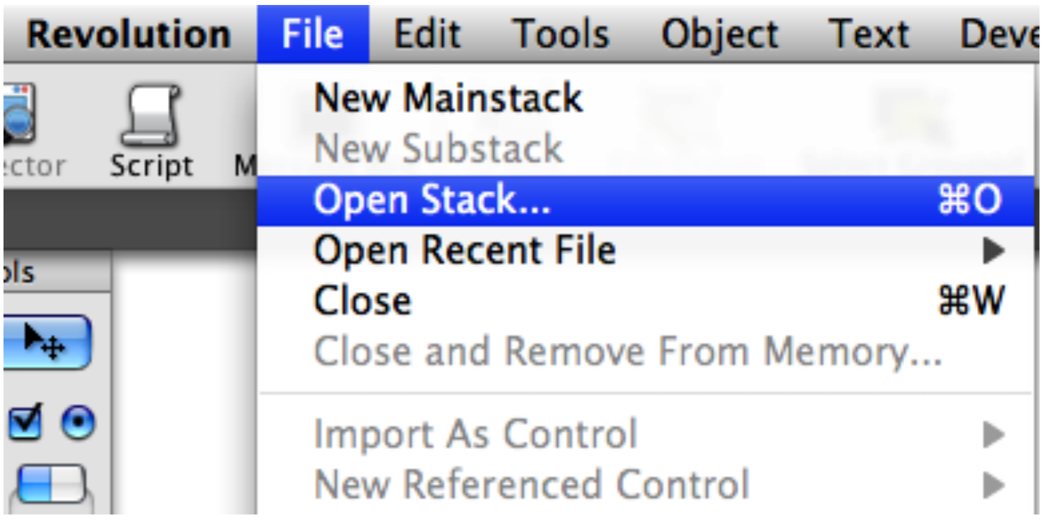
Open Blank Application Template

Now that you have downloaded the tutorial files we can get started. The first thing you need to do is open the application stored in the **Blank Application** folder that came with the To-Do distribution. This folder contains a pre-configured GLX Application Framework application that you will use throughout the rest of this manual.

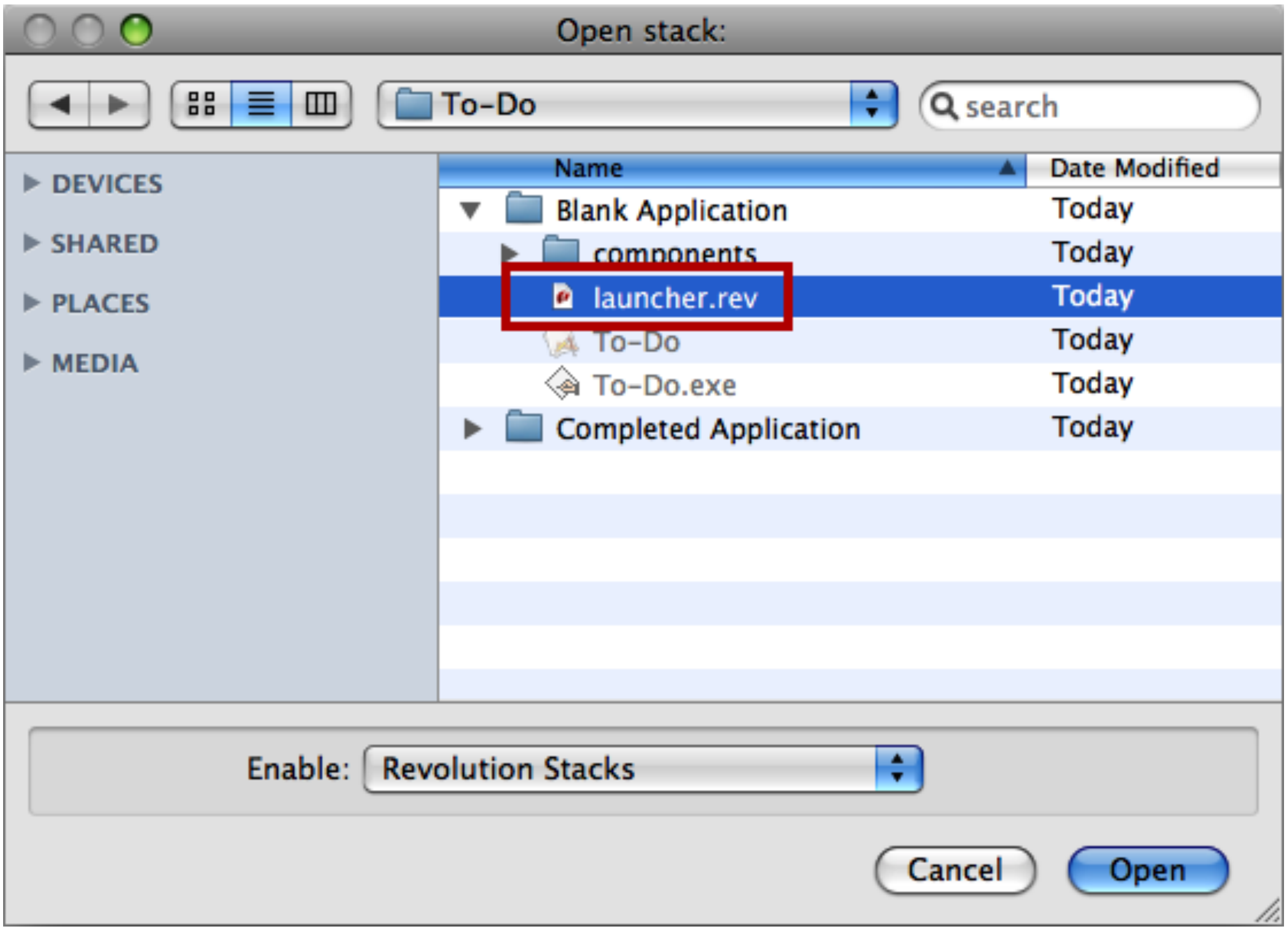


Open Blank Template Application
Unlock SQL Yoga Library
Create SQL Yoga Database Object
Create a button to store the Database Object
Initialize SQL Yoga when application starts

Open Stack

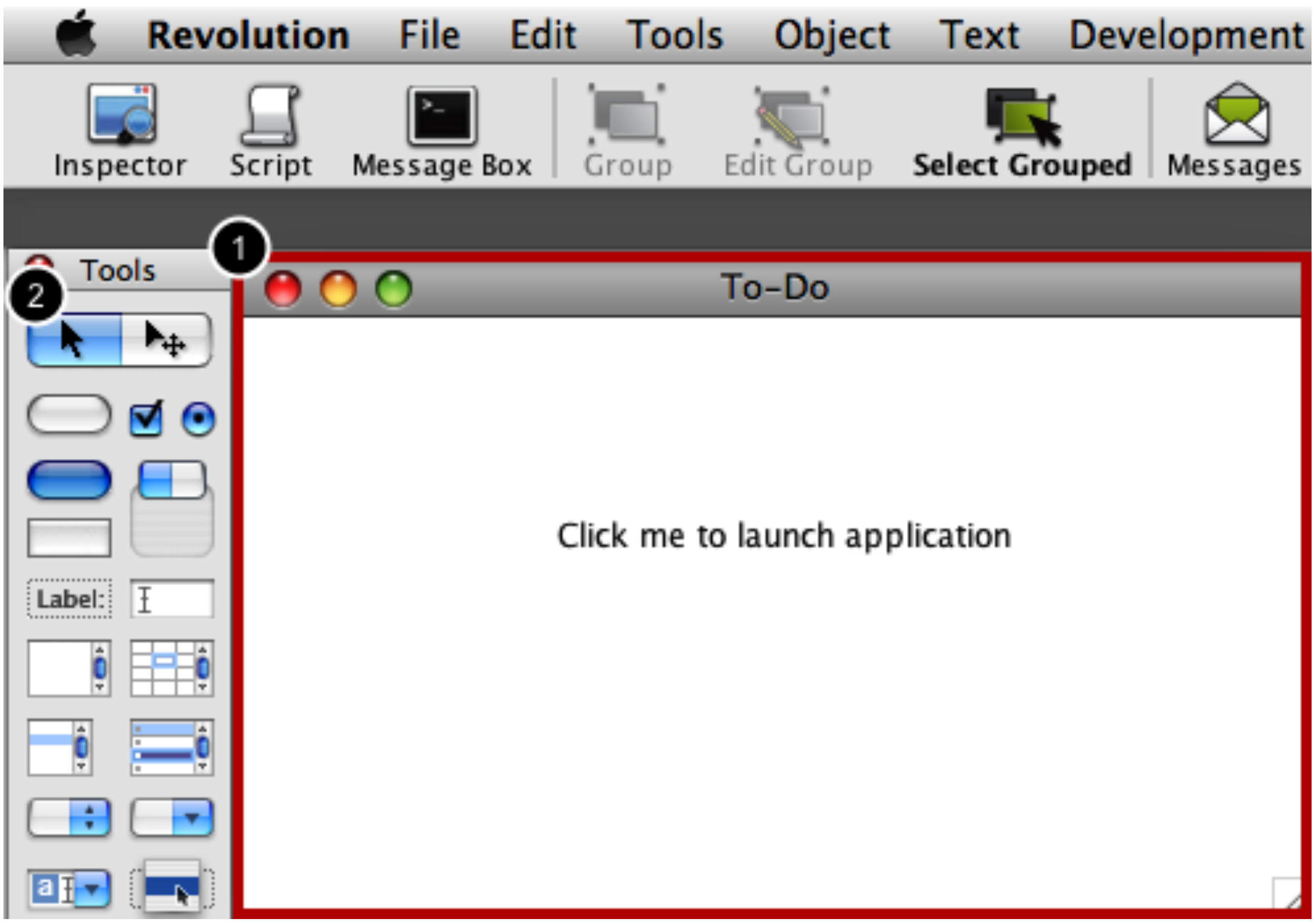


Choose **File > Open Stack...**

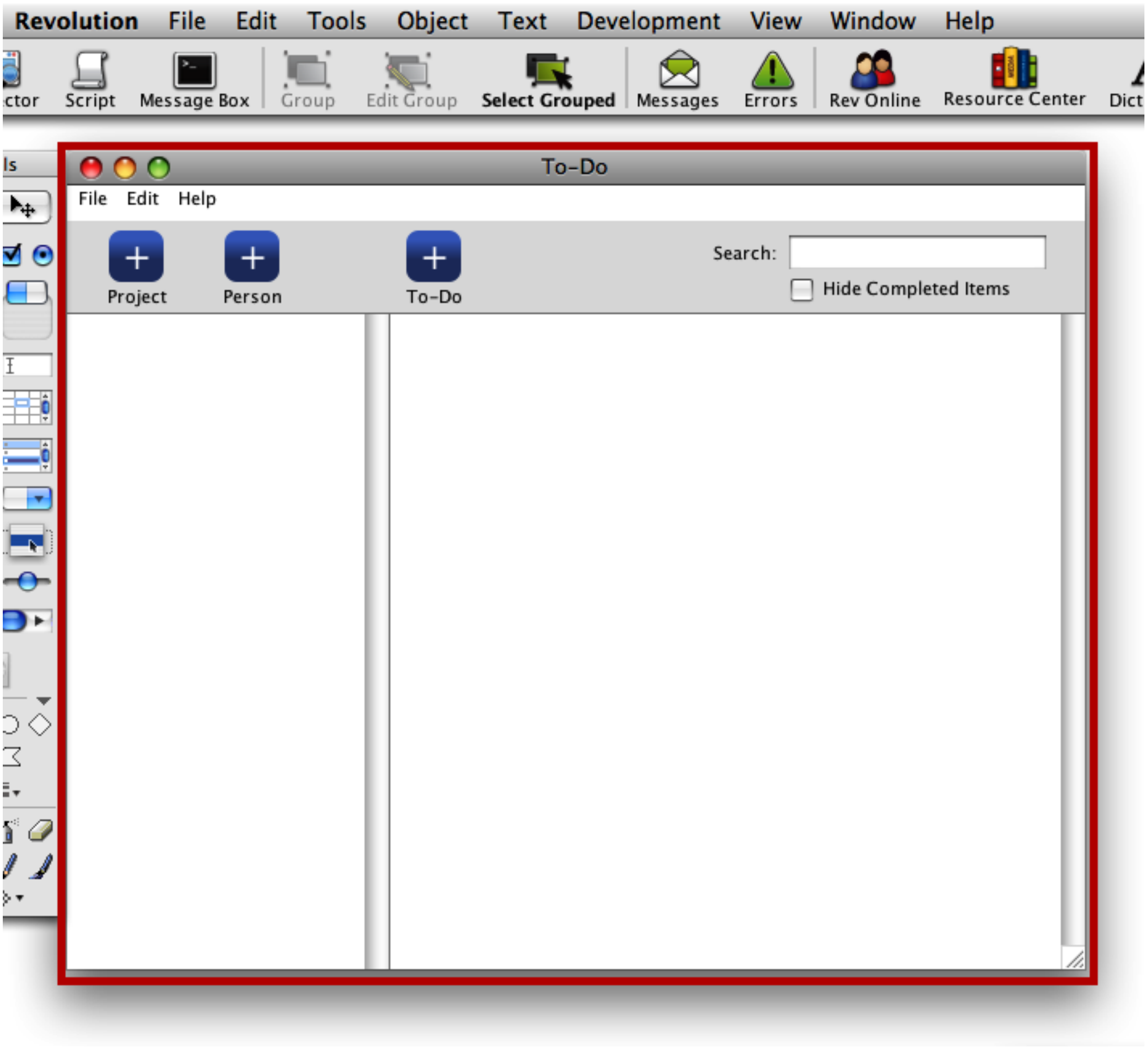


Select the **launcher.rev** file located in the **Blank Application** folder.

Load Application



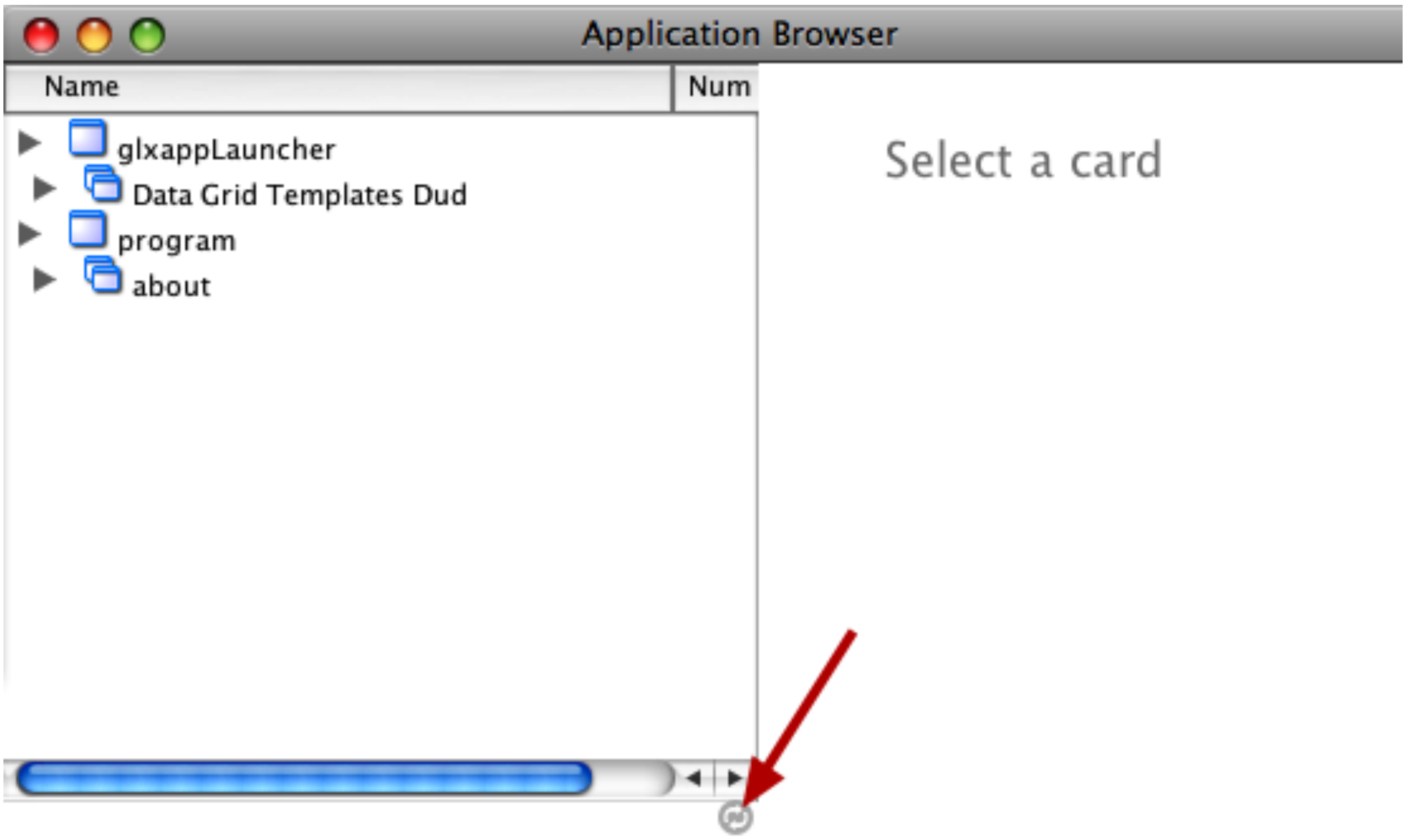
The To-Do mainstack stored in the launcher.rev stack file will appear in the IDE (1). The mainstack is named **glxappLauncher** and is the stack that the executables are built from. To load the rest of the application into the IDE select the **Browse** tool (2) and then click anywhere on the To-Do list stack (except the title bar).


Revolution File Edit Tools Object Text Development View Window Help
ctor Script Message Box Group Edit Group Select Grouped Messages Errors Rev Online Resource Center Dict
To-Do
File Edit Help
Project
Person
To-Do
Search:
Hide Completed Items

After clicking you will see the main To-Do application stack window. This stack is named **program** and titled "To-Do" (the name that appears in the title bar).

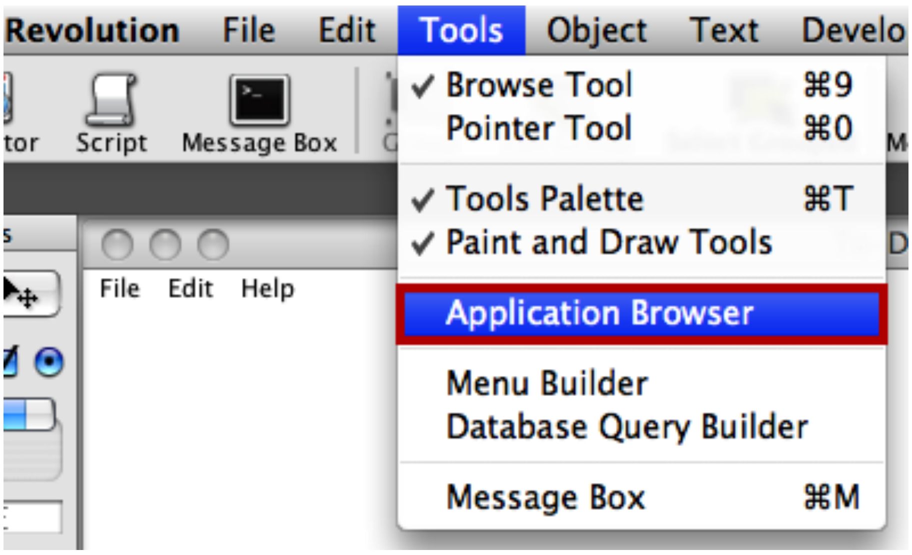
Let's take a look at the all of the stacks that are now in memory using the **Application Browser**.

Refresh the Application Browser



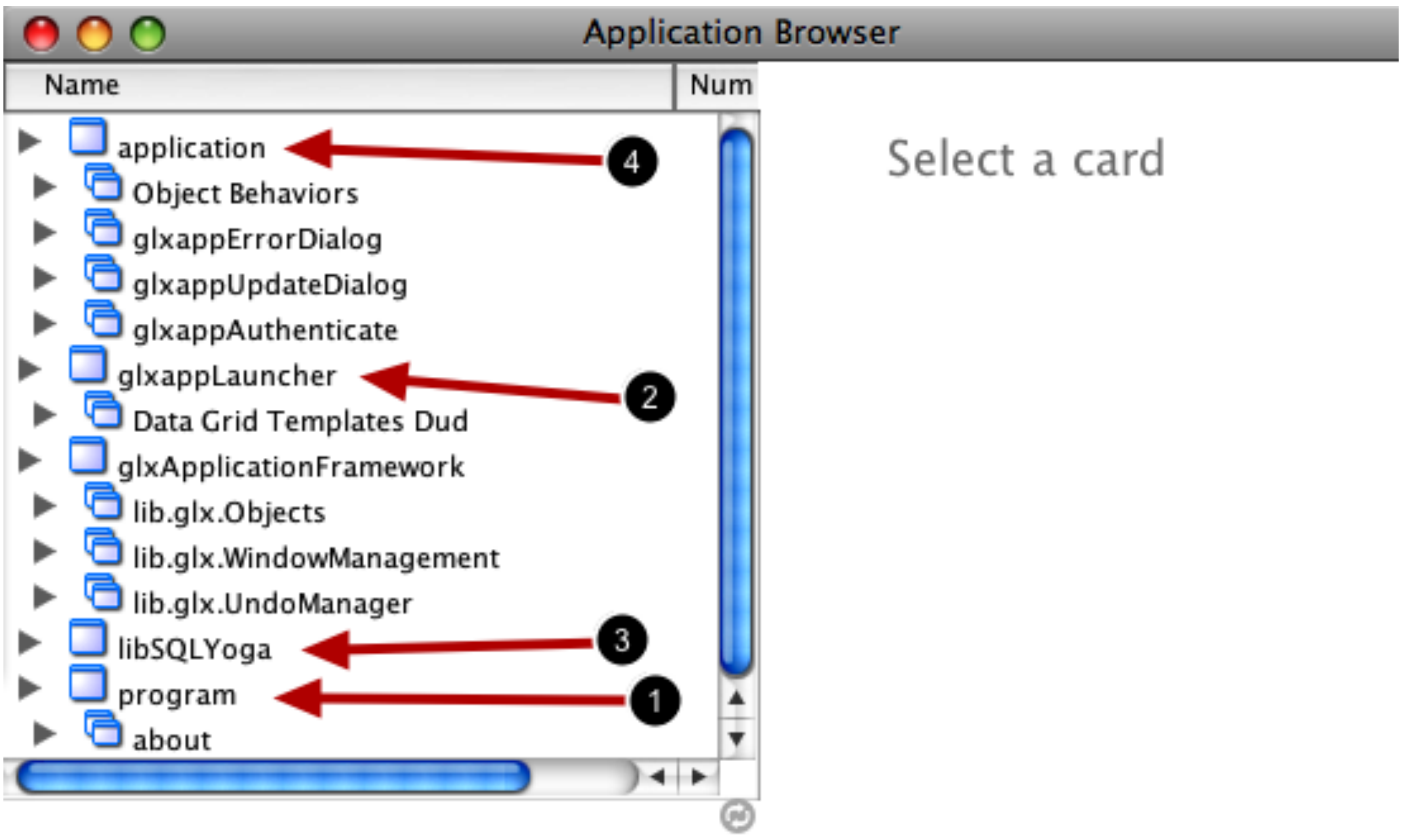
If the Application Browser was open when you clicked on the glxappLauncher stack then the Application Browser might not refresh to show all of the stacks that the application framework loaded into memory. If this is the case just click on the Refresh button and all will be well.

Open The Application in Application Browser



If the Application Browser is not open then choose **Tools > Application Browser** to open it.

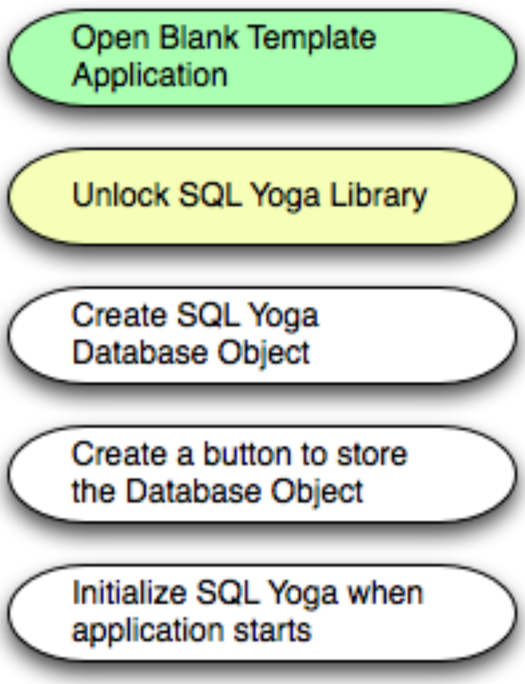
Application Stacks



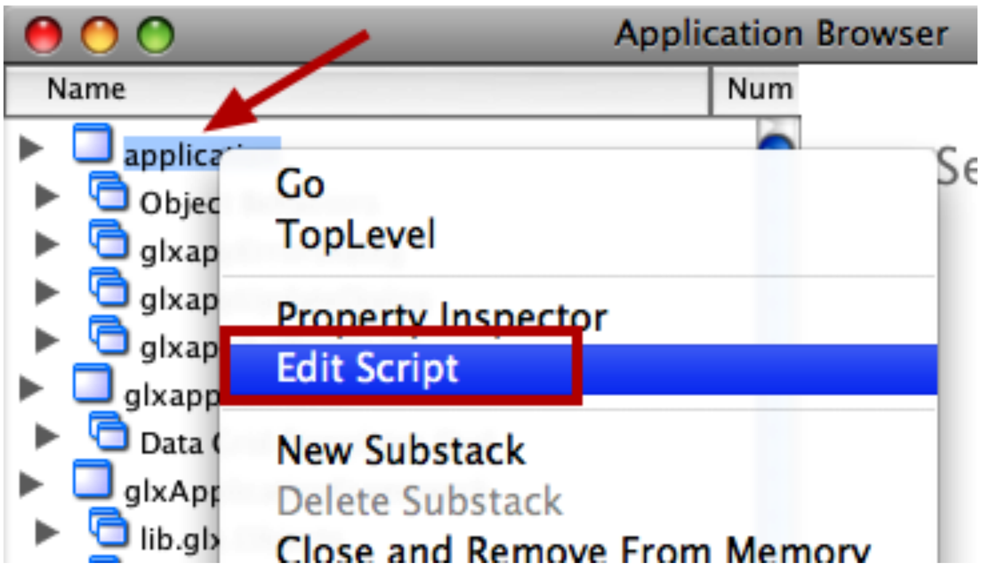
When the Application Browser opens you will see the list of stacks associated with the application. The **program** stack (1) is the window that is currently open. **glxappLauncher** (2) is the stack you initially opened and clicked on. **libSQLYoga** (3) is the SQL Yoga library and the **application** stack (4) is where the scripts that run when your application starts up and shuts down are kept.

Unlocking the SQL Yoga Library

Now that you have the blank To-Do application open in the IDE we can get started with SQL Yoga. First I will show you how to register the SQL Yoga library. The SQL Yoga library must be unlocked once per session. That means that each time your application launches sqlyoga_register should be called. Doing so will remove the demo limitations of the library.

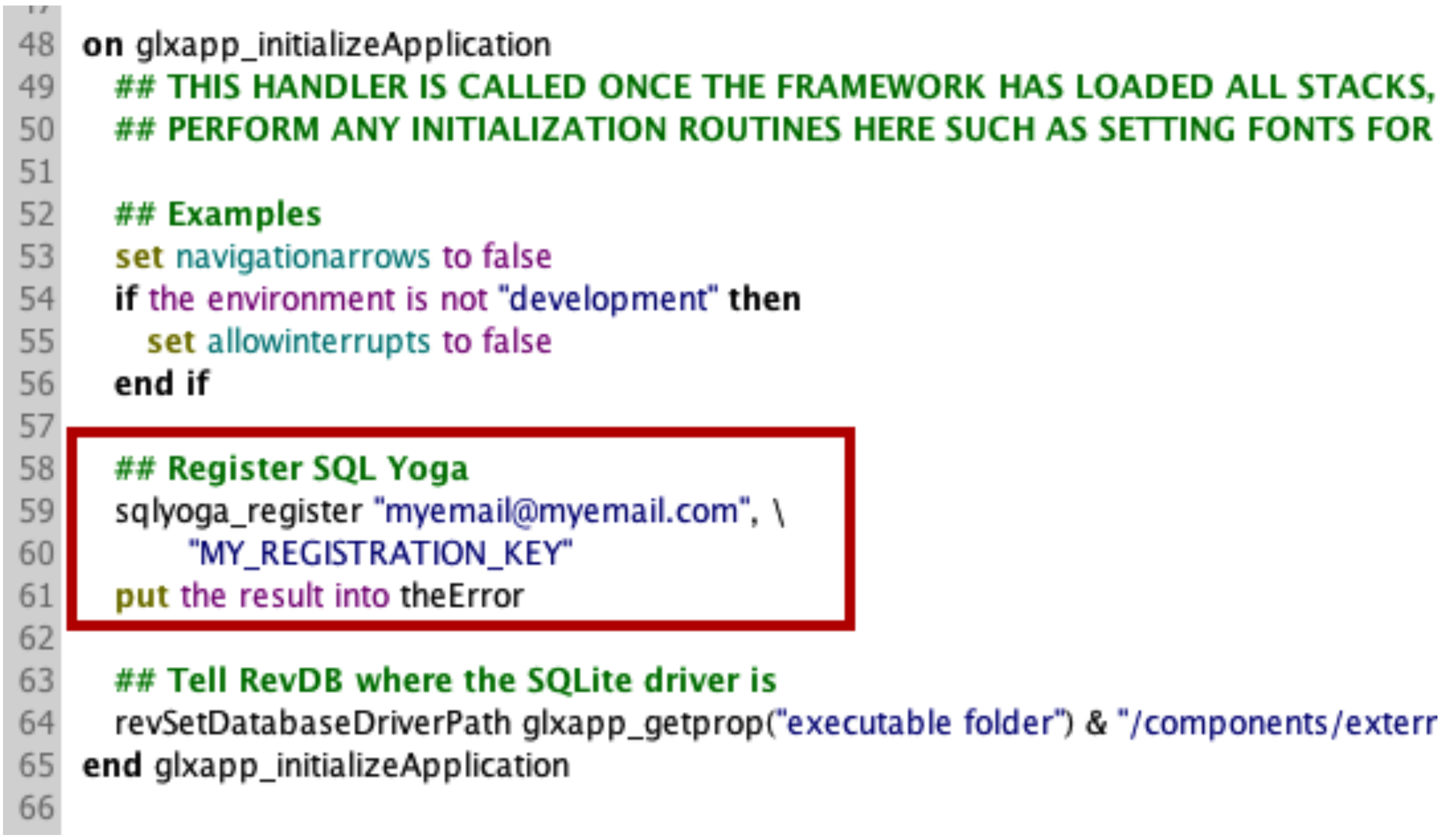

Open Blank Template Application
Unlock SQL Yoga Library
Create SQL Yoga Database Object
Create a button to store the Database Object
Initialize SQL Yoga when application starts

Edit application Stack Script



In the GLX Application Framework code that runs when your application launches is stored in the **application** stack script. Edit the script of the application stack.

sqlyoga_register



```
on glxapp_initializeApplication
  ## THIS HANDLER IS CALLED ONCE THE FRAMEWORK HAS LOADED ALL STACKS,
  ## PERFORM ANY INITIALIZATION ROUTINES HERE SUCH AS SETTING FONTS FOR

  ## Examples
  set navigationarrows to false
  if the environment is not "development" then
    set allowinterrupts to false
  end if

  ## Register SQL Yoga
  sqlyoga_register "myemail@myemail.com", \
        "MY_REGISTRATION_KEY"
  put the result into theError

  ## Tell RevDB where the SQLite driver is
  revSetDatabaseDriverPath glxapp_getprop("executable folder") & "/components/exterr
end glxapp_initializeApplication
```

SQL Yoga is incorporated into your application as a library stack named **libSQLYoga**. The GLX Application Framework takes care of the loading and putting in use of the sql_yoga.rev stack. All you need to worry about is unlocking the library each time your application launches.

To unlock SQL Yoga add a call to **sqlyoga_register** in the **glxapp_initializeApplication** handler and pass in your email and registration key. The glxapp_initializeApplication message is only called once when your application first launches so it is a good place to place the sqlyoga_register command.

Notice that in the code the result of sqlyoga_register is stored in theError but nothing is done with theError afterwards. If SQL Yoga is unable to validate the email/key combination then the error will not be empty. Use the error to help troubleshoot why your registration information was not accepted.

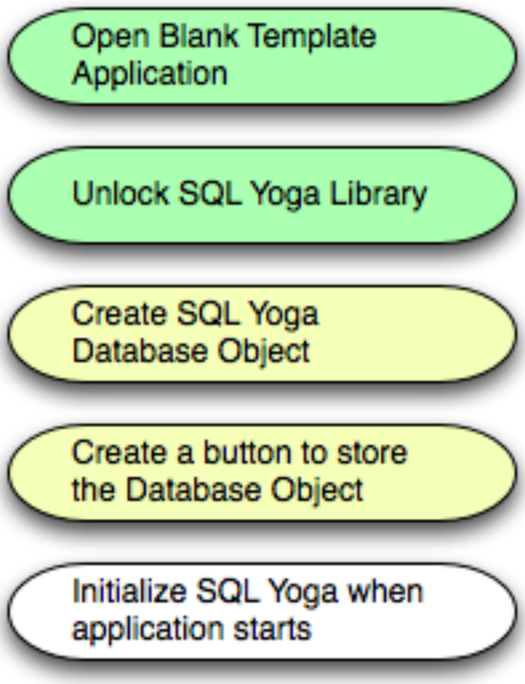
What if I don't have a SQL Yoga registration key? Don't worry, SQL Yoga will run in demo mode if it isn't registered. In demo mode no more than 10 rows of data will be returned and a dialog will appear every 10 minutes reminding you that you are in demo mode.

Creating A SQL Yoga Database Object

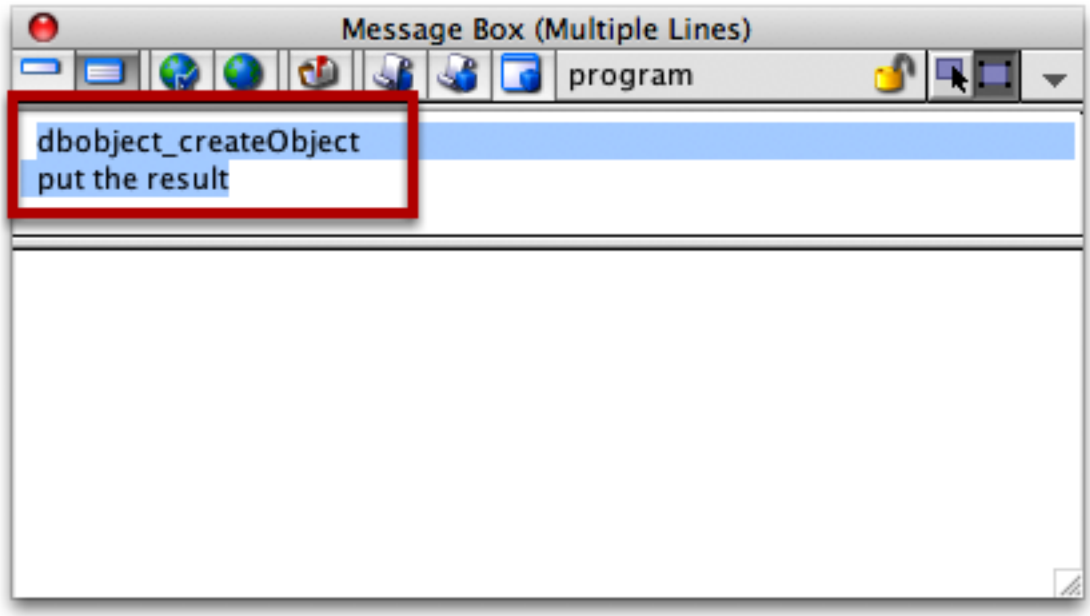
The first time you incorporate SQL Yoga into one of your applications you will need to perform a couple of tasks before you get to work. This lesson will walk you through those tasks.

Note that you should not have any other SQL Yoga projects open when performing the following steps (unless you already know what you are doing). You should start with a freshly launched IDE that only has the tutorial application open just to make sure you don't run into any problems.

To get started, open the Message Box by choosing the **Tools > Message Box** menu.


Open Blank Template Application
Unlock SQL Yoga Library
Create SQL Yoga Database Object
Create a button to store the Database Object
Initialize SQL Yoga when application starts

Create a Database Object



```
dbobject_createObject
put the result
```

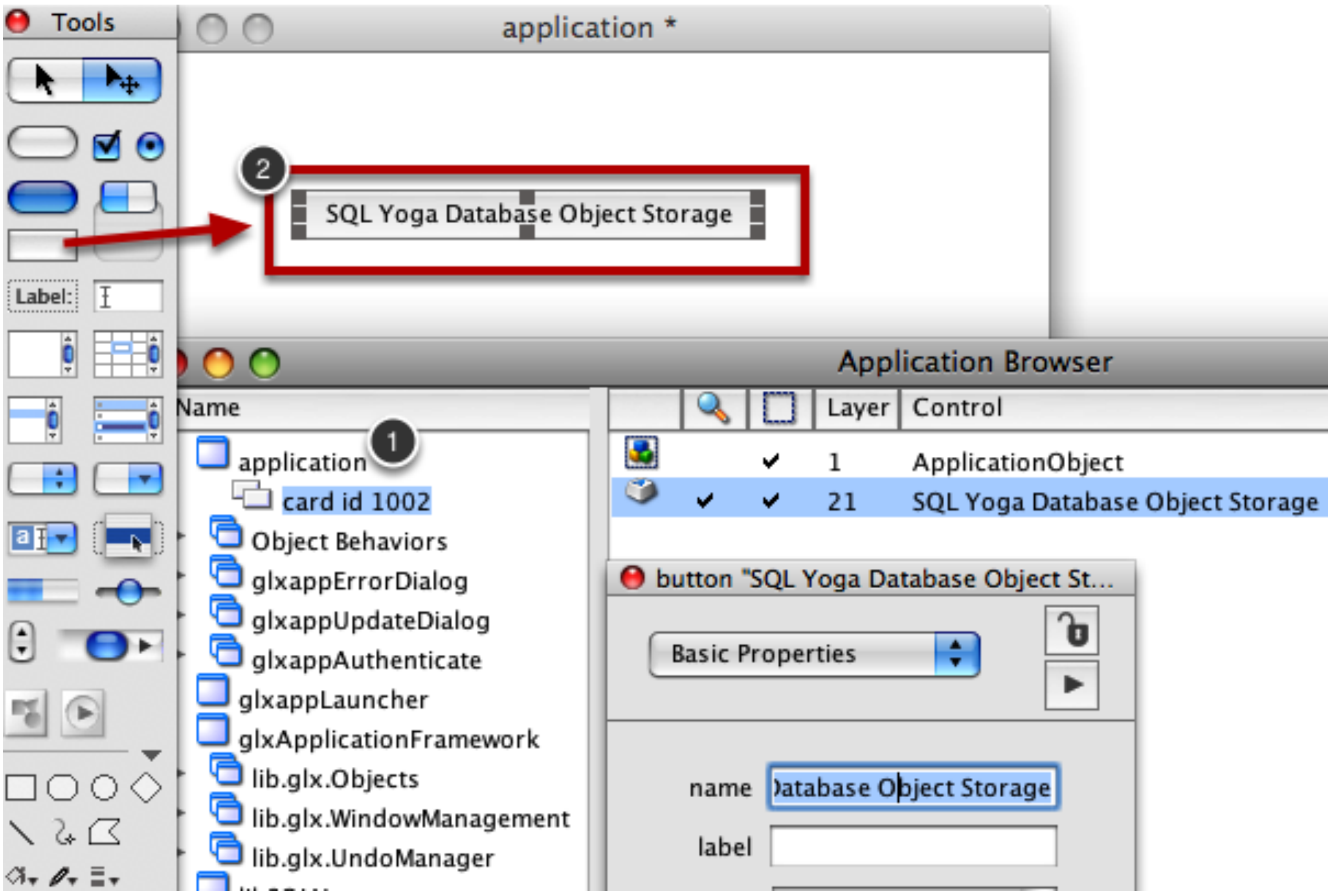
SQL Yoga stores information about your database in a **Database** object. A Database object stores all information about your database schema and other objects that you will use to interact with the database (Tables, Relationships, Scopes, etc.).

Creating a Database object only needs to be done once and can be done by calling **dbobject_createObject** in the Message Box. 'the result' will be empty if everything goes well, otherwise you will see an error appear in the Message Box.

==========
Copy & Paste
==========

```
dbobject_createObject
put the result
```

Create A Storage Object



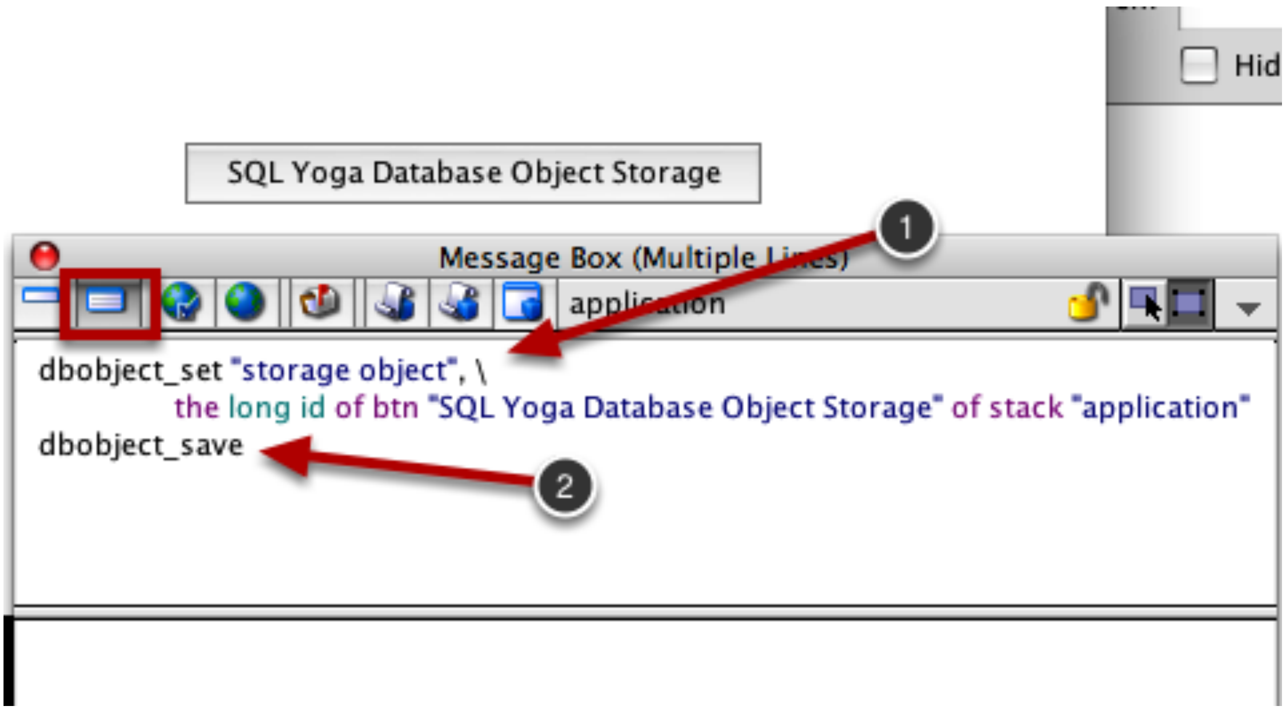
name: Database Object Storage

Now there is a Database object named 'default' that exists in memory. The next step is to create a place to store that Database object between sessions. You can store a Database object in a Revolution control (as a custom property) or a file (as an encoded array). We will look at how to store the object in a button.

Open the **application** stack (1) and create a button on the card by dragging it from the Tools Palette. Name it **SQL Yoga Database Object Storage** (2). Since this application uses the GLX Application Framework I'm going to create the button on the **application** stack as it is a good place to put controls that are application specific rather than tied to a particular window in the program.

Note: A quick way to open the application stack is to right-click on it in the Application Browser and select **Go** from the contextual menu.

Tell Database Object Where It Should Be Stored


Hid
SQL Yoga Database Object Storage
Message Box (Multiple Lines)
application
1
2

```
dbobject_set "storage object", \
        the long id of btn "SQL Yoga Database Object Storage" of stack "application"
dbobject_save
```

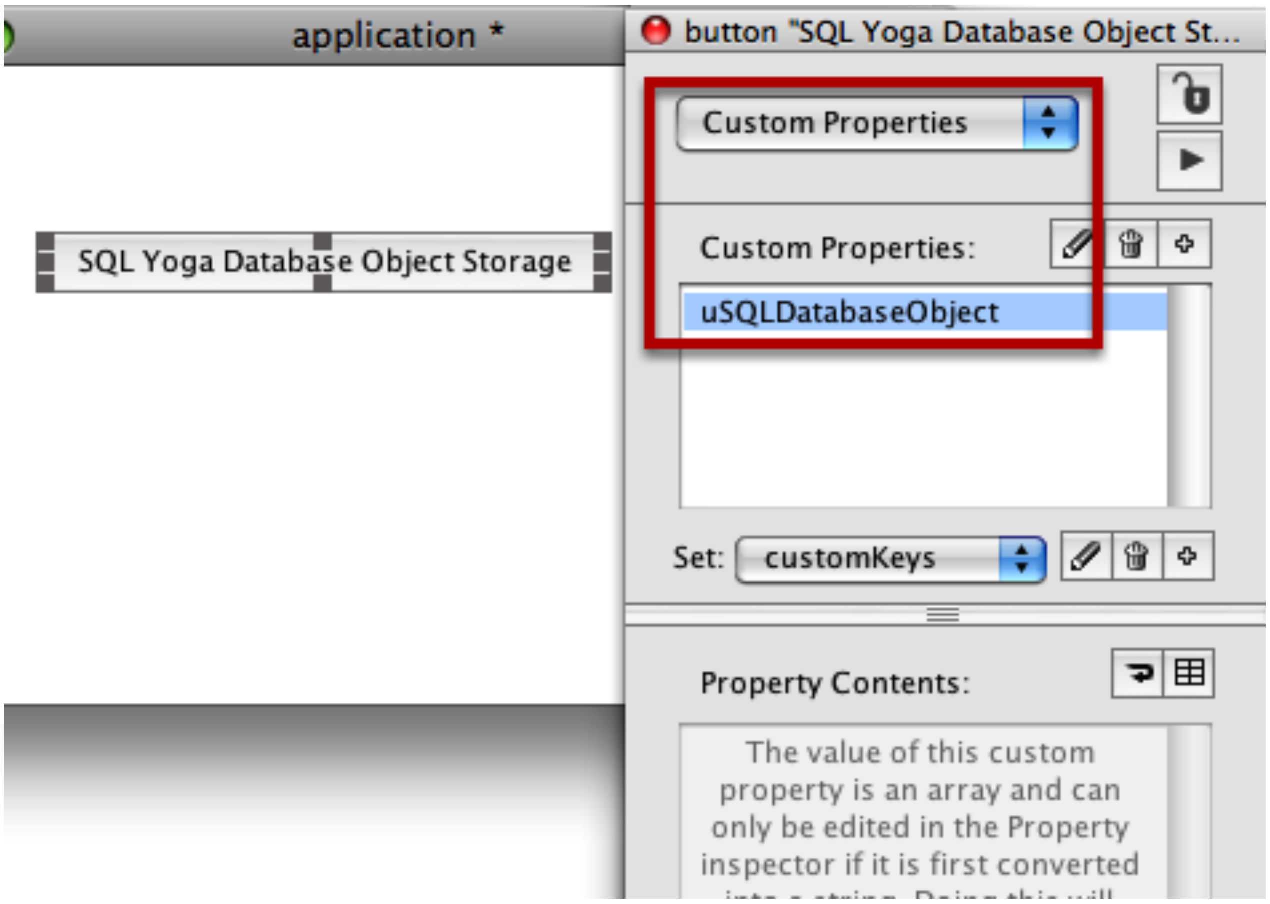
Now that you have a place to the Database object you need to tell the Database object about it. You can perform this operation in the multi-line message box. Here is what you need to do:

1) Specify the control the Database object will be saved to when calling **dbobject_save** by setting the **storage object** property and passing in the long id of the button you just created.

2) Save the Database object. **dbobject_save** will save the Database Object to the button specified in step 1.

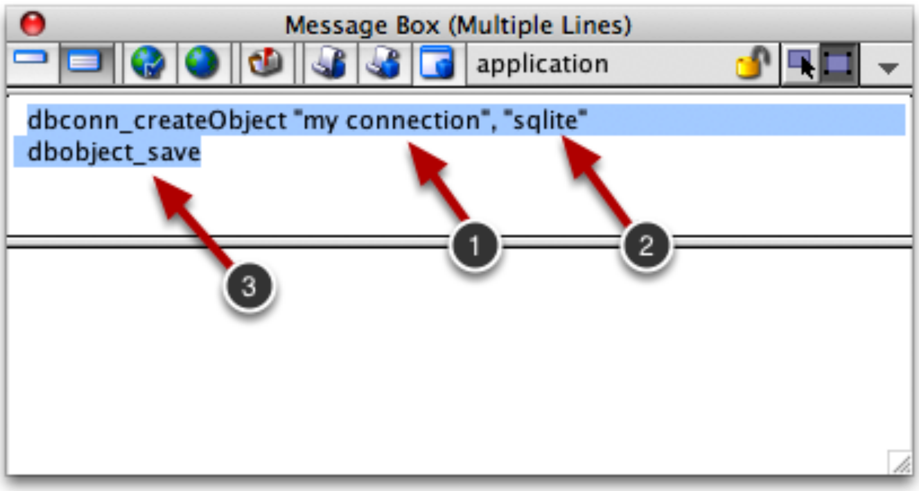
==========
Copy & Paste
==========

```
dbobject_set "storage object", \
        the long id of btn "SQL Yoga Database Object Storage" of stack "application"
dbobject_save
```


application *
SQL Yoga Database Object Storage
button "SQL Yoga Database Object St...
Custom Properties
Custom Properties:
uSQLDatabaseObject
Set: customKeys
Property Contents:
The value of this custom property is an array and can only be edited in the Property inspector if it is first converted

After executing the above code in the message box you can look at the custom properties of the button. There is now a uSQLDatabaseObject custom property that contains all the information about your Database object.

Create a Connection Object



```
dbconn_createObject "my connection", "sqlite"
dbobject_save
```

Now that you have created a Database object it is time to add other objects to it. Think of a Database object as a Group in Revolution. A Group contains other controls like buttons and fields. A Database object contains Connections, Tables, Relationships, Scopes and more.

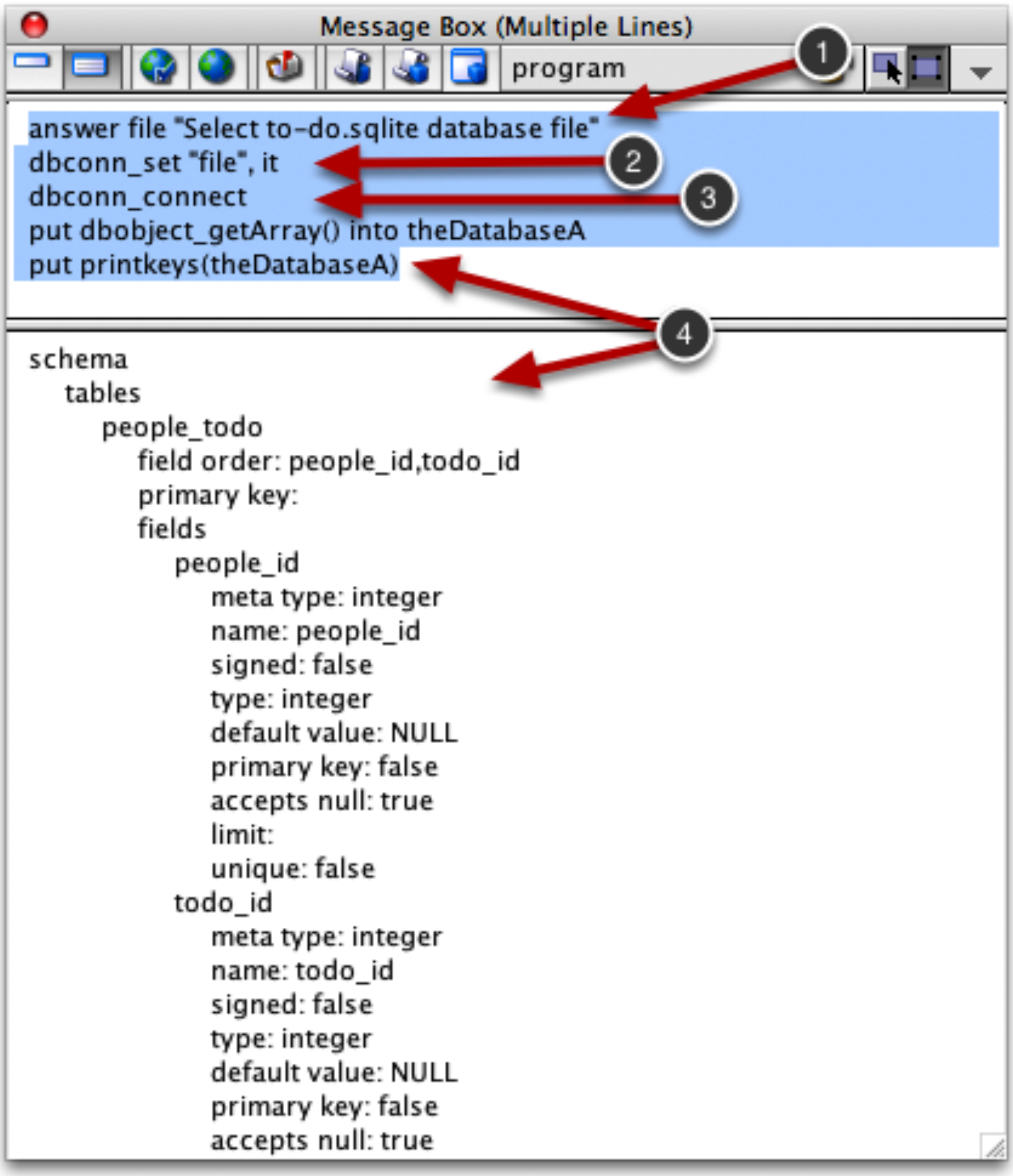
The first object you will create in the 'default' Database object you just created is a **Connection** object. Connection objects store connection settings for the databases you want to communicate with. Creating a Connection object is easy. Just execute the **dbconn_createObject** command in the message box and pass in a **name** (1) and **adaptor** (2) for the connection.

Whenever you add a new object to a Database object you should save your work. Calling **dbobject_save** (3) will save the Database object to the **SQL Yoga Database Object Storage** button. Note that dbobject_save merely saves the object to the button. You still need to save the stack containing the button to permanently store the changes to disk.

==========
Copy & Paste
==========

```
dbconn_createObject "my connection", "sqlite"
dbobject_save
```

Read In Schema By Testing Connection



```
answer file "Select to-do.sqlite database file"
dbconn_set "file", it
dbconn_connect
put dbobject_getArray() into theDatabaseA
put printkeys(theDatabaseA)
```

```
schema
    tables
        people_todo
            field order: people_id,todo_id
            primary key:
            fields
                people_id
                    meta type: integer
                    name: people_id
                    signed: false
                    type: integer
                    default value: NULL
                    primary key: false
                    accepts null: true
                    limit:
                    unique: false
                todo_id
                    meta type: integer
                    name: todo_id
                    signed: false
                    type: integer
                    default value: NULL
                    primary key: false
                    accepts null: true
```

The first time that SQL Yoga connects to a database it asks the database for information about the tables and fields in it. SQL Yoga stores this information in a **Schema** object that is stored inside the Database object. All of the SQL Yoga objects provide their automated features by using this cached information.

Let's read in the schema of the To-Do list database. Follow these steps in the multi-line Message Box:

1) Display a file selection dialog. Select the **to-do.sqlite** database file that is located in the **./Blank**

Application/components/ folder of the tutorials distribution folder.

2) Set the **file** property of the Connection object you created. Note that when you created the first Connection object SQL Yoga assigned it as the **default connection** for the Database object. The 'default connection' is assumed in all handlers with the *dbconn_* prefix so there is no need to pass in the connection name to dbconn_set.

3) Connect to the database. SQL Yoga will import the database schema when you connect.

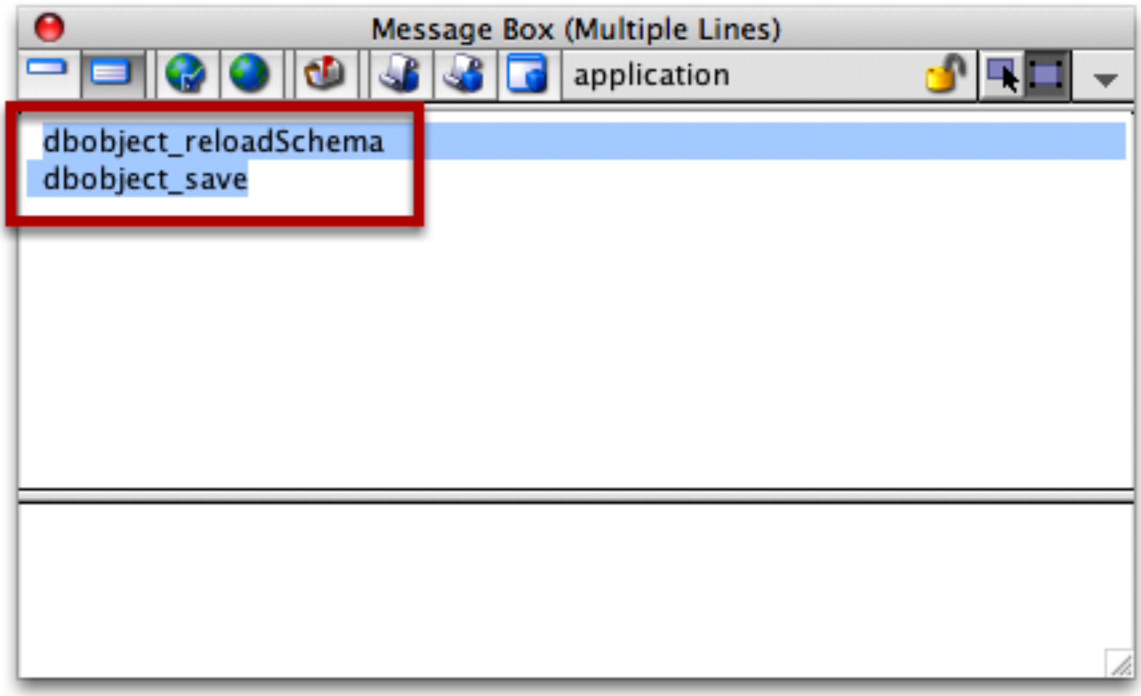
4) See what the schema looks like. You can use **dbobject_getArray**() and the helper function **printKeys** (part of SQL Yoga library) to see a printout of the Database object.

Now would be a good time to execute **dbobject_save** in the message box so that the new information that has been saved in the Schema object is saved.

==========
Copy & Paste
==========

```
answer file "Select to-do.sqlite database file"
dbconn_set "file", it
dbconn_connect
put dbobject_getArray() into theDatabaseA
put printkeys(theDatabaseA)
```

Let SQL Yoga Know If You Update Your Database Schema



```
dbobject_reloadSchema
dbobject_save
```

If you ever add, remove or alter tables in your database then you need to tell SQL Yoga about it. To do this just call **dbobject_reloadSchema** which clears out the existing schema stored in your Database object and imports it again using the default Connection object.

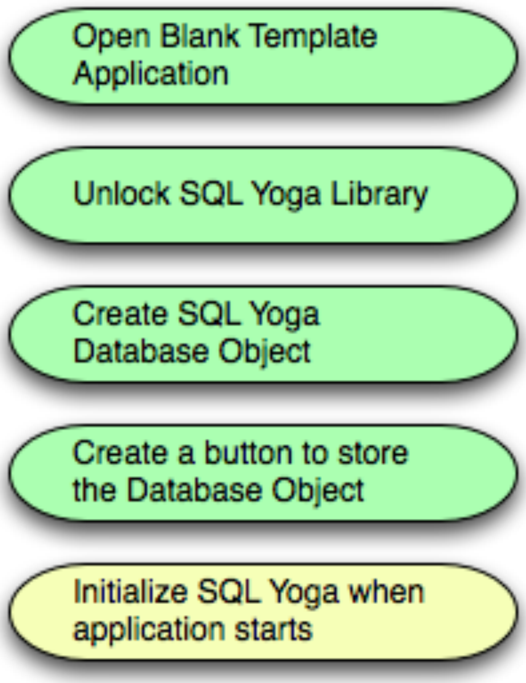
==========
Copy & Paste
==========
dbobject_reloadSchema
dbobject_save

That's It

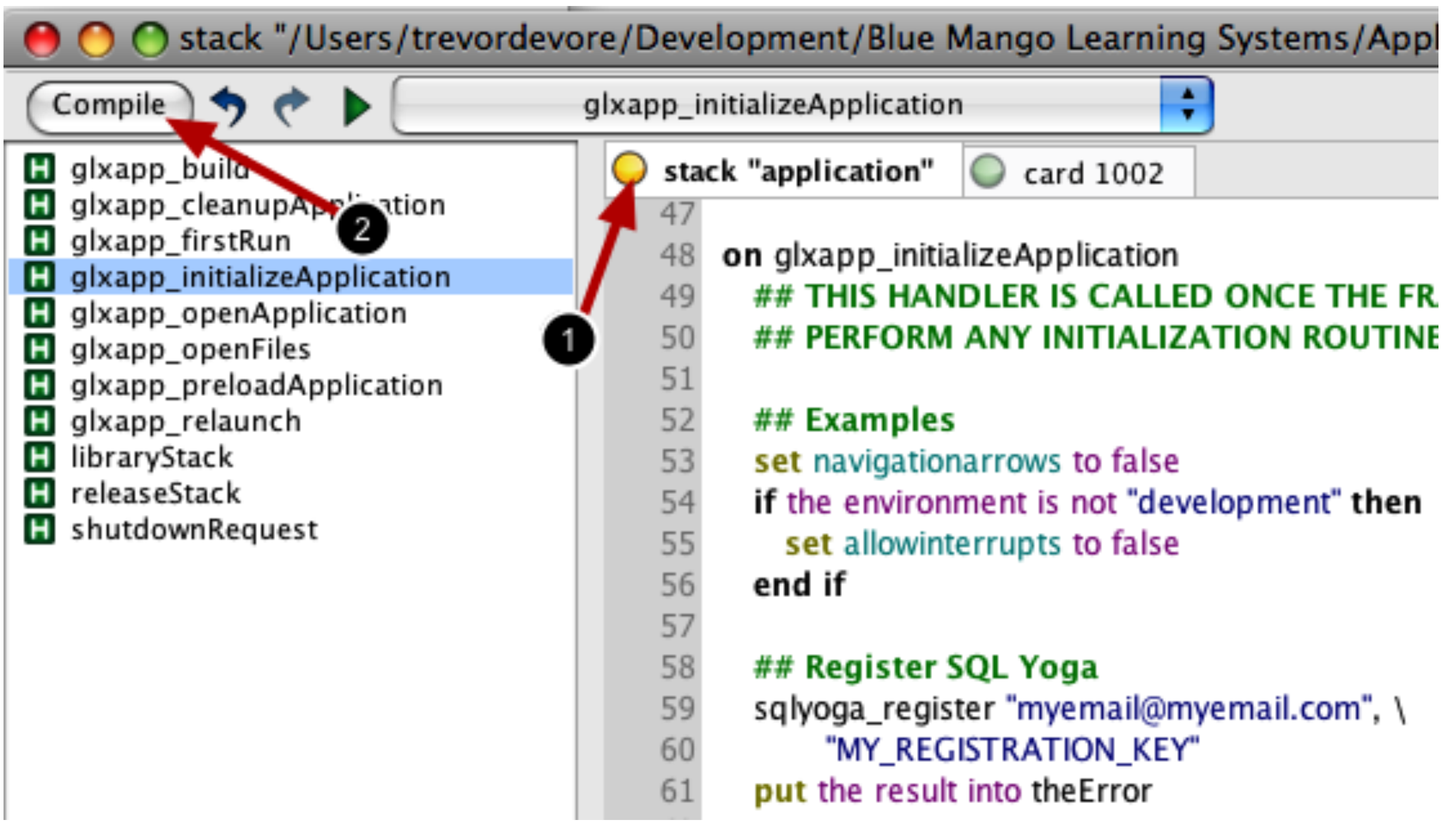
You have now performed all of the preliminary tasks for setting up SQL Yoga for a project.

Initializing SQL Yoga At Launch

Now that you have configured a SQL Yoga Database and Connection object we will look at how to initialize your Database object each time the application launches.


Open Blank Template Application
Unlock SQL Yoga Library
Create SQL Yoga Database Object
Create a button to store the Database Object
Initialize SQL Yoga when application starts

Update 'application' Stack Script



stack "/Users/trevordevore/Development/Blue Mango Learning Systems/Appl
Compile
glxapp_initializeApplication
glxapp_build
glxapp_cleanupApplication
glxapp_firstRun
glxapp_initializeApplication
glxapp_openApplication
glxapp_openFiles
glxapp_preloadApplication
glxapp_relaunch
libraryStack
releaseStack
shutdownRequest
stack "application"
card 1002
1
2

```
on glxapp_initializeApplication
   ## THIS HANDLER IS CALLED ONCE THE FR
   ## PERFORM ANY INITIALIZATION ROUTINE

   ## Examples
   set navigationarrows to false
   if the environment is not "development" then
      set allowinterrupts to false
   end if

   ## Register SQL Yoga
   sqlyoga_register "myemail@myemail.com", \
         "MY_REGISTRATION_KEY"
   put the result into theError
```

Before we walk through the code, update the **application** stack script by replacing the **glxapp_initializeApplication** handler with the following RevTalk. After you insert the code we will go through it step by step.

After pasting in the updated code the script tab will display a yellow circle (1). Make sure you compile the script so that the changes are saved (2).

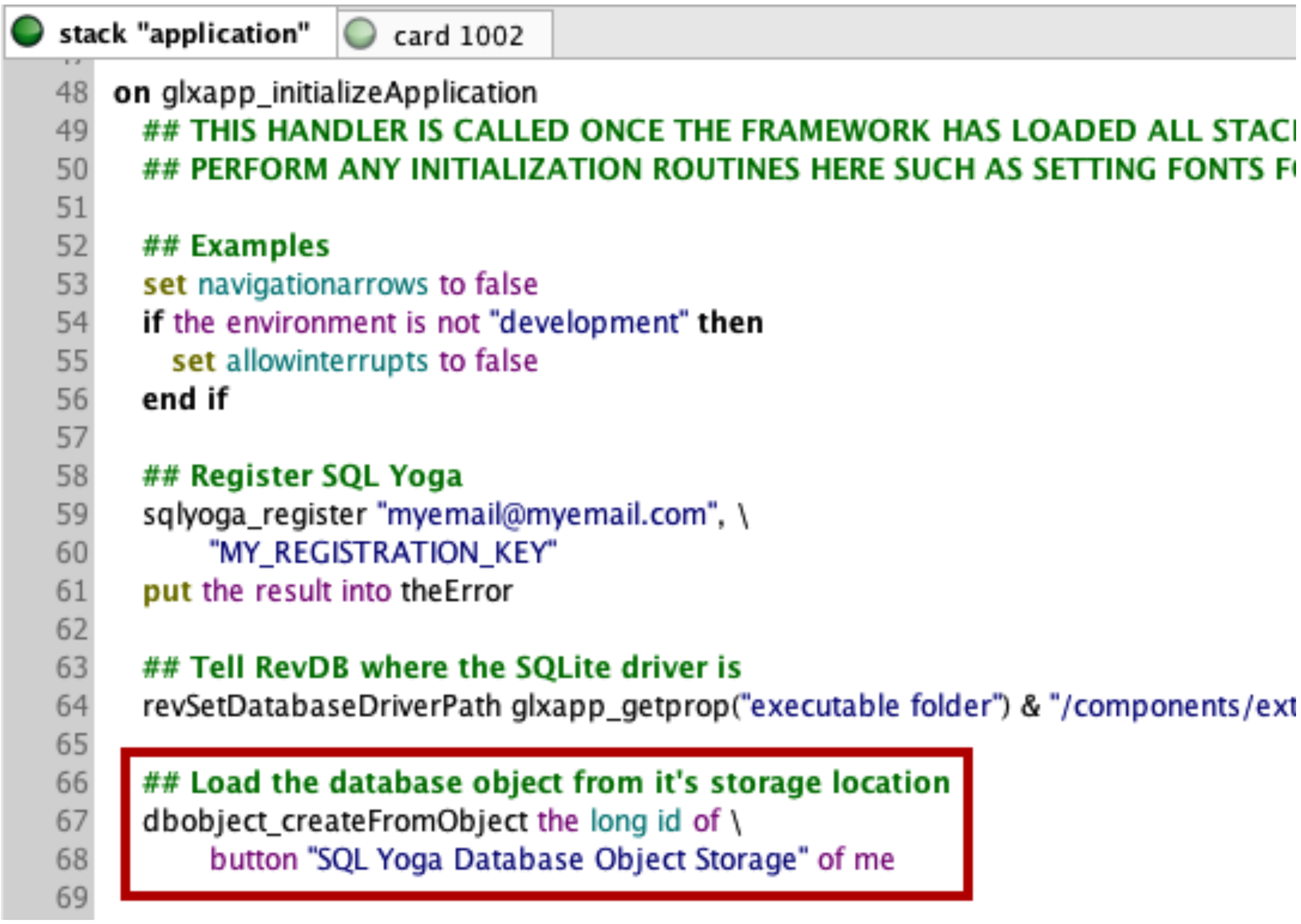
Copy & Paste The Following Code
----

on glxapp_initializeApplication
   ## THIS HANDLER IS CALLED ONCE THE FRAMEWORK HAS LOADED ALL STACKS, LIBRARIES AND EXTERNALS.
   ## PERFORM ANY INITIALIZATION ROUTINES HERE SUCH AS SETTING FONTS FOR STACKS, navigationArrows, ETC

   ## Examples
   set navigationarrows to false
   if the environment is not "development" **then**

```
    set allowinterrupts to false
  end if

  ## Register SQL Yoga
  sqlyoga_register "myemail@myemail.com", \
        "MY_REGISTRATION_KEY"
  put the result into theError

  ## Tell RevDB where the SQLite driver is
  revSetDatabaseDriverPath glxapp_getprop("executable folder") & "/components/externals"

  ## Load the database object from it's storage location
  dbobject_createFromObject the long id of \
        button "SQL Yoga Database Object Storage" of me

  ## Point Connection object to SQLite database file
  put glxapp_getprop("executable folder") & "/components/to-do.sqlite" into theFile
  dbconn_set "file", theFile

  dbconn_connect
  put the result into theError

  if theError is not empty then
    answer "Unable to initialize program (" & theError & "). Application will quit."
    if the environment is not "development" then
      ## returning false tells framework that initialization failed and app
      ## should quit.
      return false
    end if
  end if
end glxapp_initializeApplication
```

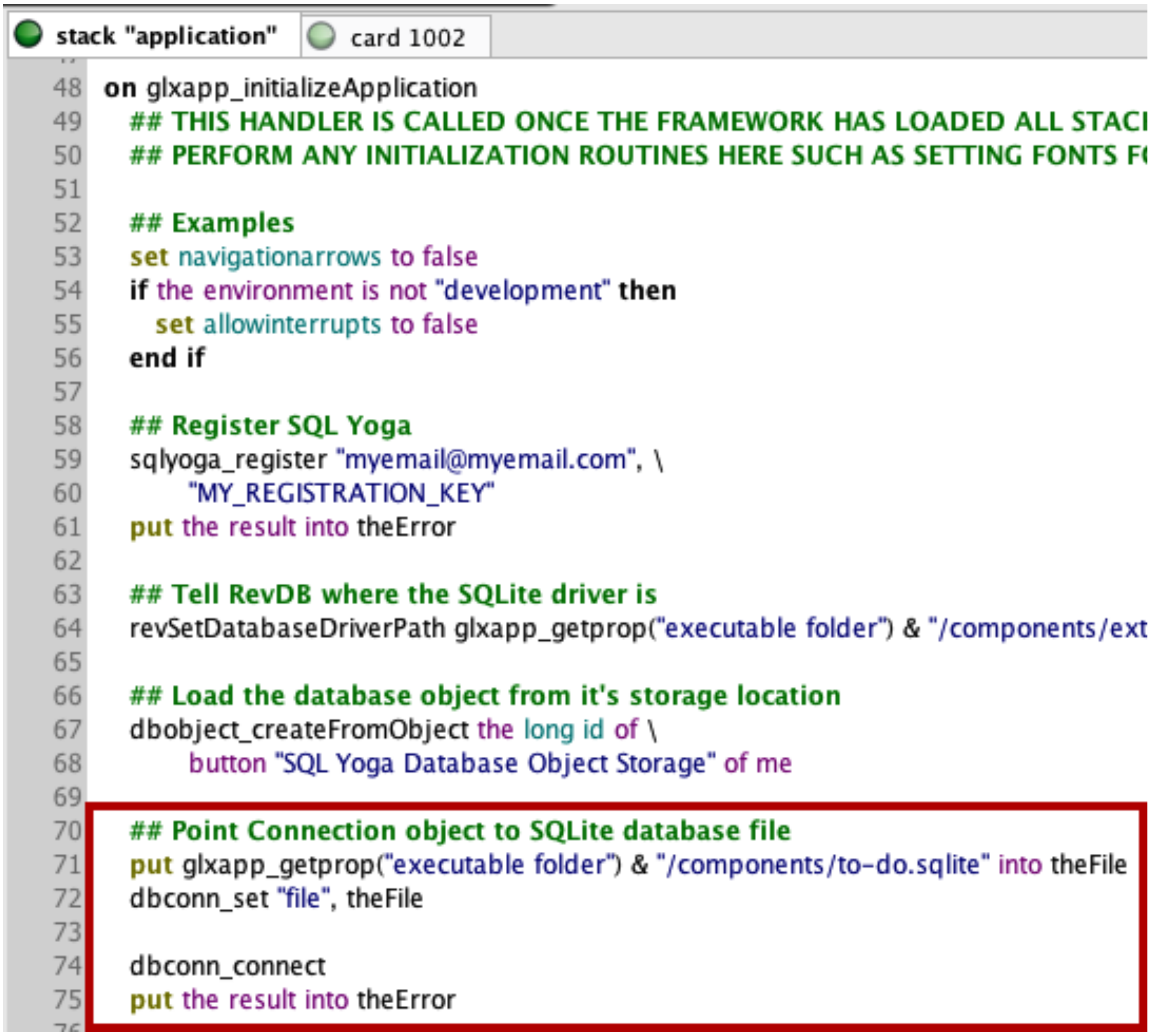
Loading The Database Object When Application Opens



```
on glxapp_initializeApplication
   ## THIS HANDLER IS CALLED ONCE THE FRAMEWORK HAS LOADED ALL STACI
   ## PERFORM ANY INITIALIZATION ROUTINES HERE SUCH AS SETTING FONTS F

   ## Examples
   set navigationarrows to false
   if the environment is not "development" then
      set allowinterrupts to false
   end if

   ## Register SQL Yoga
   sqlyoga_register "myemail@myemail.com", \
         "MY_REGISTRATION_KEY"
   put the result into theError

   ## Tell RevDB where the SQLite driver is
   revSetDatabaseDriverPath glxapp_getprop("executable folder") & "/components/ext

   ## Load the database object from it's storage location
   dbobject_createFromObject the long id of \
         button "SQL Yoga Database Object Storage" of me

```

Each time your application launches you need to load the Database object that is stored in the **SQL Yoga Database Object Storage** button into memory so that SQL Yoga has access to all of the information. You do this by calling **dbobject_createFromObject** (1). You only need to do this one time at the beginning of each application session.

Point The Connection Object to the SQLite Database File and Connect



```
on glxapp_initializeApplication
  ## THIS HANDLER IS CALLED ONCE THE FRAMEWORK HAS LOADED ALL STACI
  ## PERFORM ANY INITIALIZATION ROUTINES HERE SUCH AS SETTING FONTS FI

  ## Examples
  set navigationarrows to false
  if the environment is not "development" then
    set allowinterrupts to false
  end if

  ## Register SQL Yoga
  sqlyoga_register "myemail@myemail.com", \
       "MY_REGISTRATION_KEY"
  put the result into theError

  ## Tell RevDB where the SQLite driver is
  revSetDatabaseDriverPath glxapp_getprop("executable folder") & "/components/ext

  ## Load the database object from it's storage location
  dbobject_createFromObject the long id of \
       button "SQL Yoga Database Object Storage" of me

  ## Point Connection object to SQLite database file
  put glxapp_getprop("executable folder") & "/components/to-do.sqlite" into theFile
  dbconn_set "file", theFile

  dbconn_connect
  put the result into theError
```

We need configure Connection object properties each time the application launches. SQL Yoga does not permanently store properties such as the path to the SQLite file or the host, username and password for a MySQL database.

Even though the to-do.sqlite database file is stored alongside the application you won't know the full path to the database file until the application launches. We determine this path using an application framework property to locate the file and then setting the **file** property of the Connection object to it. Now is also a good time to connect to the database. By doing so you can check for errors and alert the user accordingly.

When working with the GLX Application Framework the **executable folder** property returns the root folder where application files are stored so you can use it to find the path to the to-do.sqlite file.

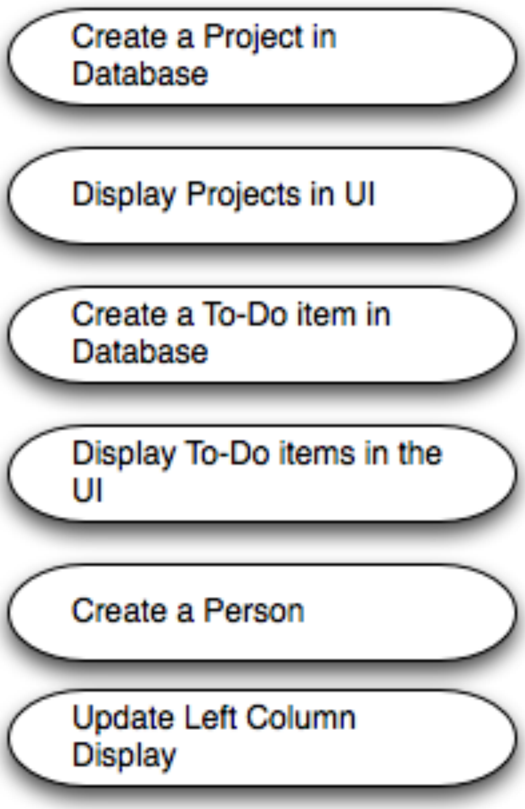
That's It!

Now you have all of the pieces in place in order to use SQL Yoga in your application. Let's go have some fun by creating some GUI controls that allow us to interact with the data.

Creating Records In and Displaying Records From A Database

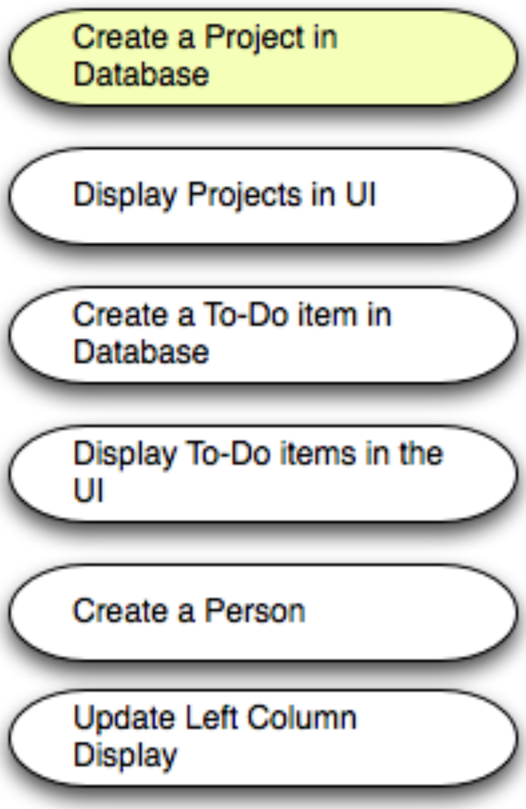
Chapter Overview

Here are the major subjects we are going to cover in this chapter.

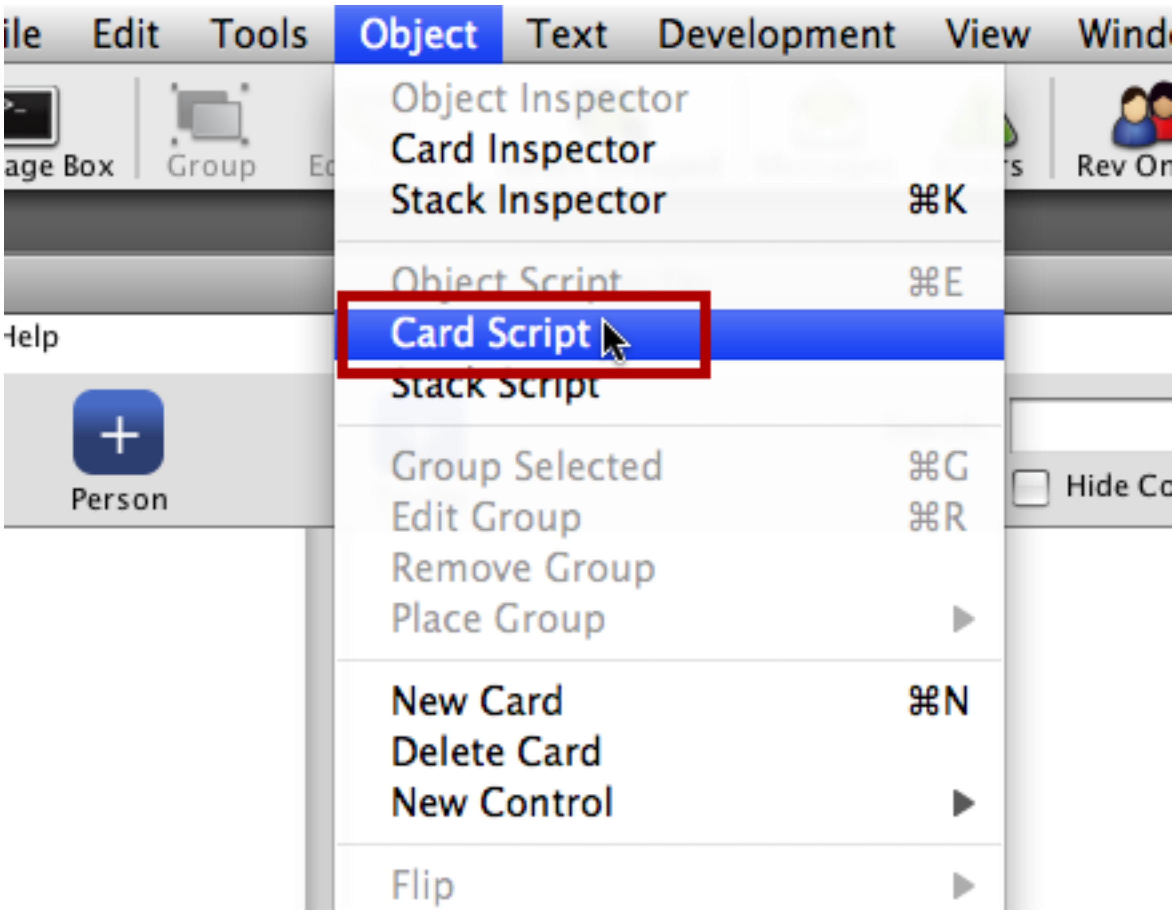

Create a Project in Database
Display Projects in UI
Create a To-Do item in Database
Display To-Do items in the UI
Create a Person
Update Left Column Display

Creating a Project

Let's begin by learning how to create a record in the database. We will start by creating a new project in the **projects** table.

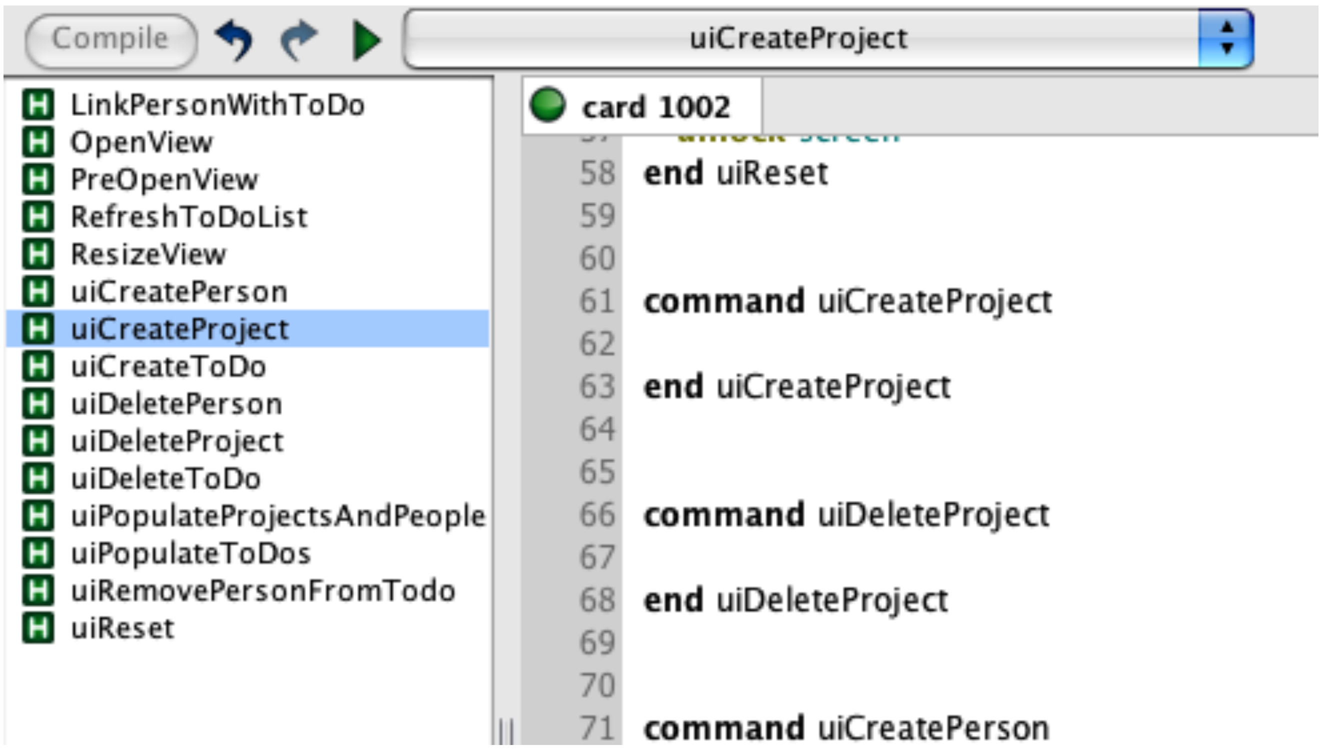

Create a Project in Database
Display Projects in UI
Create a To-Do item in Database
Display To-Do items in the UI
Create a Person
Update Left Column Display

Edit Card Script



Edit the current card script of the **program** stack. If the **To-Do** window is frontmost then you can choose **Object > Card Script**.

Locate uiCreateProject



LinkPersonWithToDo
OpenView
PreOpenView
RefreshToDoList
ResizeView
uiCreatePerson
uiCreateProject
uiCreateToDo
uiDeletePerson
uiDeleteProject
uiDeleteToDo
uiPopulateProjectsAndPeople
uiPopulateToDos
uiRemovePersonFromTodo
uiReset

```
end uiReset


command uiCreateProject

end uiCreateProject


command uiDeleteProject

end uiDeleteProject


command uiCreatePerson
```

Locate the **uiCreateProject** handler in the card script. This is where you will add the logic that creates a record in the **projects** table of the database.

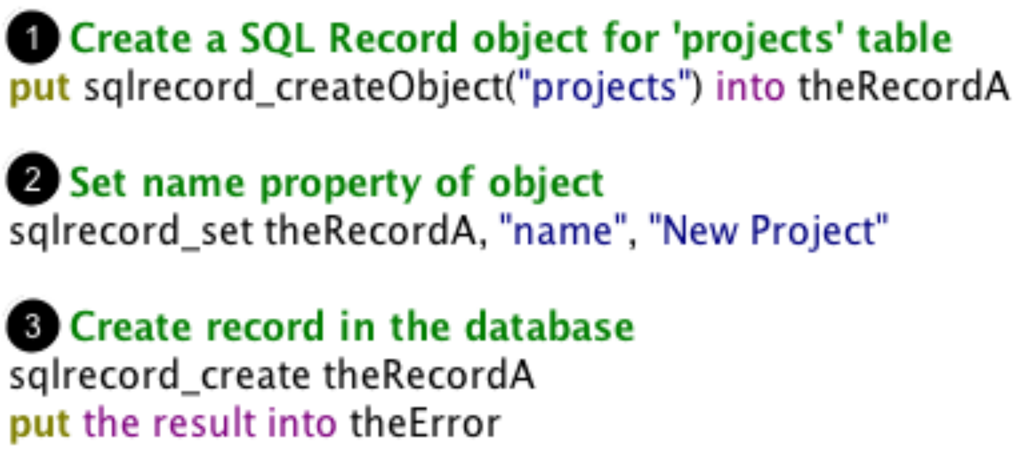
Update uiCreateProject Command

Replace the uiCreateProject handler in the card script with the RevTalk code below. Make sure and compile the script after pasting in the new code so that the updated script is applied. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command uiCreateProject
   ## Create a SQL Record object for 'projects' table
   put sqlrecord_createObject("projects") into theRecordA

   ## Set name property of object
   sqlrecord_set theRecordA, "name", "New Project"

   ## Create record in the database
   sqlrecord_create theRecordA
   put the result into theError

   if theError is empty then
      ## Refresh list
      lock screen
      RefreshProjectsPeopleList

      ## select new record (uSelectedProjectID is a custom property defined in group script)
      set the uSelectedProjectID of group "ProjectsPeople" to theRecordA["id"]

      ## tell Data Grid to select all text when field editor is opened
      set the dgTemplateFieldEditor["select text"] of group "ProjectsPeople" to true

      ## open field editor so that user can change name.
      dispatch "EditKeyOfIndex" to group "ProjectsPeople" with "name", \
            the dgHilitedIndex of group "ProjectsPeople"
      unlock screen
   end if

   if theError is not empty then
      answer "Error creating project:" && theError & "."
   end if
end uiCreateProject
```

Using SQL Yoga To Create a Record in the Database


1 Create a SQL Record object for 'projects' table
2 Set name property of object
3 Create record in the database

```
put sqlrecord_createObject("projects") into theRecordA

sqlrecord_set theRecordA, "name", "New Project"

sqlrecord_create theRecordA
put the result into theError
```

The code to create a new record is fairly straightforward.

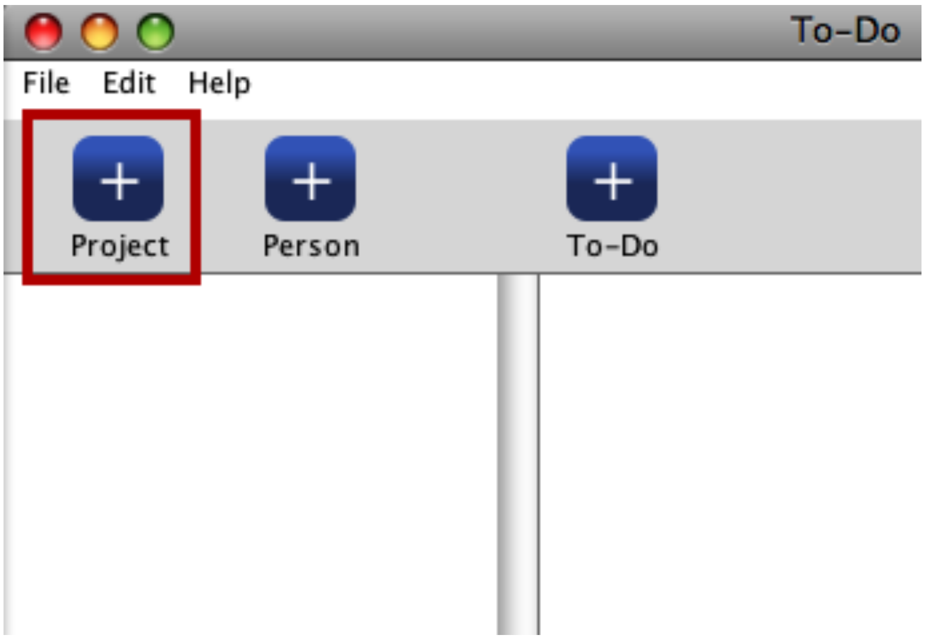
1) Create a **SQL Record object** for the **projects** table. This object is represented by an array and it

contains the keys for each of the fields (or columns) in your database. Each field key in the array contains the default value as defined by the database. Note that since the object is represented using an array it only exists as long as the array variable exists. A SQL Record object does not persist across sessions.

2) Set table field properties. We are going to assign every new project the name "New Project". The user can change the name in the UI if they wish.

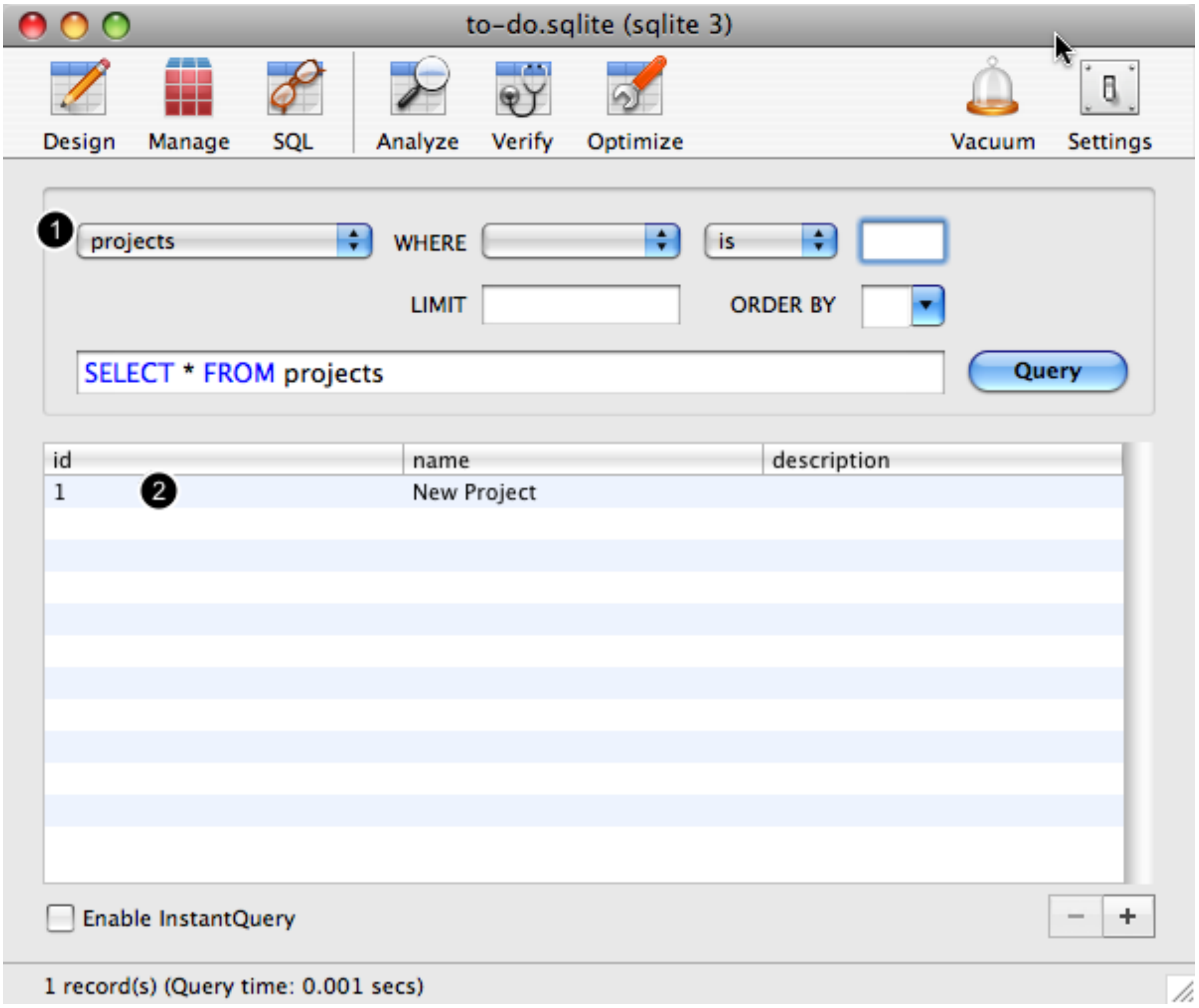
3) Tell SQL Yoga to create a record in the database that has the properties of the SQL Record object.

Test Record Creation In The Database



You that you have updated the card script you can test. The **Project** button already has a script that calls **uiCreateProject** in the card script. Click on the button to create a record. **YOU WON'T GET ANY VISUAL FEEDBACK AT THIS POINT**.

Confirm That Record Was Created



```
SELECT * FROM projects
```

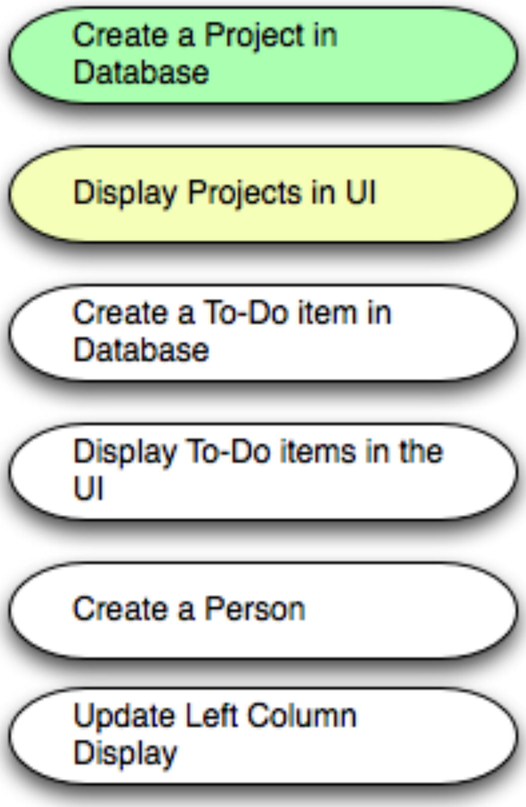
id	name	description
1	New Project	

1 record(s) (Query time: 0.001 secs)

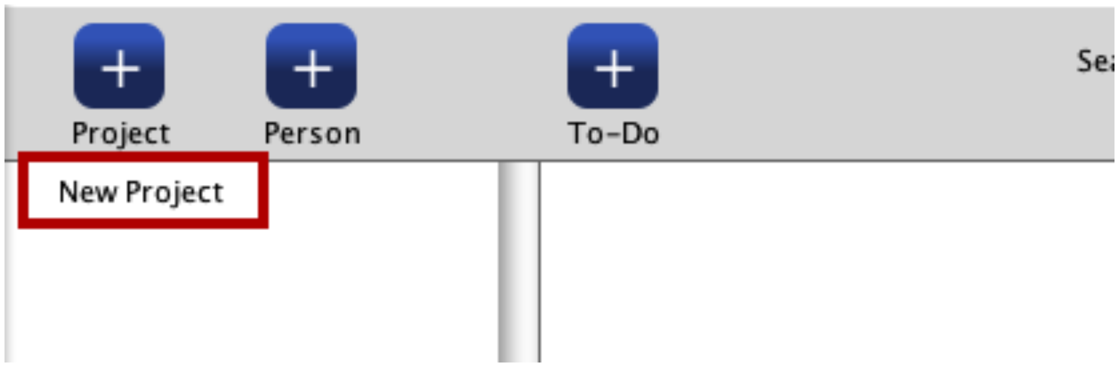
To confirm that a record was created you can open up your favorite database manager and look at the records in the **projects** table (1). You will see that one record exists named **New Project** (2).

Display Projects in the UI

Now that we have a way of creating a new project using the UI we need to be able to display them. Let's look at how to do that.


Create a Project in Database
Display Projects in UI
Create a To-Do item in Database
Display To-Do items in the UI
Create a Person
Update Left Column Display

What We Are Going To Do



Now we are going to add the RevTalk code that will display projects from the database in the UI.

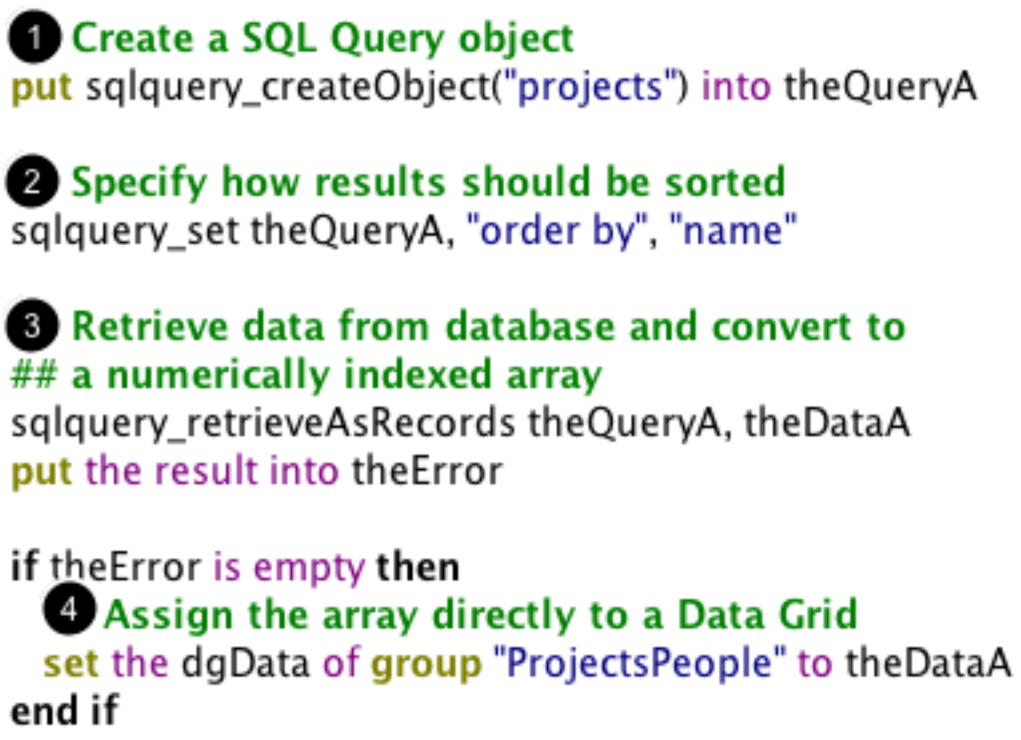
Update uiPopulateProjectsAndPeople Handler

Begin by replacing the existing uiPopulateProjectsAndPeople handler in the card script with the RevTalk code below. Remember to compile the script. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command uiPopulateProjectsAndPeople
   ## Create a SQL Query object
   put sqlquery_createObject("projects") into theQueryA

   ## Specify how results should be sorted
   sqlquery_set theQueryA, "order by", "name"

   ## Retrieve data from database and convert to
   ## a numerically indexed array
   sqlquery_retrieveAsRecords theQueryA, theDataA
   put the result into theError

   if theError is empty then
      ## Assign the array directly to a Data Grid
      set the dgData of group "ProjectsPeople" to theDataA
   end if

   if theError is not empty then
      answer "Error populating projects and people:" && theError & "."
   end if
end uiPopulateProjectsAndPeople
```

Using a SQL Query Object to Retrieve Data



```
1 Create a SQL Query object
put sqlquery_createObject("projects") into theQueryA

2 Specify how results should be sorted
sqlquery_set theQueryA, "order by", "name"

3 Retrieve data from database and convert to
## a numerically indexed array
sqlquery_retrieveAsRecords theQueryA, theDataA
put the result into theError

if theError is empty then
  4 Assign the array directly to a Data Grid
  set the dgData of group "ProjectsPeople" to theDataA
end if
```

We are going to retrieve projects from the database using a SQL Query object. A SQL Query object is similar to a SQL Record object in that it is represented by an array variable. It does not persist across sessions. Here is how you can get an array variable that contains a list of projects and display the list in a Data Grid:

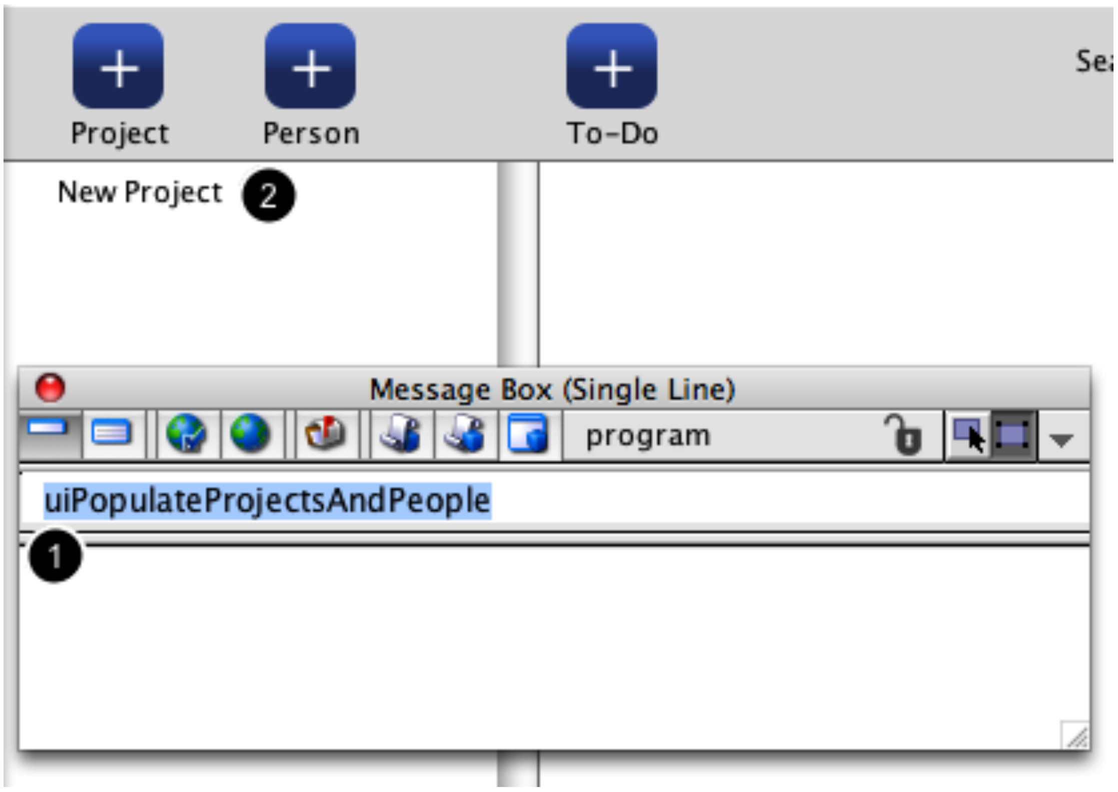
1) Create a **SQL Query object** for the **projects** table.

2) Specify how you would like the projects sorted by setting the **sort by** property.

3) Tell SQL Yoga to retrieve the projects from the database and convert the result to a numerically indexed array of SQL Record objects.

4) Assign the array returned by SQL Yoga to the **ProjectsPeople** Data Grid. This Data Grid has already been configured for displaying the content.

Test

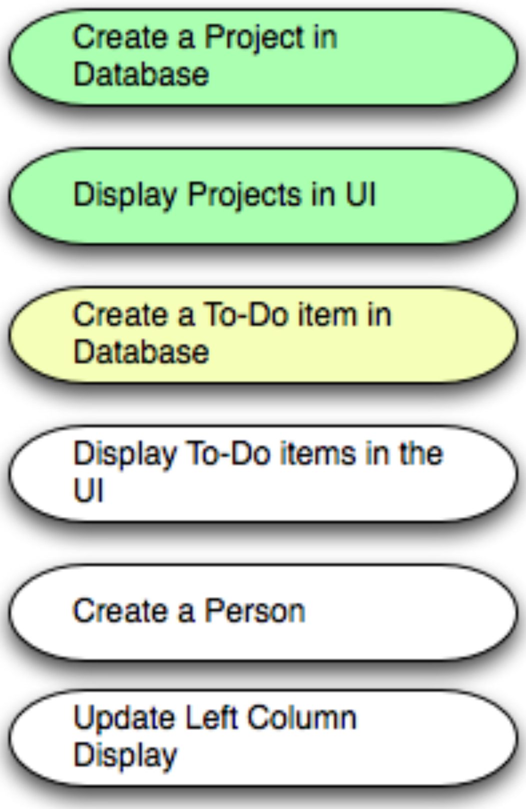


uiPopulateProjectsAndPeople

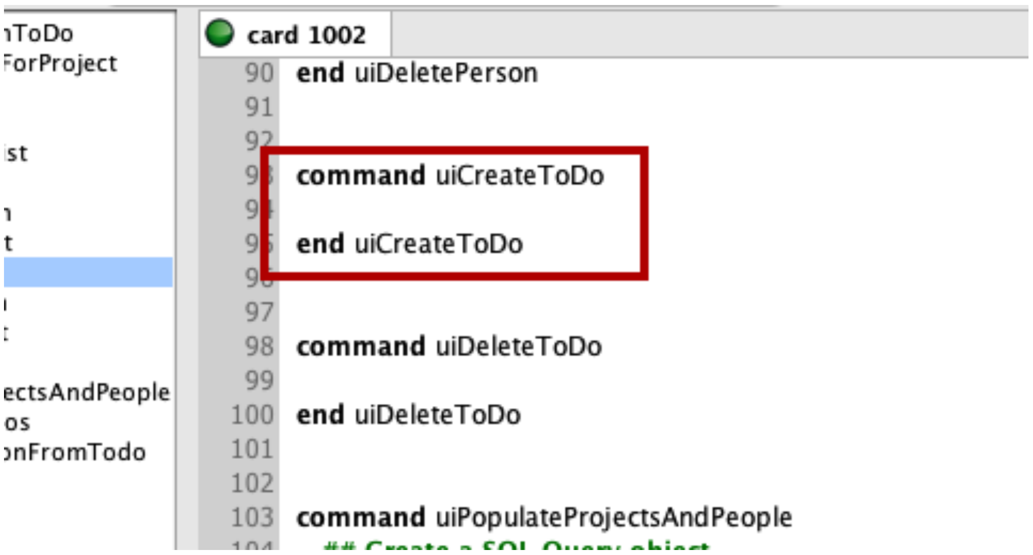
To test, open the Message Box and type in **uiPopulateProjectsAndPeople** (1) and press the **Return** key. You should see the text **New Project** appear in the Data Grid (2).

Creating a To-Do Item

Now let's look at how to create to-do items. The procedure is pretty much the same as it is for a project.


Create a Project in Database
Display Projects in UI
Create a To-Do item in Database
Display To-Do items in the UI
Create a Person
Update Left Column Display

Locate uiCreateToDo in the Card Script



```
end uiDeletePerson


command uiCreateToDo

end uiCreateToDo


command uiDeleteToDo

end uiDeleteToDo


command uiPopulateProjectsAndPeople
```

In the card script you will find a **uiCreateToDo** command.

Update uiCreateToDo Command

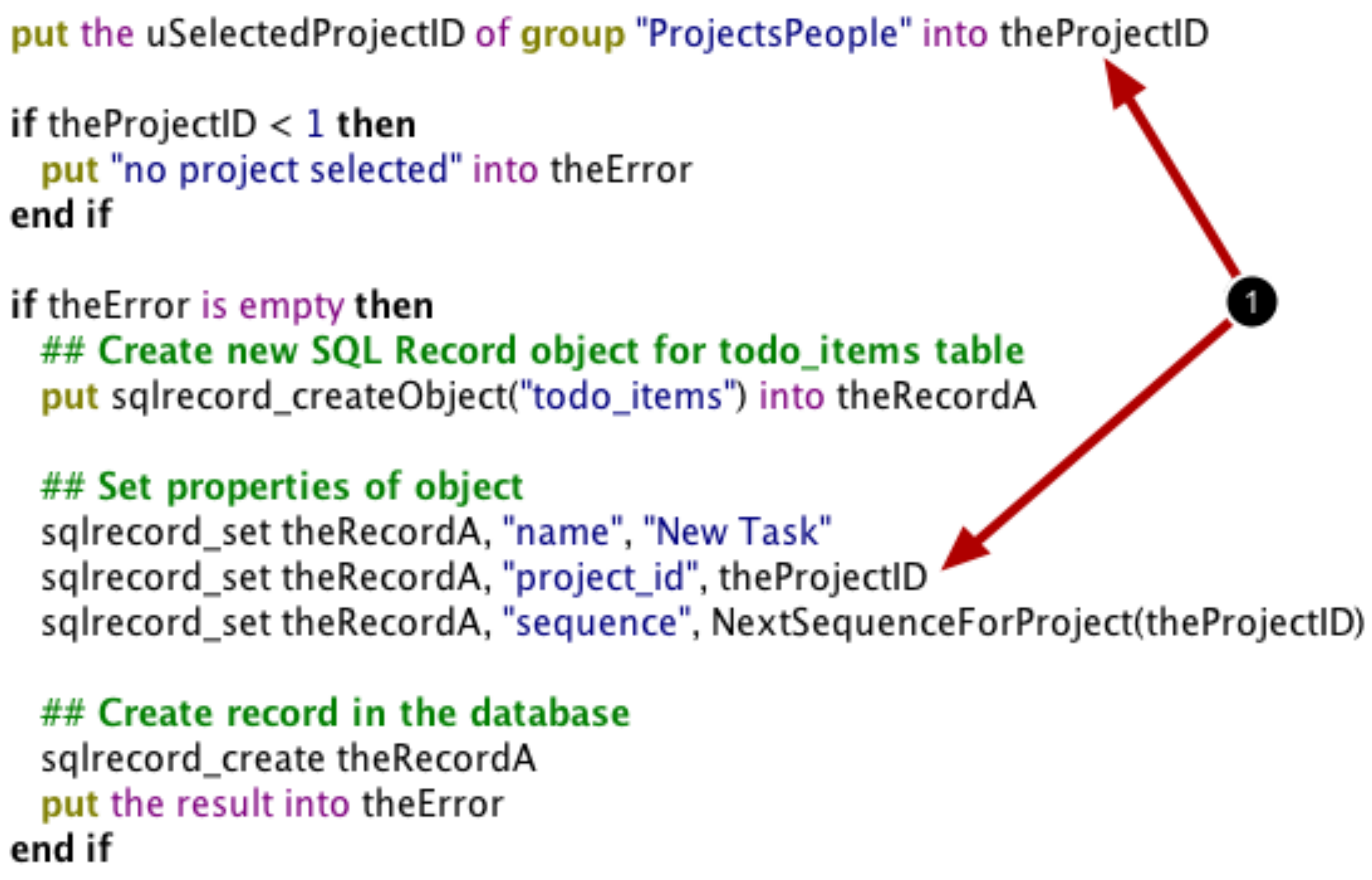
Replace the uiCreateToDo command in the card script with the RevTalk code below. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command uiCreateToDo
   put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

   if theProjectID < 1 then
      put "no project selected" into theError
   end if

   if theError is empty then
      ## Create new SQL Record object for todo_items table
      put sqlrecord_createObject("todo_items") into theRecordA

      ## Set properties of object
      sqlrecord_set theRecordA, "name", "New Task"
      sqlrecord_set theRecordA, "project_id", theProjectID
      sqlrecord_set theRecordA, "sequence", NextSequenceForProject(theProjectID)

      ## Create record in the database
      sqlrecord_create theRecordA
      put the result into theError
   end if

   ## Add new record to Data Grid and open for editing
   if theError is empty then
      ## Refresh list, select new record and open field editor
      ## so that user can change name.
      lock screen
      RefreshToDoList
      set the uSelectedID of group "ToDo" to theRecordA["id"]
      set the dgTemplateFieldEditor["select text"] of group "ToDo" to true
      dispatch "EditKeyOfIndex" to group "ToDo" with "name", the dgHilitedIndex of group "ToDo"
      unlock screen
```

```
    end if

    if theError is not empty then
        answer "Error creating to-do item:" && theError & "."
    end if
end uiCreateToDo
```

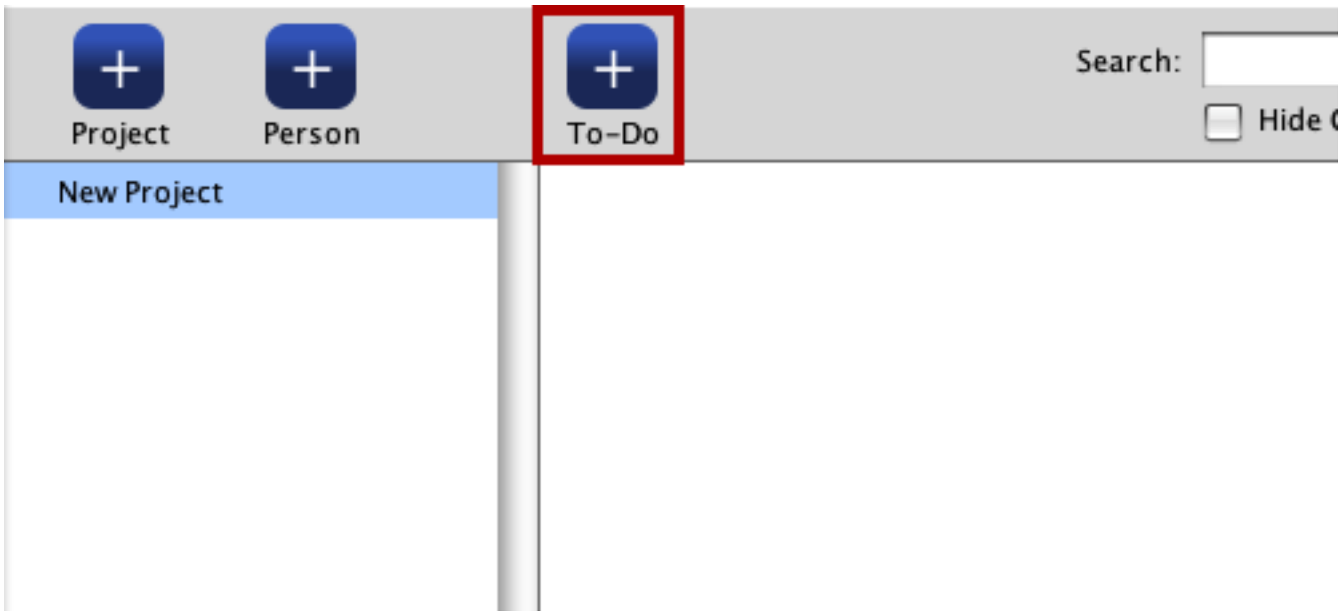
Creating a To-Do Item



```
put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

if theProjectID < 1 then
  put "no project selected" into theError
end if

if theError is empty then
  ## Create new SQL Record object for todo_items table
  put sqlrecord_createObject("todo_items") into theRecordA

  ## Set properties of object
  sqlrecord_set theRecordA, "name", "New Task"
  sqlrecord_set theRecordA, "project_id", theProjectID
  sqlrecord_set theRecordA, "sequence", NextSequenceForProject(theProjectID)

  ## Create record in the database
  sqlrecord_create theRecordA
  put the result into theError
end if
```

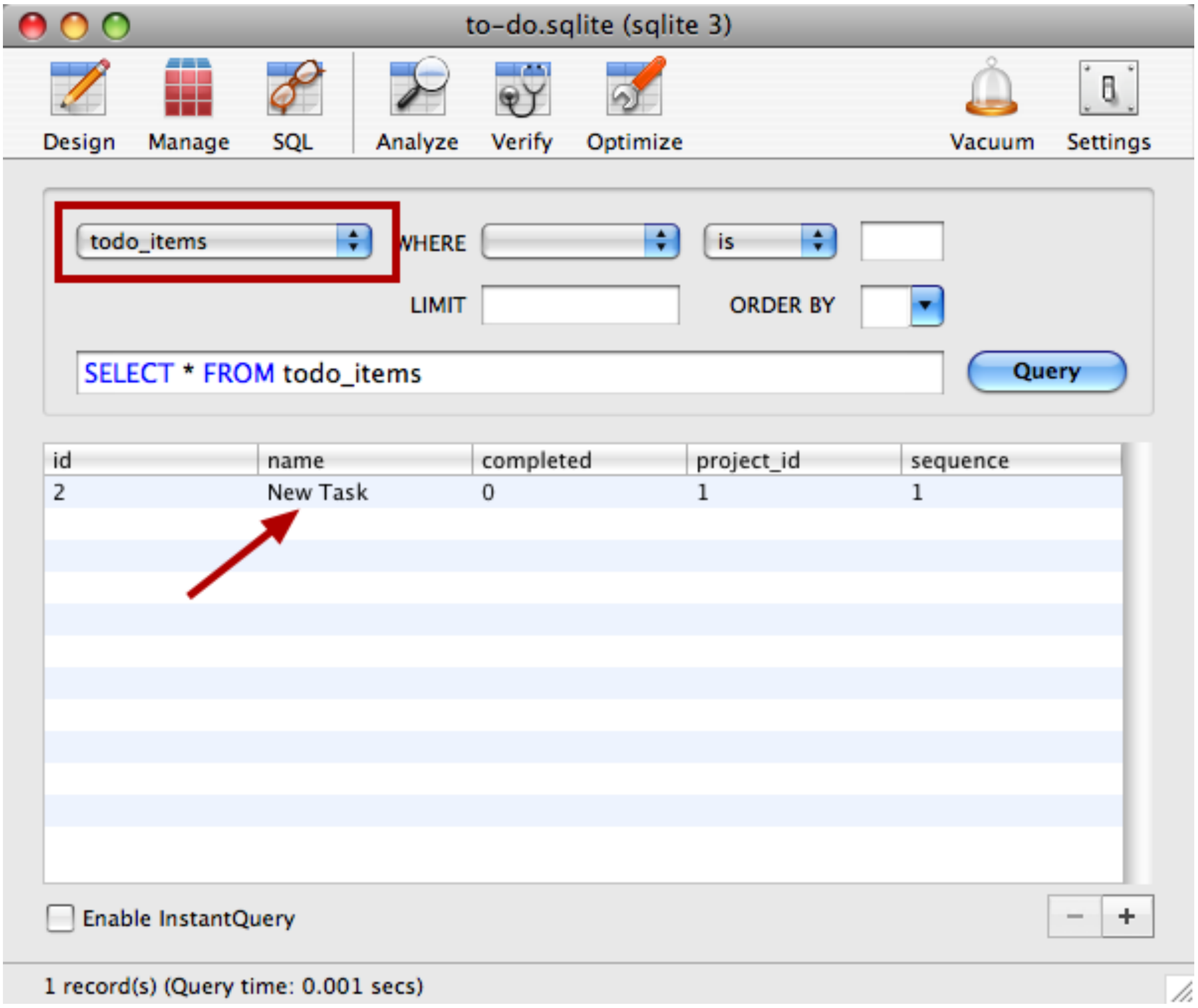
This command needs to add a to-do item to the todo_items table in the database. The steps are the same as the ones you followed to create a project. The only difference is that when creating a to-do item you must link it to a project (1) using the id of the currently selected project.

Test Record Creation In The Database



With **New Project** selected in the left column click the **Add To-Do** button. This button has been hooked up to call uiCreateToDo. **YOU WON'T GET ANY VISUAL FEEDBACK AT THIS POINT**.

Verify Result


to-do.sqlite (sqlite 3)

```
SELECT * FROM todo_items
```

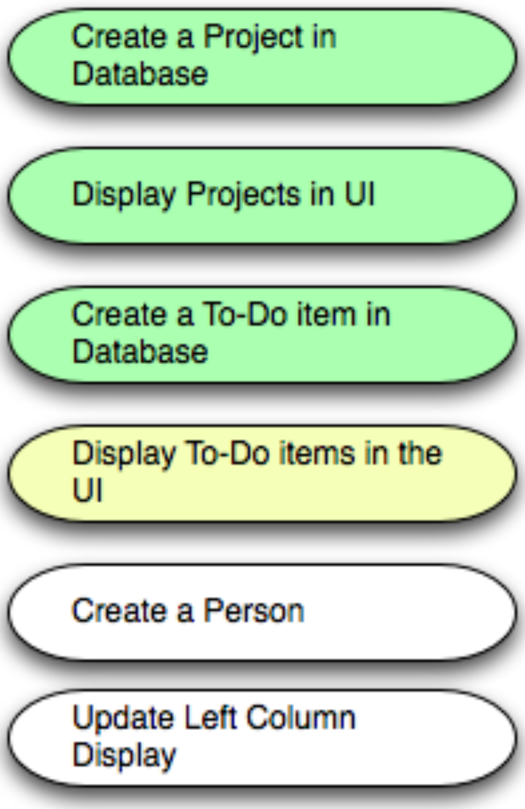
id	name	completed	project_id	sequence
2	New Task	0	1	1

1 record(s) (Query time: 0.001 secs)

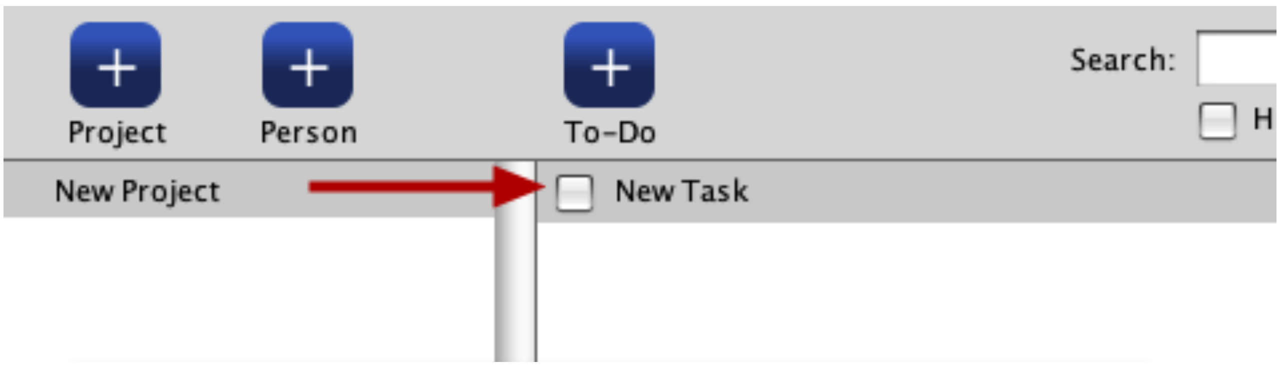
You can verify that the new record was created using your SQLite manager program. Look at the records for the todo_items table.

Display To-Do Items in the UI

Now we will look at how to display the to-do items that we can create.


Create a Project in Database
Display Projects in UI
Create a To-Do item in Database
Display To-Do items in the UI
Create a Person
Update Left Column Display

What We Are Going To Do



When the user clicks on a project in the left column a list of to-do items should appear to the right. We are now going to add the code that does that.

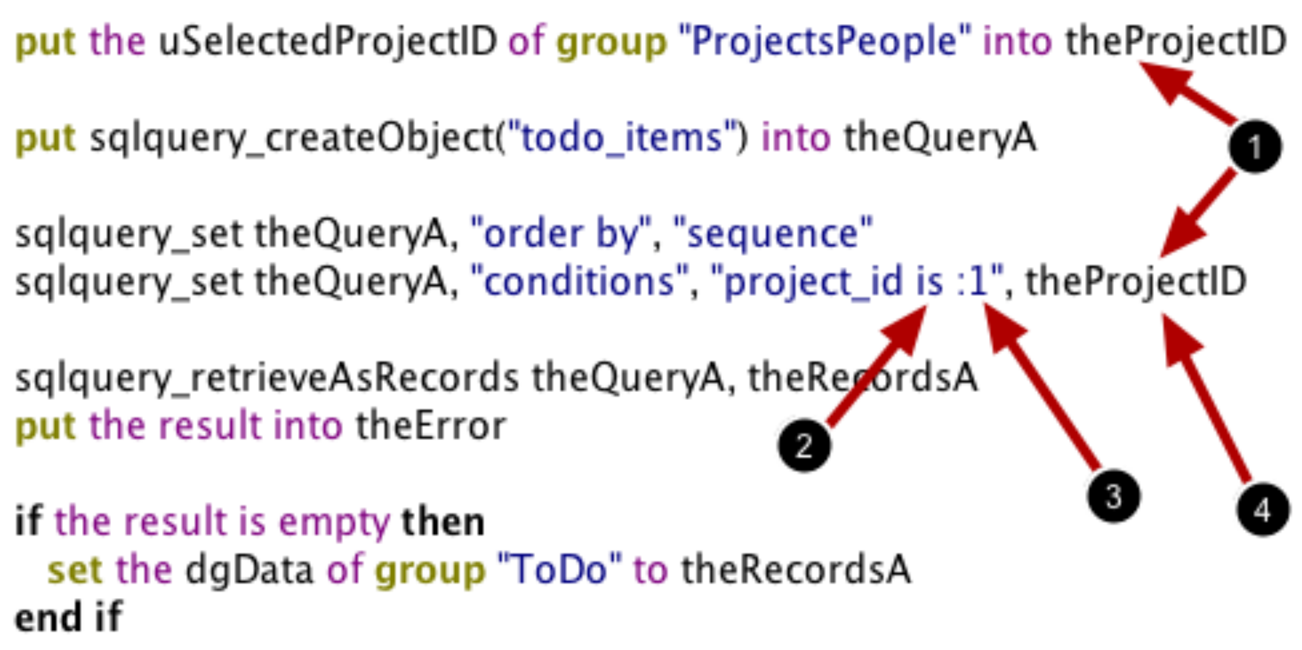
Update the uiPopulateToDos Command

Replace the uiPopulateToDos command in the card script with the RevTalk code below. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command uiPopulateToDos
   put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

   put sqlquery_createObject("todo_items") into theQueryA

   sqlquery_set theQueryA, "order by", "sequence"
   sqlquery_set theQueryA, "conditions", "project_id is :1", theProjectID

   sqlquery_retrieveAsRecords theQueryA, theRecordsA
   put the result into theError

   if the result is empty then
      set the dgData of group "ToDo" to theRecordsA
   end if

   if theError is not empty then
      answer "Error populating to-do items:" && theError & "."
   end if
end uiPopulateToDos
```

Using the 'conditions' Property of a SQL Query Object



```
put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

put sqlquery_createObject("todo_items") into theQueryA

sqlquery_set theQueryA, "order by", "sequence"
sqlquery_set theQueryA, "conditions", "project_id is :1", theProjectID

sqlquery_retrieveAsRecords theQueryA, theRecordsA
put the result into theError

if the result is empty then
  set the dgData of group "ToDo" to theRecordsA
end if
```

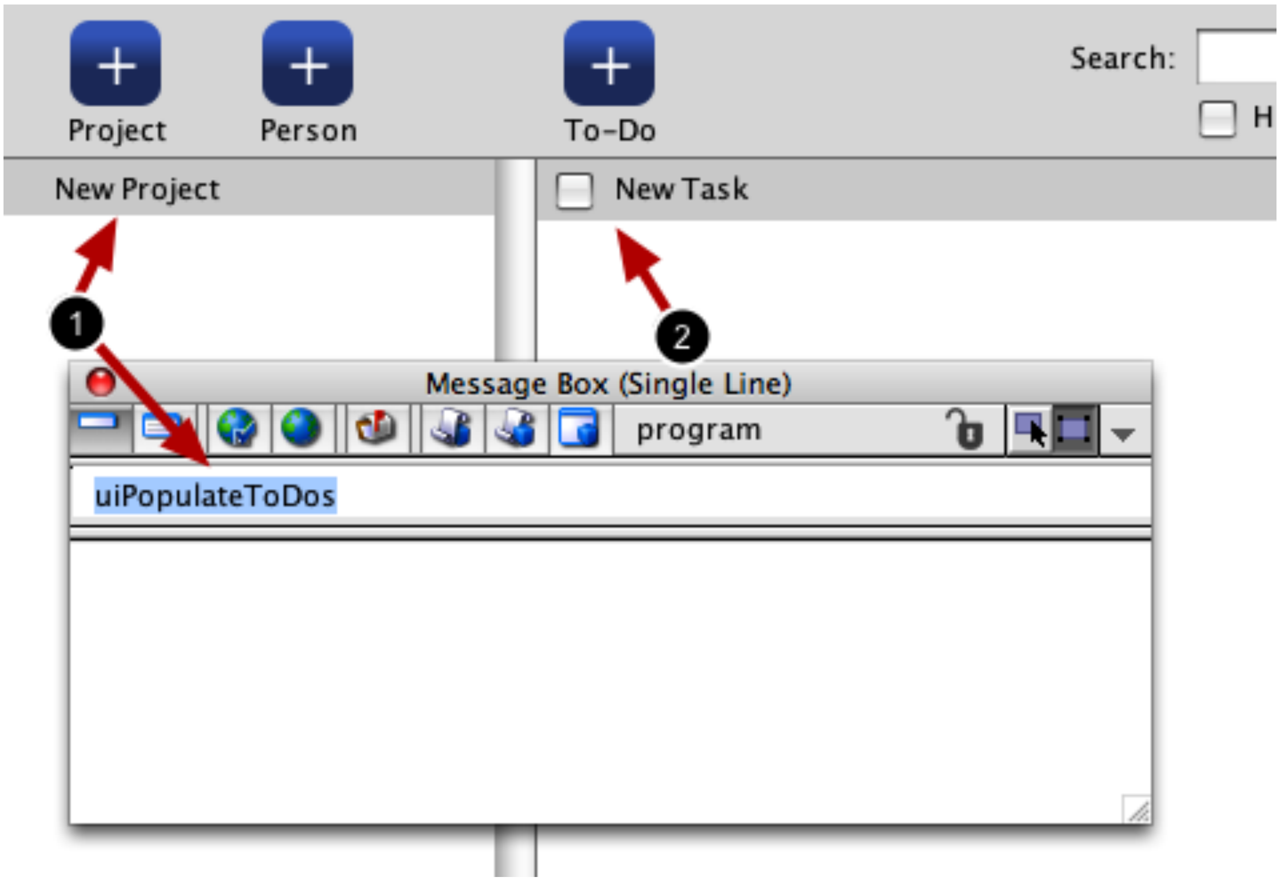
The RevTalk that displays to-do items is very similar to the RevTalk that displays projects. The only real difference is that you need to set the **conditions** property of the SQL Query object in order to filter the list of to-do items that are returned.

The 'conditions' property is part of what defines the WHERE clause of the SQL query that is generated. Since to-do items are linked to projects we only want to retrieve to-do items that are linked to the currently selected project (1).

I would like to point out two things in the line that sets the 'conditions' property. First is that you can use english search terms. For example, I use 'is' rather than '=' (2). You can also use terms like 'begins with', 'ends with' and 'is in'.

Second is that when setting the 'conditions' property you can using bindings. Notice the ':1' in the string (3). Using bindings can make your code easier to read. In addition, SQL Yoga will cleanse special characters that appear in strings that you add using bindings. The ':1' in the string is replaced by the next parameter passed to sqlquery_set (4). I could also include a ':2' or ':3' in the string if I passed in additional parameters.

Verify

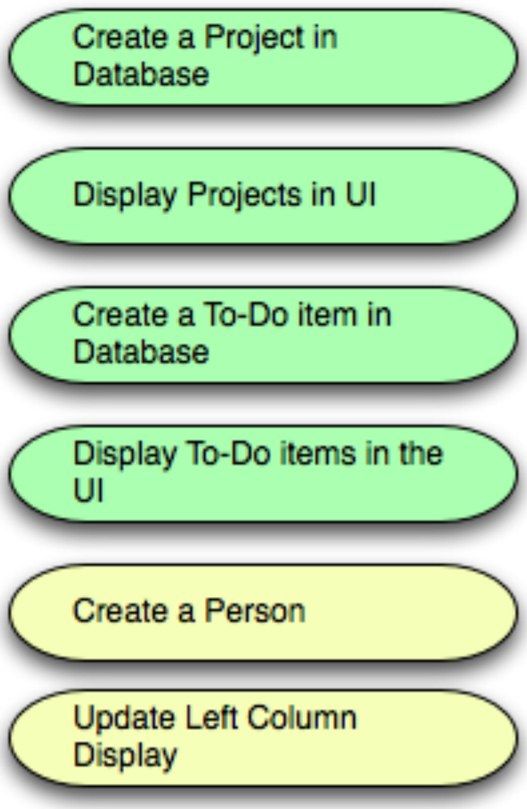


```
uiPopulateToDos
```

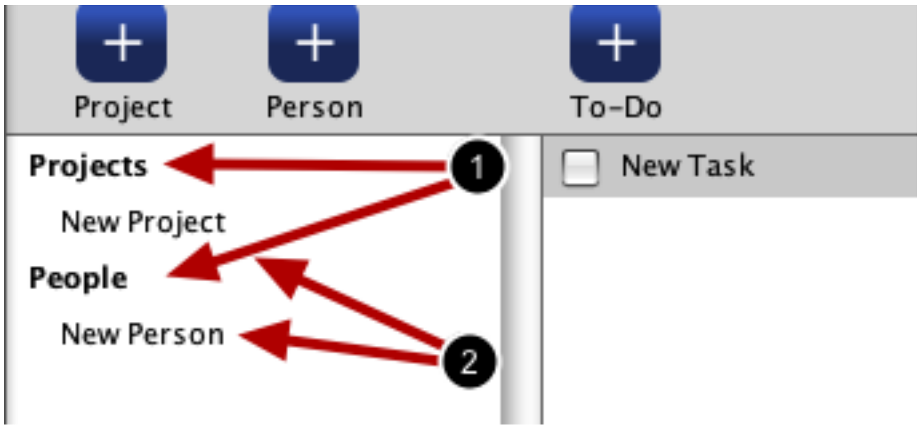
To verify that the to-do item was created click on **New Projects** and execute uiPopulateToDos in the message box (1). You should see the **New Task** entry appear in the to-do list (2).

Creating A Person And Updating Left Column Display

Now that we can create and view projects and to-do items let's look at creating people. People are going to be displayed in the left column along with projects so we will make some changes to the uiPopulateProjectsAndPeople handler in order to accommodate that.


Create a Project in Database
Display Projects in UI
Create a To-Do item in Database
Display To-Do items in the UI
Create a Person
Update Left Column Display

What We Are Going To Do



In order to display both Projects and People in the left column we will create headings (1). Underneath the headings the list of projects or people will appear (2).

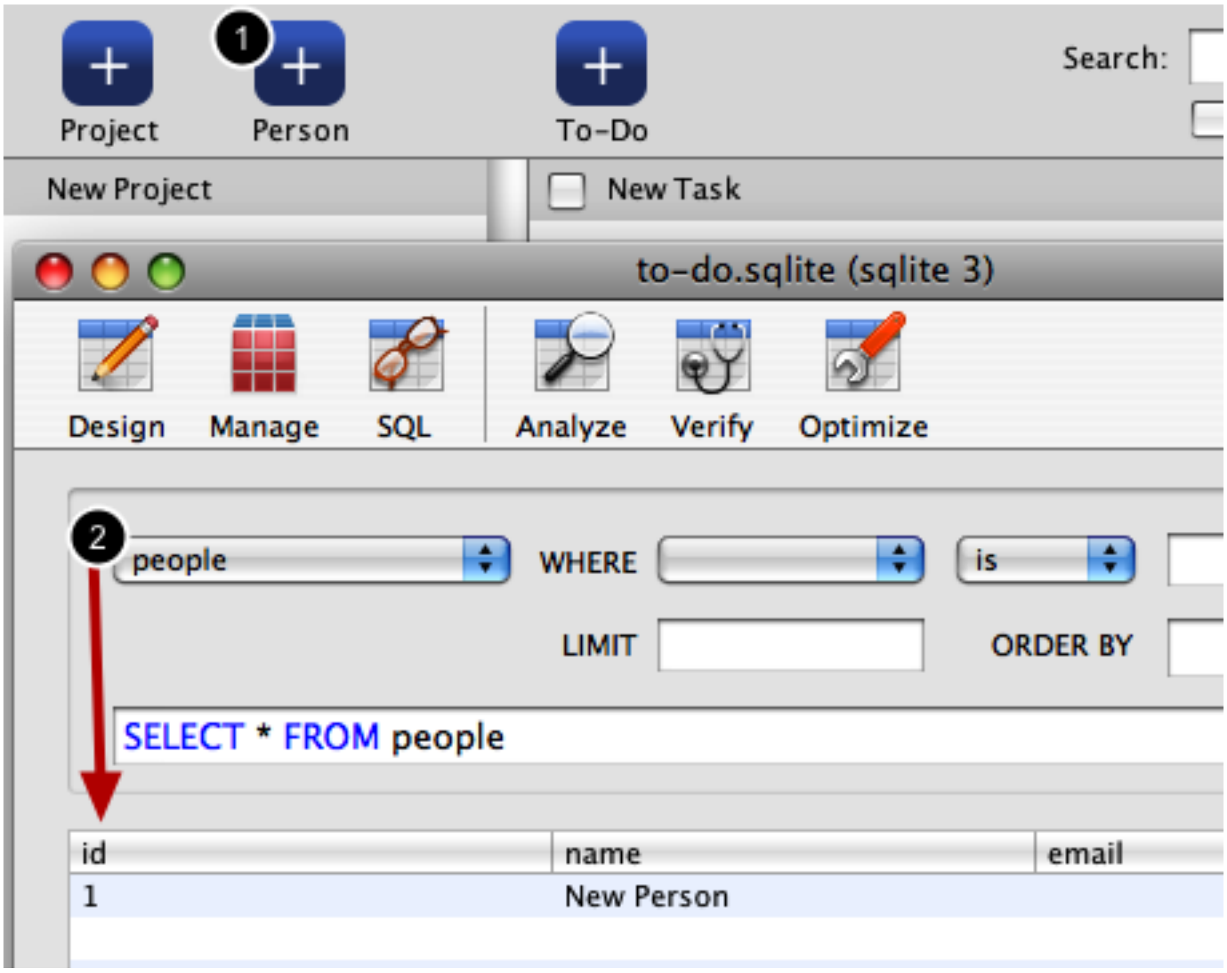
Update uiCreatePerson Handler

Replace the uiCreatePerson command in the card script with the script below. There is nothing going on in this handler that you haven't seen before.

Copy & Paste The Following Code

```
command uiCreatePerson
   ## Create a SQL Record object for 'people' table
   put sqlrecord_createObject("people") into theRecordA

   ## Set name property of object
   sqlrecord_set theRecordA, "name", "New Person"

   ## Create record in the database
   sqlrecord_create theRecordA
   put the result into theError

   if theError is empty then
      ## Refresh list, select new record and open field editor
      ## so that user can change name.
      lock screen
      RefreshProjectsPeopleList
      set the uSelectedPersonID of group "ProjectsPeople" to theRecordA["id"]
      set the dgTemplateFieldEditor["select text"] of group "ProjectsPeople" to true
      dispatch "EditKeyOfIndex" to group "ProjectsPeople" with "name", the dgHilitedIndex of group
"ProjectsPeople"
      unlock screen
   end if

   if theError is not empty then
      answer "Error creating person:" && theError & "."
   end if
end uiCreatePerson
```

Test Record Creation In The Database



```
SELECT * FROM people
```

id	name	email
1	New Person	

After compiling the card script you can click on the **Add Person** button (1). You should see a new record in your database manager for the **people** table (2). **YOU WON'T GET ANY VISUAL FEEDBACK AT THIS POINT**.

Update the uiPopulateProjectsAndPeople Handler

Replace the uiPopulateProjectsAndPeople command in the card script with the RevTalk code below. After you insert the code we will go through the relevant parts.

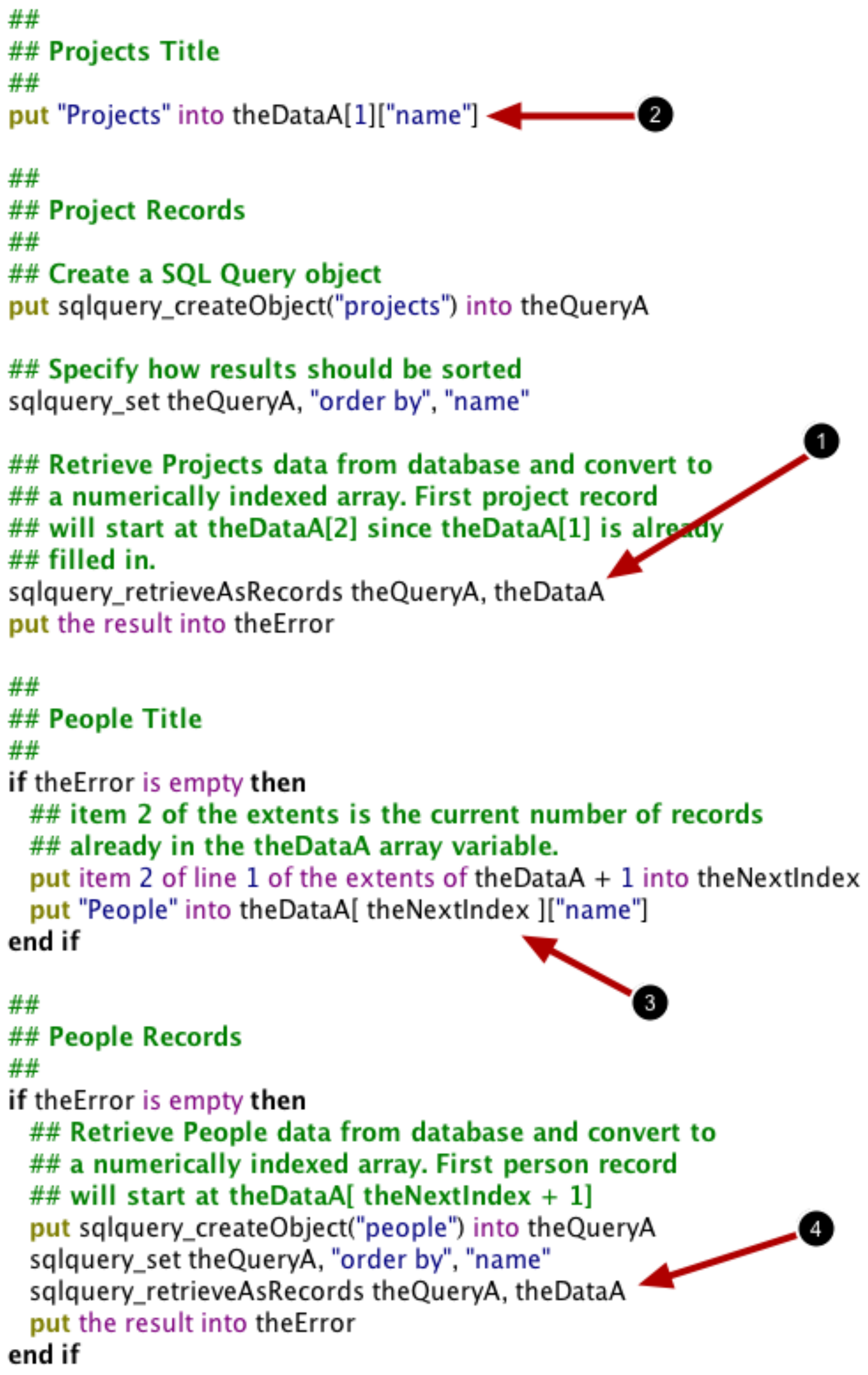
Copy & Paste The Following Code

```
command uiPopulateProjectsAndPeople
    ##
    ## Projects Title
    ##
```

```
put "Projects" into theDataA[1]["name"]

##
## Project Records
##
## Create a SQL Query object
put sqlquery_createObject("projects") into theQueryA

## Specify how results should be sorted
sqlquery_set theQueryA, "order by", "name"

## Retrieve Projects data from database and convert to
## a numerically indexed array. First project record
## will start at theDataA[2] since theDataA[1] is already
## filled in.
sqlquery_retrieveAsRecords theQueryA, theDataA
put the result into theError

##
## People Title
##
if theError is empty then
   ## item 2 of the extents is the current number of records
   ## already in the theDataA array variable.
   put item 2 of line 1 of the extents of theDataA + 1 into theNextIndex
   put "People" into theDataA[ theNextIndex ]["name"]
end if

##
## People Records
##
if theError is empty then
   ## Retrieve People data from database and convert to
   ## a numerically indexed array. First person record
   ## will start at theDataA[ theNextIndex + 1]
   put sqlquery_createObject("people") into theQueryA
   sqlquery_set theQueryA, "order by", "name"
   sqlquery_retrieveAsRecords theQueryA, theDataA
   put the result into theError
end if
```

```
    if theError is empty then
        ## Assign the array directly to a Data Grid
        set the dgData of group "ProjectsPeople" to theDataA
    end if

    if theError is not empty then
        answer "Error populating projects and people:" && theError & "."
    end if
end uiPopulateProjectsAndPeople
```

Aggregating Static Data And Multiple Database Queries Into a Single Array



```
##
## Projects Title
##
put "Projects" into theDataA[1]["name"]

##
## Project Records
##
## Create a SQL Query object
put sqlquery_createObject("projects") into theQueryA

## Specify how results should be sorted
sqlquery_set theQueryA, "order by", "name"

## Retrieve Projects data from database and convert to
## a numerically indexed array. First project record
## will start at theDataA[2] since theDataA[1] is already
## filled in.
sqlquery_retrieveAsRecords theQueryA, theDataA
put the result into theError

##
## People Title
##
if theError is empty then
  ## item 2 of the extents is the current number of records
  ## already in the theDataA array variable.
  put item 2 of line 1 of the extents of theDataA + 1 into theNextIndex
  put "People" into theDataA[ theNextIndex ]["name"]
end if

##
## People Records
##
if theError is empty then
  ## Retrieve People data from database and convert to
  ## a numerically indexed array. First person record
  ## will start at theDataA[ theNextIndex + 1]
  put sqlquery_createObject("people") into theQueryA
  sqlquery_set theQueryA, "order by", "name"
  sqlquery_retrieveAsRecords theQueryA, theDataA
  put the result into theError
end if
```

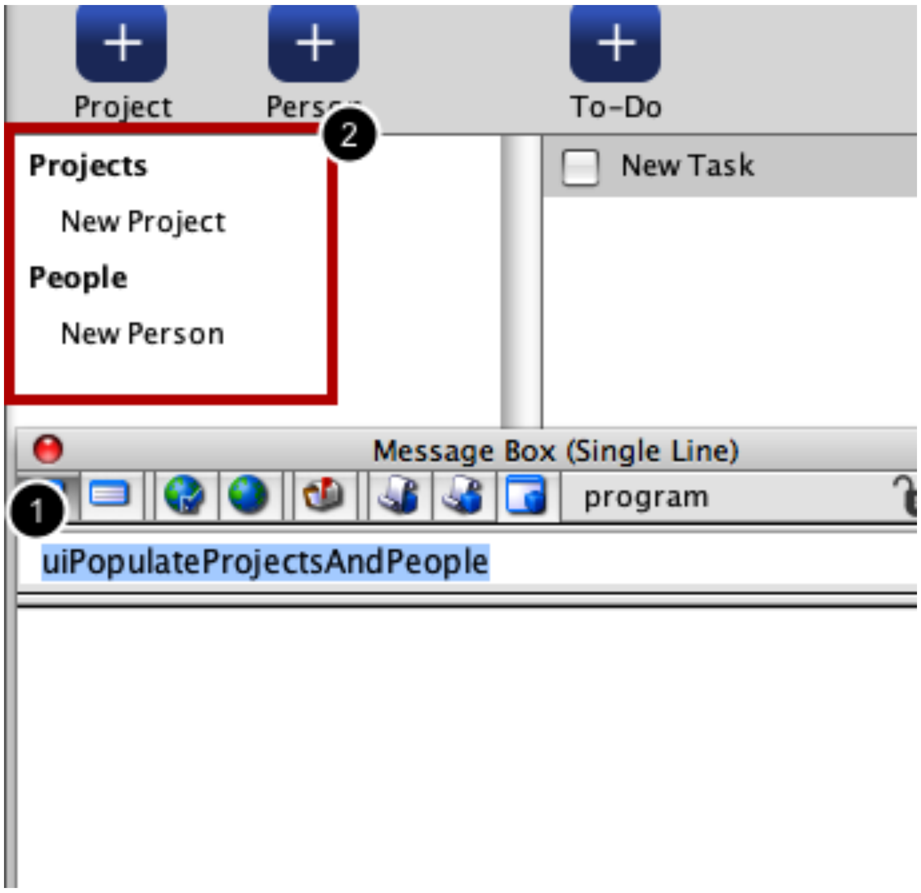
Since we are going to display both projects and people in the same Data Grid we need to update the uiPopulateProjectsAndPeople handler. SQL Yoga makes it easy to aggregate results across multiple queries when retrieving data as an array or as SQL Record objects.

If you pass a numerically indexed array to sqlquery_retrieveAsRecords, SQL Yoga will append the results to the array. For example, before retrieving projects, the theDataA array variable has a nested array assigned to index [1] (2). Because theDataA already has a numeric index of 1 sqlquery_retrieveAsRecords will start inserting records into the array starting with [2].

The People title string is then appended to the end of the theDataA array variable (3) and then the people records are finally appended as well (4).

In the end you have a single, numerically indexed array that you can assign to a Data Grid.

Test



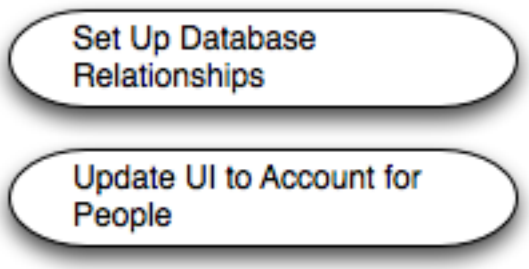
```
uiPopulateProjectsAndPeople
```

Open the Message Box and execute the command **uiPopulateProjectsAndPeople** (1) to see the result (2).

More With Displaying Database Records

Chapter Overview

Here are the major subjects we are going to cover in this chapter.

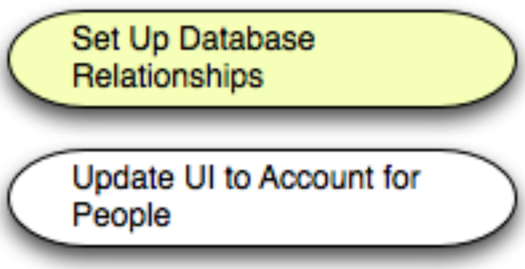

Set Up Database Relationships
Update UI to Account for People

Tell SQL Yoga About Table Relationships

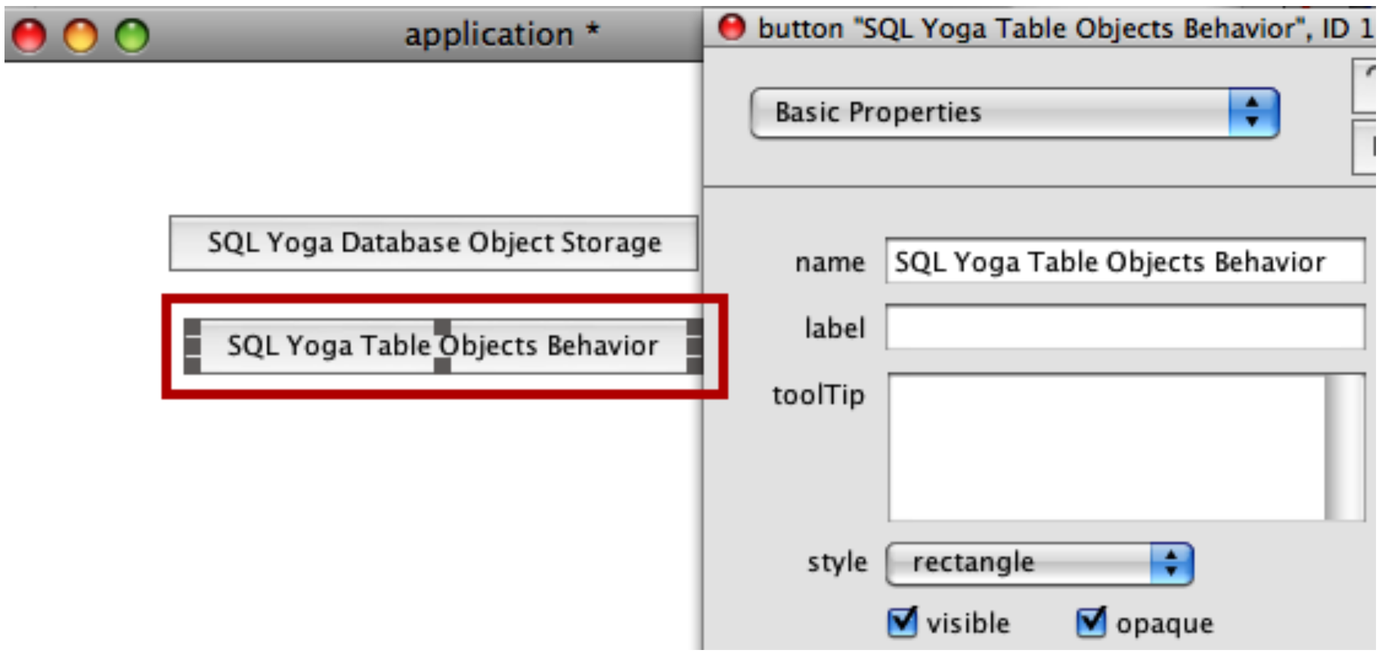
So far we have created records in and displayed records from a database with very little code. SQL Yoga allowed you to create objects that you set properties on rather than writing out SQL queries by hand.

Now we are going to extend the basic functionality that SQL Yoga provides by defining a **table objects behavior** for the Database object where we can define special objects that provide extra functionality. We will start by looking at Table and Relationship objects because SQL Yoga will make working with people and to-do items much easier if we tell it about the relationship.

Let's look at how to create and define Table and Relationship objects.


Set Up Database Relationships
Update UI to Account for People

Create The 'table objects behavior'

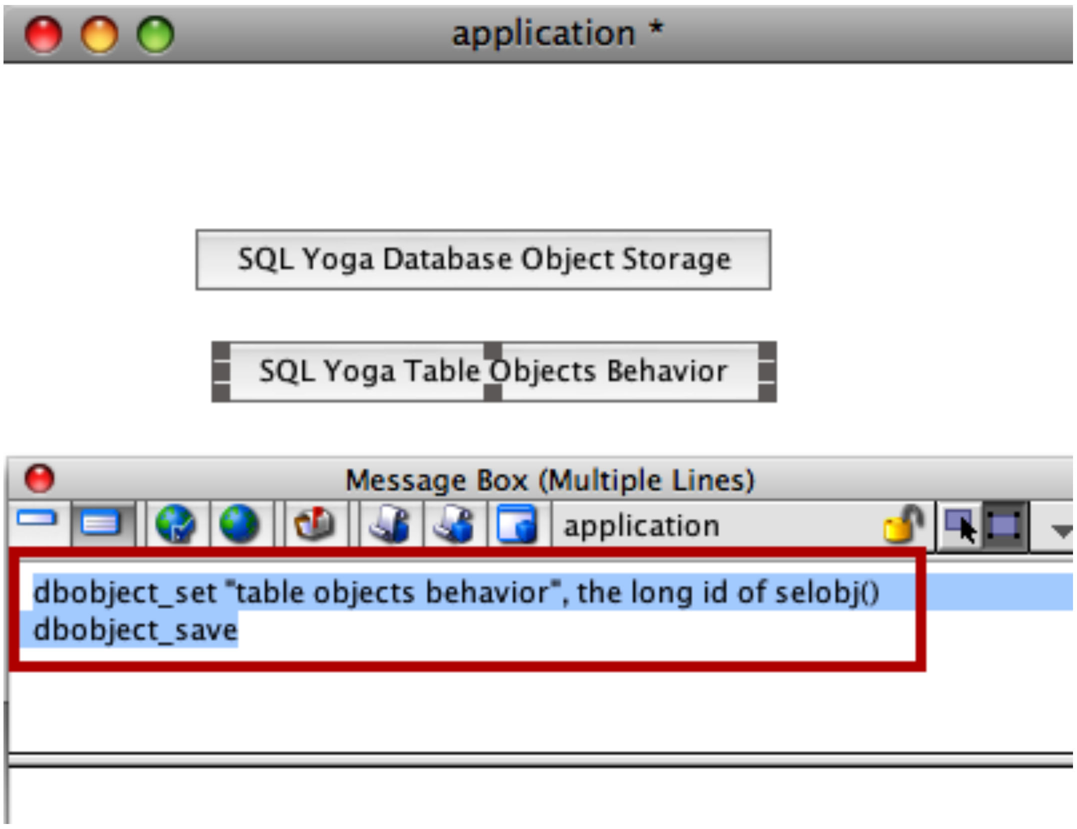

application *
button "SQL Yoga Table Objects Behavior", ID 1
Basic Properties
SQL Yoga Database Object Storage
SQL Yoga Table Objects Behavior

name: SQL Yoga Table Objects Behavior
label:
toolTip:
style: rectangle
☑ visible ☑ opaque

In Revolution a **behavior** script extends the functionality of the object that it is attached to. SQL Yoga allows you to set a behavior for a Database object so that you can enable additional features.

Revolution stores a behavior script in a button control so open the **application** stack again and add a new button to it. Name the button **SQL Yoga Table Objects Behavior**.

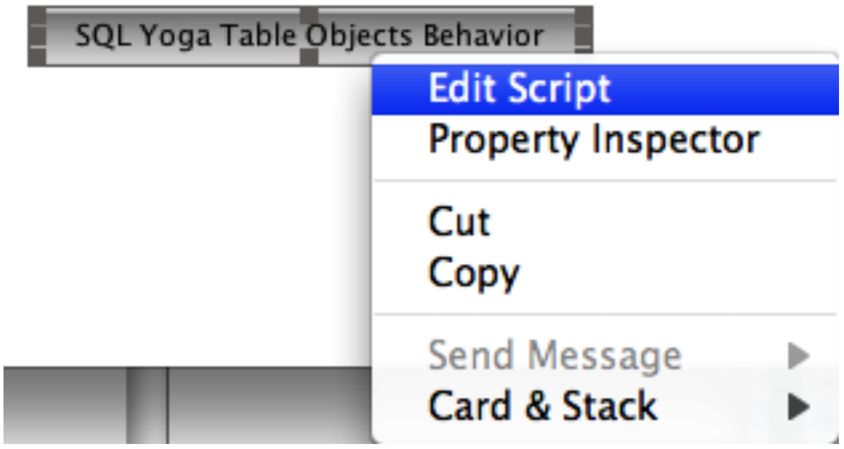
Set the 'table objects behavior' Property of the Database Object


application *
SQL Yoga Database Object Storage
SQL Yoga Table Objects Behavior
Message Box (Multiple Lines)
application

```
dbobject_set "table objects behavior", the long id of selobj()
dbobject_save
```

Next, set the **table objects behavior** property of the Database object to the long id of the **SQL Yoga Table Objects Behavior** button and save the Database object. The 'table objects behavior' property tells SQL Yoga which button contains the behavior script that will extend the functionality of the Database object.

Edit Behavior Script



Now that you have assigned the table objects behavior of the Database object you can begin to write the behavior script. Edit the script of the **SQL Yoga Table Objects Behavior** script.

Set Behavior Script

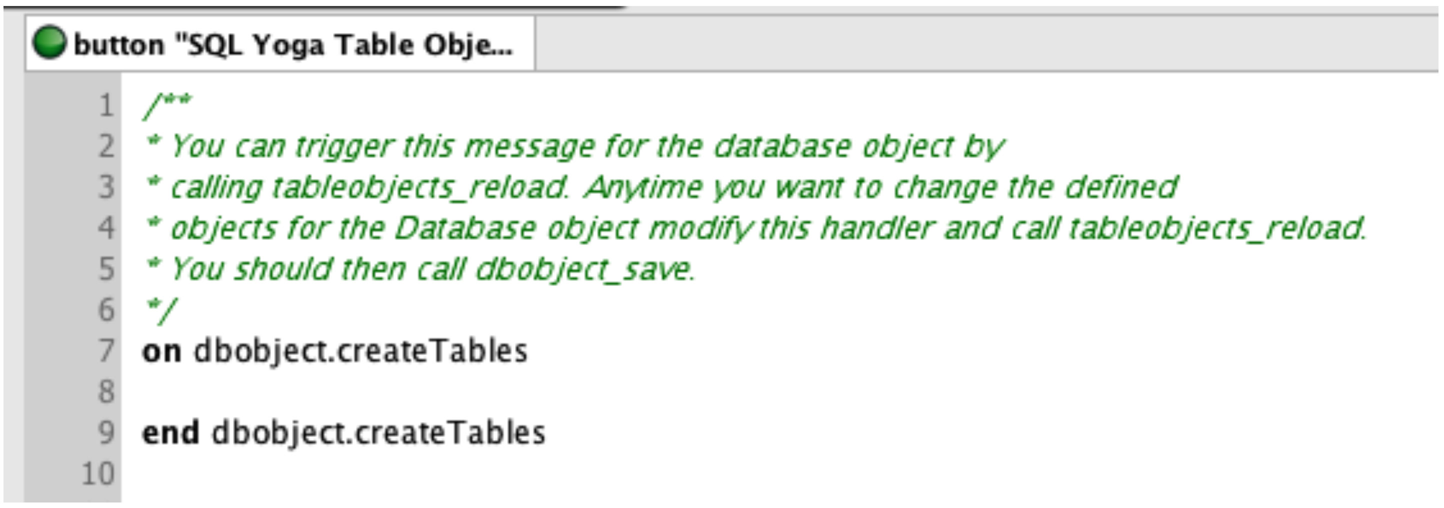
Paste the following RevTalk code into the **SQL Yoga Table Objects Behavior** button script and compile. If there is a mouseUp handler in the script make sure and delete it. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
/**
* You can trigger this message for the database object by
* calling tableobjects_reload. Anytime you want to change the defined
* objects for the Database object modify this handler and call tableobjects_reload.
* You should then call dbobject_save.
*/
on dbobject.createTables
   ## Create Table Objects in order to define
   ## relationships
   tableobj_createObject "projects"
   tableobj_createObject "todo_items"
   tableobj_createObject "people"

   ## Now that table objects exist create relationships
   _CreateRelationships
```

```
end dbobject.createTables


private command _CreateRelationships
   ----------
   ## Define relationship between projects and to-do items
   tblrelation_createObject "projects to todo items"
   tblrelation_set "projects to todo items", "type", "one-to-many"

   tblrelation_set "projects to todo items", "left table", "projects"
   tblrelation_set "projects to todo items", "left table key", "id"

   tblrelation_set "projects to todo items", "right table", "todo_items"
   tblrelation_set "projects to todo items", "right table key", "project_id"

   tblrelation_set "projects to todo items", "order by", "todo_items.sequence"
   ----------


   ----------
   ## Define relationship between people and to-do items
   tblrelation_createObject "people to todo items"
   tblrelation_set "people to todo items", "type", "many-to-many"

   tblrelation_set "people to todo items", "left table", "people"
   tblrelation_set "people to todo items", "left table key", "id"

   tblrelation_set "people to todo items", "cross-reference table", "people_todo"
   tblrelation_set "people to todo items", "cross-reference table key for left table", "people_id"
   tblrelation_set "people to todo items", "cross-reference table key for right table", "todo_id"

   tblrelation_set "people to todo items", "right table", "todo_items"
   tblrelation_set "people to todo items", "right table key", "id"

   tblrelation_set "people to todo items", "order by", "todo_items.name"
   ----------
end _CreateRelationships
```

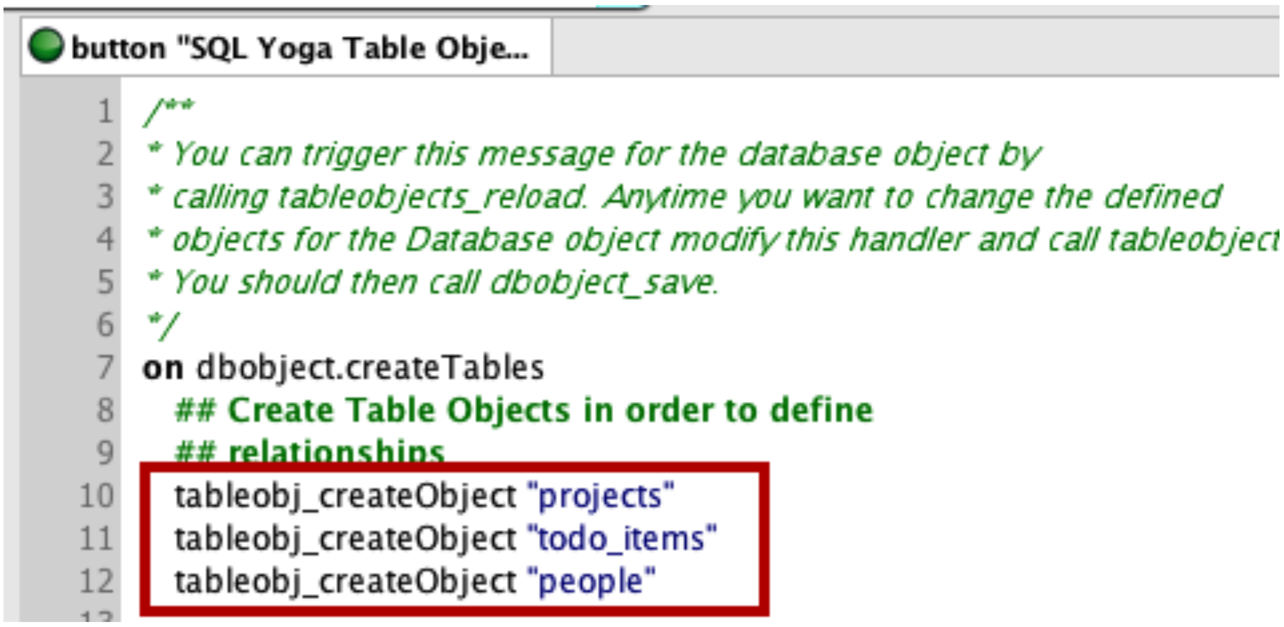
Defining the dbobject.createTables Message For a Database Object



```
/**
* You can trigger this message for the database object by
* calling tableobjects_reload. Anytime you want to change the defined
* objects for the Database object modify this handler and call tableobjects_reload.
* You should then call dbobject_save.
*/
on dbobject.createTables

end dbobject.createTables
```

With SQL Yoga you define all of the objects that extend functionality of your Database object in a special message named **dbobject.createTables**. Within this message you create the Table, Relationship, Scope and SQL Query Templates that your application will use.

Create Table Objects



```
/**
* You can trigger this message for the database object by
* calling tableobjects_reload. Anytime you want to change the defined
* objects for the Database object modify this handler and call tableobject
* You should then call dbobject_save.
*/
on dbobject.createTables
  ## Create Table Objects in order to define
  ## relationships
  tableobj_createObject "projects"
  tableobj_createObject "todo_items"
  tableobj_createObject "people"
```

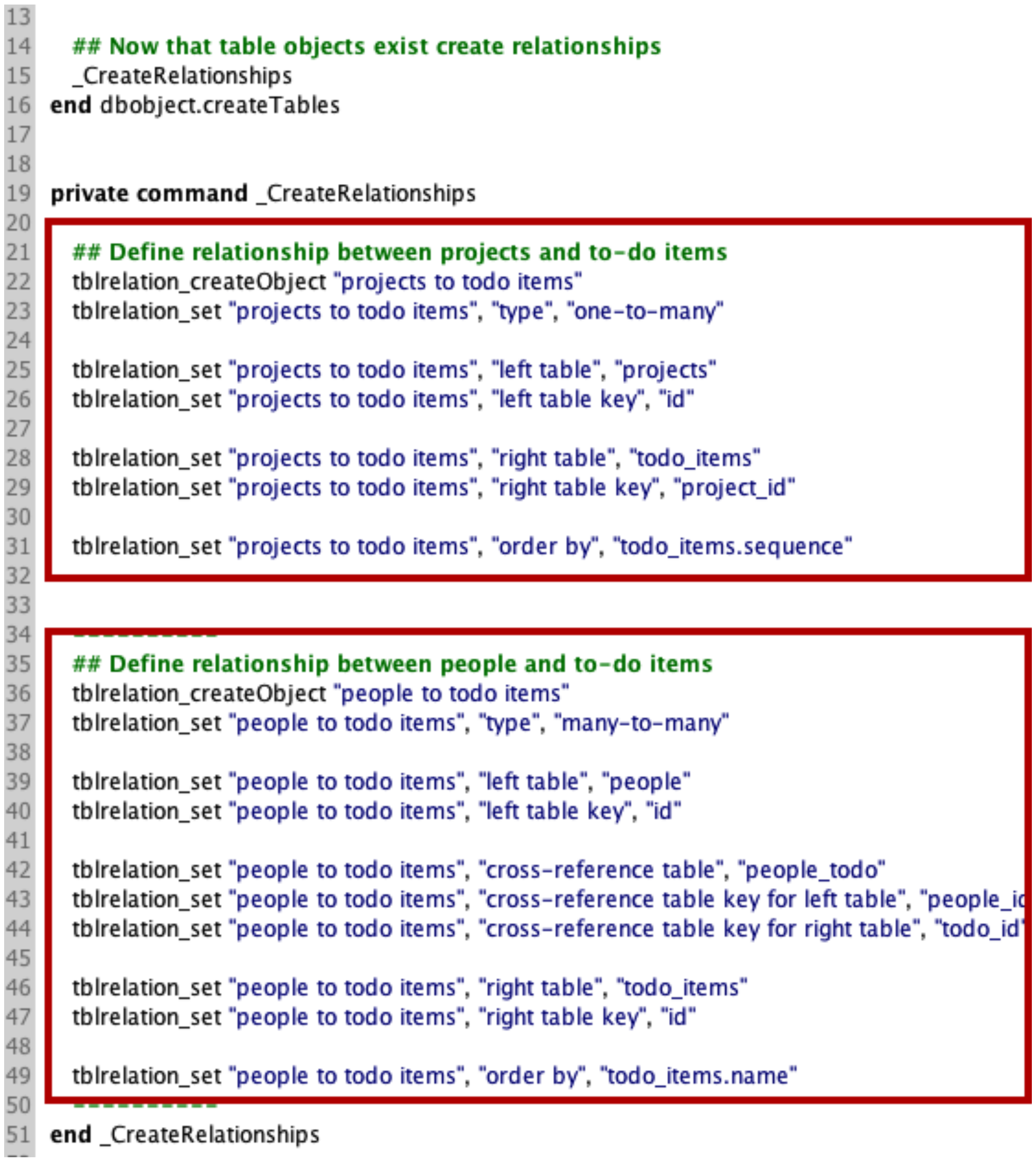
Since we are going to define relationships between various tables we need to create what are called Table objects. A Table object represents a table in your database and allows you to extend the basic properties of that table.

For example, you can define additional properties for a table using a Table object. Say you have a table with first_name and last_name fields. You could define a 'name' property for the table object that returned a concatenation of those two fields.

Or perhaps you have a table with a description field that can contain a lot of text. You could define a 'short description' property that returned a shortened version of the description.

Right now we are just going to extend the projects, todo_items and peoples tables by defining the relationships between them. To do this you create a Table object for each table by calling **tableobj_createObject**. This command will add an object to the Database object.

Create Relationship Objects

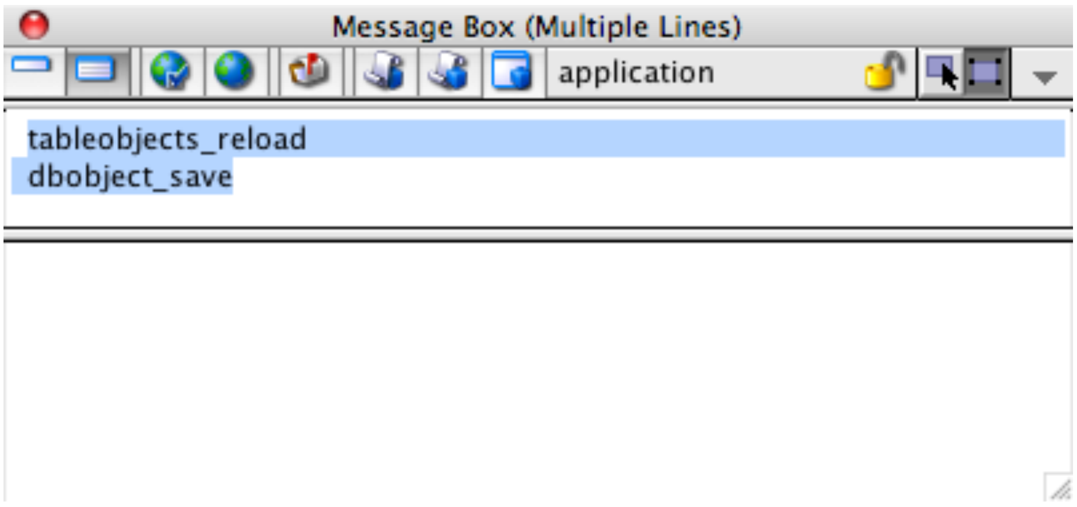


```

  ## Now that table objects exist create relationships
  _CreateRelationships
end dbobject.createTables


private command _CreateRelationships

  ## Define relationship between projects and to-do items
  tblrelation_createObject "projects to todo items"
  tblrelation_set "projects to todo items", "type", "one-to-many"

  tblrelation_set "projects to todo items", "left table", "projects"
  tblrelation_set "projects to todo items", "left table key", "id"

  tblrelation_set "projects to todo items", "right table", "todo_items"
  tblrelation_set "projects to todo items", "right table key", "project_id"

  tblrelation_set "projects to todo items", "order by", "todo_items.sequence"



  ## Define relationship between people and to-do items
  tblrelation_createObject "people to todo items"
  tblrelation_set "people to todo items", "type", "many-to-many"

  tblrelation_set "people to todo items", "left table", "people"
  tblrelation_set "people to todo items", "left table key", "id"

  tblrelation_set "people to todo items", "cross-reference table", "people_todo"
  tblrelation_set "people to todo items", "cross-reference table key for left table", "people_i
  tblrelation_set "people to todo items", "cross-reference table key for right table", "todo_id

  tblrelation_set "people to todo items", "right table", "todo_items"
  tblrelation_set "people to todo items", "right table key", "id"

  tblrelation_set "people to todo items", "order by", "todo_items.name"

end _CreateRelationships
```

A Relationship object provides SQL Yoga the information it needs to dynamically generate SQL queries based off of the relationships. For example, the SQL Yoga command **sqlrecord_link** can automatically link records in two tables for you. **sqlrecord_getRelated** will automatically retrieve records from a table related to a SQL Record object's table. SQL Query objects use relationships to automatically generate JOIN queries if needed. It's all automated and it's all available to you by defining Relationship objects.

Defining a Relationship only requires that you tell SQL Yoga the type of relationship that exists between the two tables as well as which fields in each table are used to link the two tables together. As you can see in this code you can also specify which field is used to sort the related records that are returned (1).

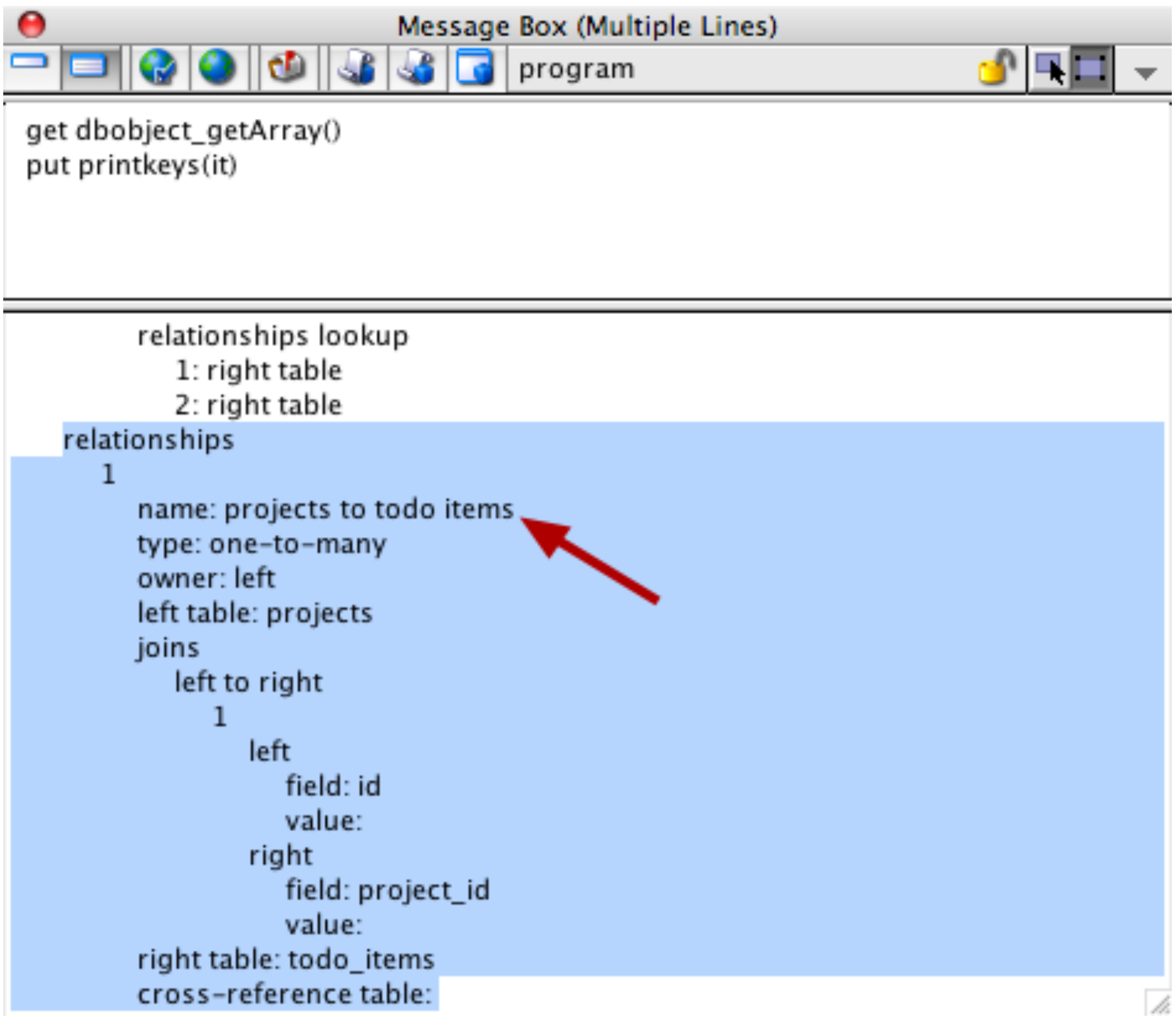
Trigger the dbobject.createTables.Default Message



```
tableobjects_reload
dbobject_save
```

Now that we have finished writing the dbobject.createTables.Default message we need to trigger it. Whenever you make an update to this handler (i.e. you add/remove a Table or other object) you can call **tableobjects_reload**. This deletes any existing objects in your Database object and calls the handler.

Confirm Your Work



```
get dbobject_getArray()
put printkeys(it)
```

```
            relationships lookup
                1: right table
                2: right table
        relationships
            1
                name: projects to todo items
                type: one-to-many
                owner: left
                left table: projects
                joins
                    left to right
                        1
                            left
                                field: id
                                value:
                            right
                                field: project_id
                                value:
                right table: todo_items
                cross-reference table:
```

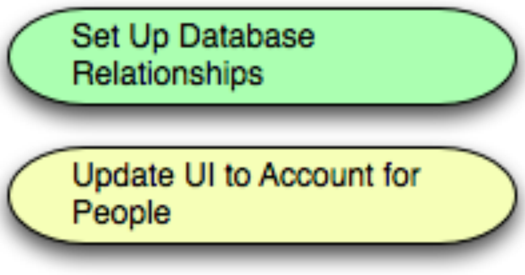
If you want to confirm that dbobject.createTables was called and did what it was supposed to then you can print out the array representation of the Database object in the message box. Do this by executing:

```
get dbobject_getArray()
put printkeys(it)
```

If you search through the printout you should see an entry for **relationships** along with the relationships you just created. For example, **projects to todo items**.

Update the UI Code to Account For People

Now that the left column displays both projects and people we will need to update the **uiPopulateToDos** handler accordingly. We will look at how the newly defined relationships can help us get to-do items associated with a person.


Set Up Database Relationships
Update UI to Account for People

Update the uiPopulateToDos Command

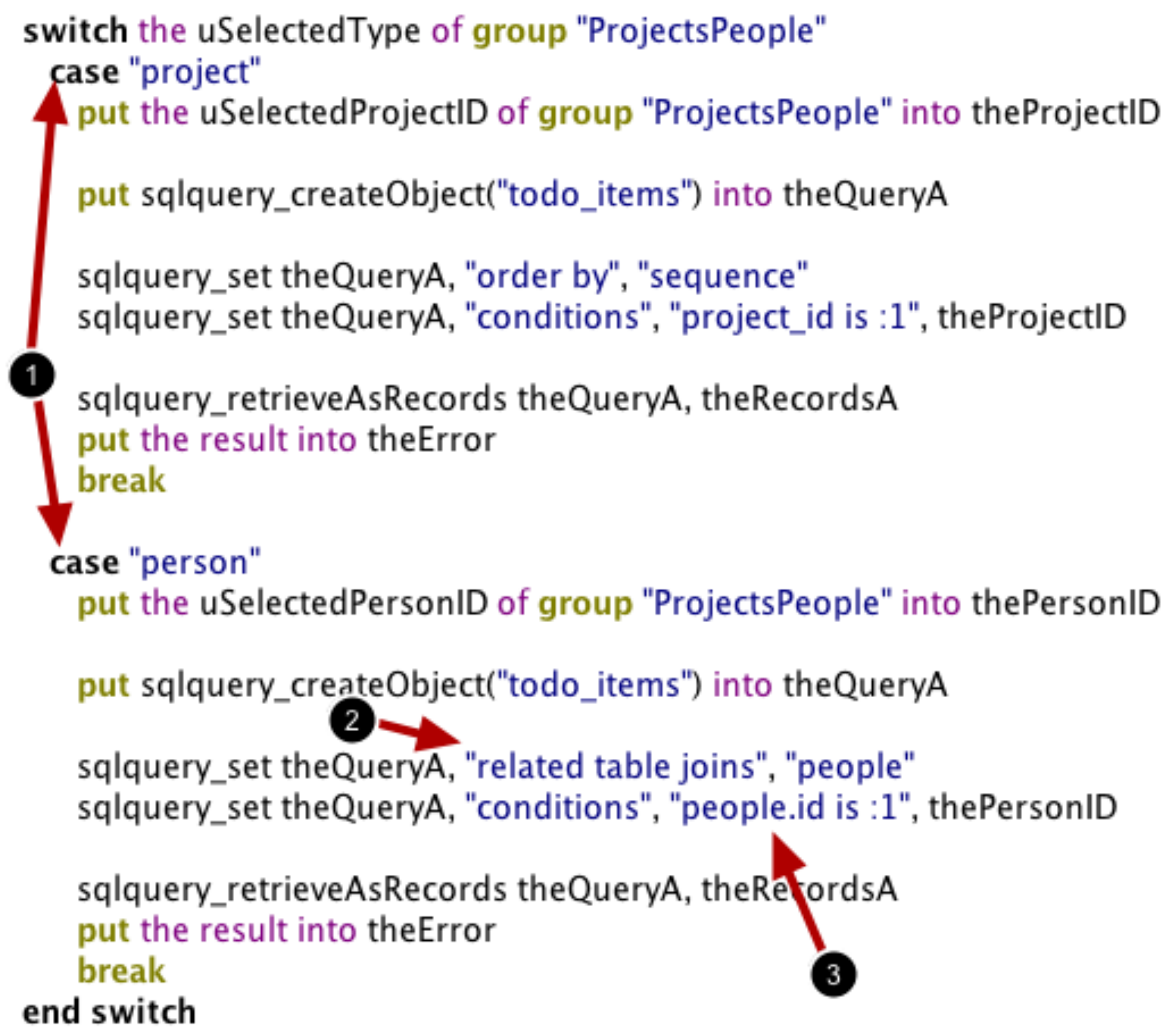
Replace the uiPopulateToDos command in the card script with the following RevTalk code. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command uiPopulateToDos
   local theRecordsA,theError

   switch the uSelectedType of group "ProjectsPeople"
      case "project"
         put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

         put sqlquery_createObject("todo_items") into theQueryA

         sqlquery_set theQueryA, "order by", "sequence"
         sqlquery_set theQueryA, "conditions", "project_id is :1", theProjectID

         sqlquery_retrieveAsRecords theQueryA, theRecordsA
         put the result into theError
         break

      case "person"
```

```
        put the uSelectedPersonID of group "ProjectsPeople" into thePersonID

        put sqlquery_createObject("todo_items") into theQueryA

        sqlquery_set theQueryA, "related table joins", "people"
        sqlquery_set theQueryA, "conditions", "people.id is :1", thePersonID

        sqlquery_retrieveAsRecords theQueryA, theRecordsA
        put the result into theError
        break
  end switch

  if theError is empty then
     set the dgData of group "ToDo" to theRecordsA
  end if

  if theError is not empty then
     answer "Error populating to-do items:" && theError & "."
  end if
end uiPopulateToDos
```

Getting To-Do Items Related To Selected Person

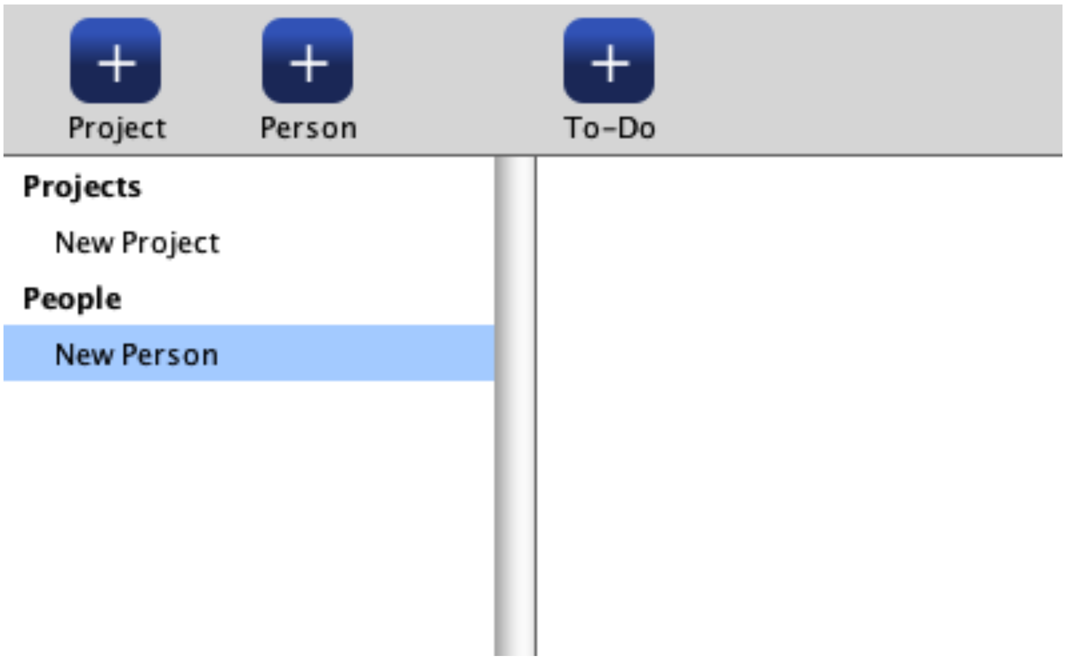


```
switch the uSelectedType of group "ProjectsPeople"
 case "project"
   put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

   put sqlquery_createObject("todo_items") into theQueryA

   sqlquery_set theQueryA, "order by", "sequence"
   sqlquery_set theQueryA, "conditions", "project_id is :1", theProjectID

   sqlquery_retrieveAsRecords theQueryA, theRecordsA
   put the result into theError
   break

 case "person"
   put the uSelectedPersonID of group "ProjectsPeople" into thePersonID

   put sqlquery_createObject("todo_items") into theQueryA

   sqlquery_set theQueryA, "related table joins", "people"
   sqlquery_set theQueryA, "conditions", "people.id is :1", thePersonID

   sqlquery_retrieveAsRecords theQueryA, theRecordsA
   put the result into theError
   break
end switch
```

You are looking at the modified code for uiPopulateToDos. Notice that the code now branches based on whether a person or project is selected (1).

What is interesting about the new code for retrieving to-do items is the use of the **related table joins** property (2). Remember that we told SQL Yoga that 'people' are related to 'todo_items'. By telling the SQL Query object that the query should include 'people' SQL Yoga will automatically add the necessary SQL to include the 'people' table in the query. This allows us to use 'people.id' in the 'conditions' property (3) to filter which to-do items are returned. Basically the query searches for all to-do items that are associated with the selected person.

Test

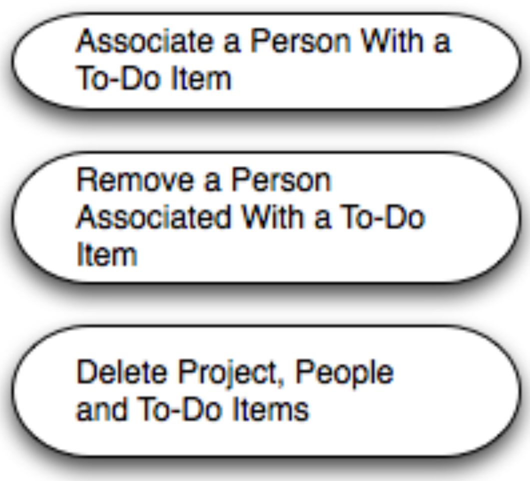


Click on **New Person** in the left column. You should see an empty list on the right. If you select **New Project** on the left then the **New Task** record should appear in the list.

Linking and Deleting Records

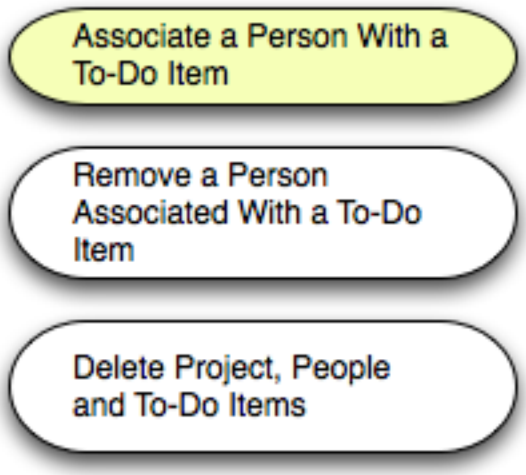
Chapter Overview

Here are the major subjects we are going to cover in this chapter.

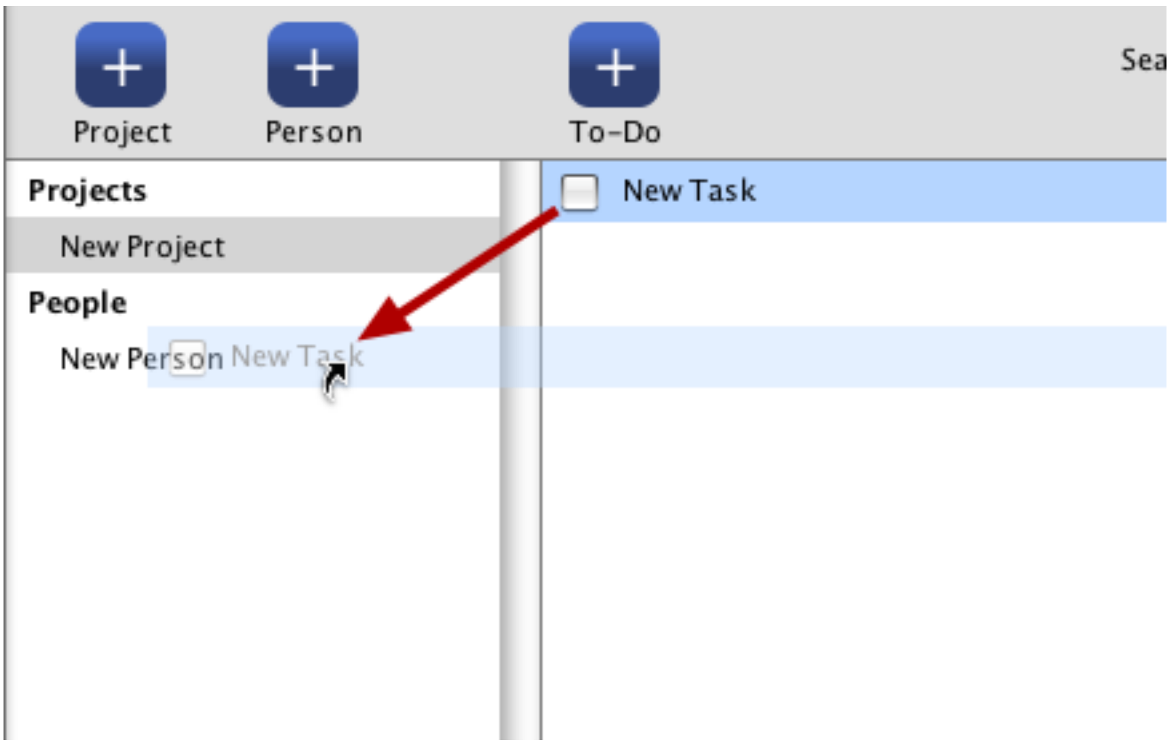

Associate a Person With a To-Do Item
Remove a Person Associated With a To-Do Item
Delete Project, People and To-Do Items

Associating a Person With a To-Do Item

In the To-Do application a person can be associated with a to-do item. The **people** table and **todo_items** table have a many-to-many relationship through the **people_todo** table. Let's look at how easy it is to connect these records using SQL Yoga.


Associate a Person With a To-Do Item
Remove a Person Associated With a To-Do Item
Delete Project, People and To-Do Items

What We Are Going To Do



To associate a task with a person we want to drag the task from the list onto the person's name in the left column. The handlers dealing with drag and drop have already been defined so we just need to add the RevTalk that performs the actual link.

Note: the dimmed image of the to-do item that is being dragged may not appear in the Revolution IDE. It relies on a relatively new Data Grid feature that will be included with Revolution 4.0.

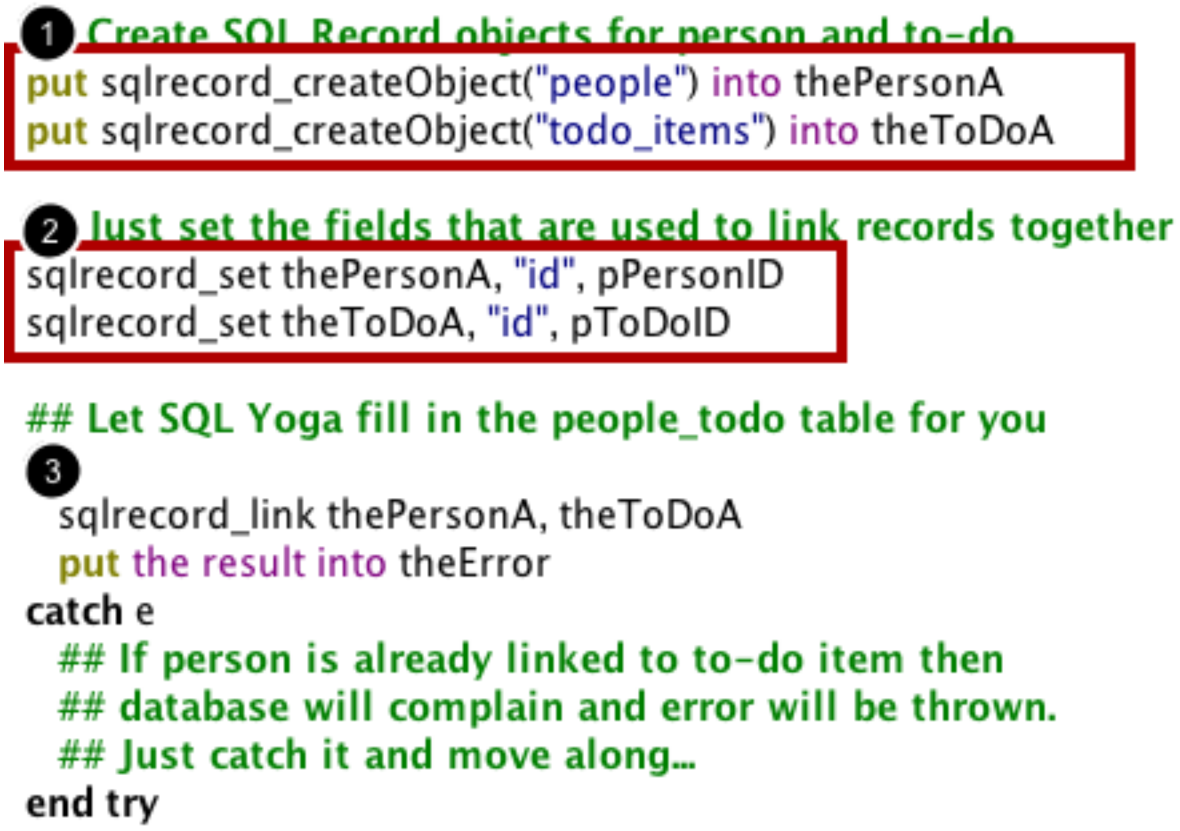
Update the LinkPersonWithToDo Command

Replace the LinkPersonWithToDo command in the card script with the following RevTalk code. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command LinkPersonWithToDo pPersonID, pToDoID
   ## Create SQL Record objects for person and to-do
   put sqlrecord_createObject("people") into thePersonA
   put sqlrecord_createObject("todo_items") into theToDoA

   ## Just set the fields that are used to link records together
   sqlrecord_set thePersonA, "id", pPersonID
   sqlrecord_set theToDoA, "id", pToDoID

   ## Let SQL Yoga fill in the people_todo table for you
   try
      sqlrecord_link thePersonA, theToDoA
      put the result into theError
   catch e
      ## If person is already linked to to-do item then
      ## database will complain and error will be thrown.
      ## Just catch it and move along...
   end try

   if theError is empty then
      ## Refresh list
      RefreshToDoList
   end if

   if theError is not empty then
      answer "Error linking person to to-do item:" && theError & "."
   end if
end LinkPersonWithToDo
```

Linking Two Related Database Records Together


```
1 Create SQL Record objects for person and to-do
put sqlrecord_createObject("people") into thePersonA
put sqlrecord_createObject("todo_items") into theToDoA

2 Just set the fields that are used to link records together
sqlrecord_set thePersonA, "id", pPersonID
sqlrecord_set theToDoA, "id", pToDoID

## Let SQL Yoga fill in the people_todo table for you
3
  sqlrecord_link thePersonA, theToDoA
  put the result into theError
catch e
  ## If person is already linked to to-do item then
  ## database will complain and error will be thrown.
  ## Just catch it and move along...
end try
```

Open the card script and find the empty LinkPersonWithToDo handler. To link two records together you can use SQL Record objects and the **sqlrecord_link** command. Here is how it works:

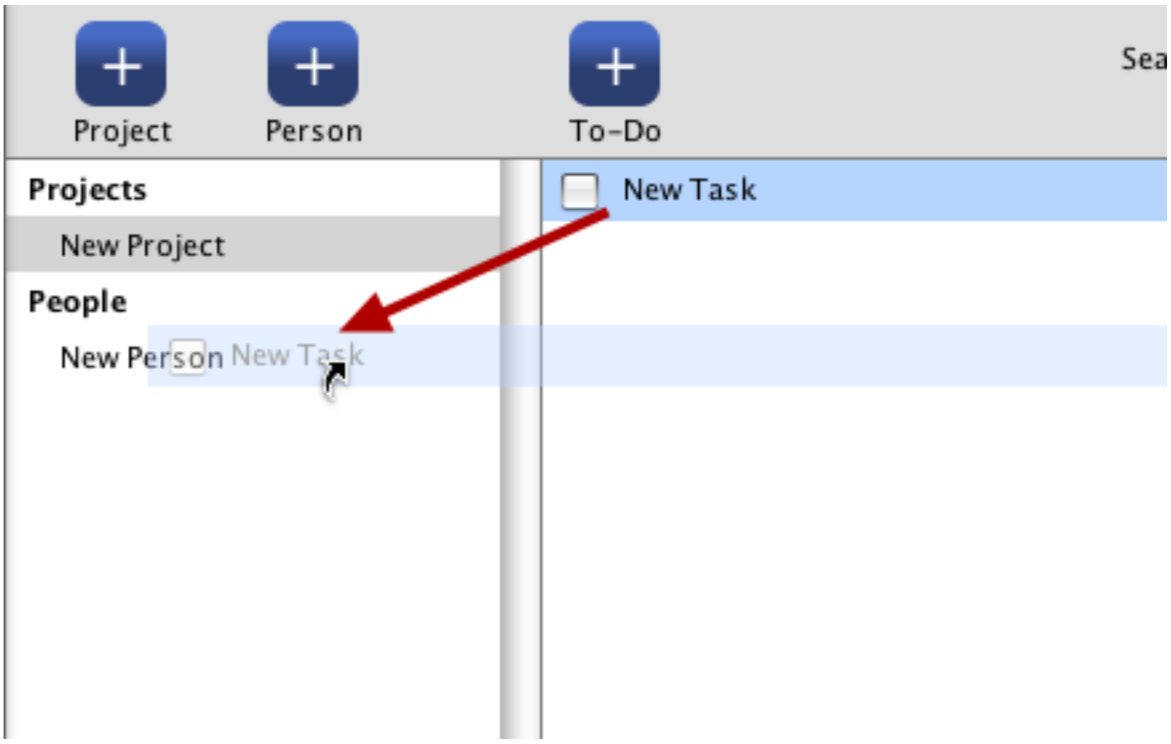
1) Create SQL Record objects for the two tables containing the records that you want to link. In this example we are going to use sqlrecord_createObject and then manually assign the 'id' properties for each object. We could also use sqlrecord_find which would grab the entire record from the database.

2) Fill in the 'id' properties (only necessary if not using sqlrecord_find). The 'id' properties for projects and todo_items are the unique fields used in the relationship. That is the minimum amount of information that SQL Yoga needs to link the records together.

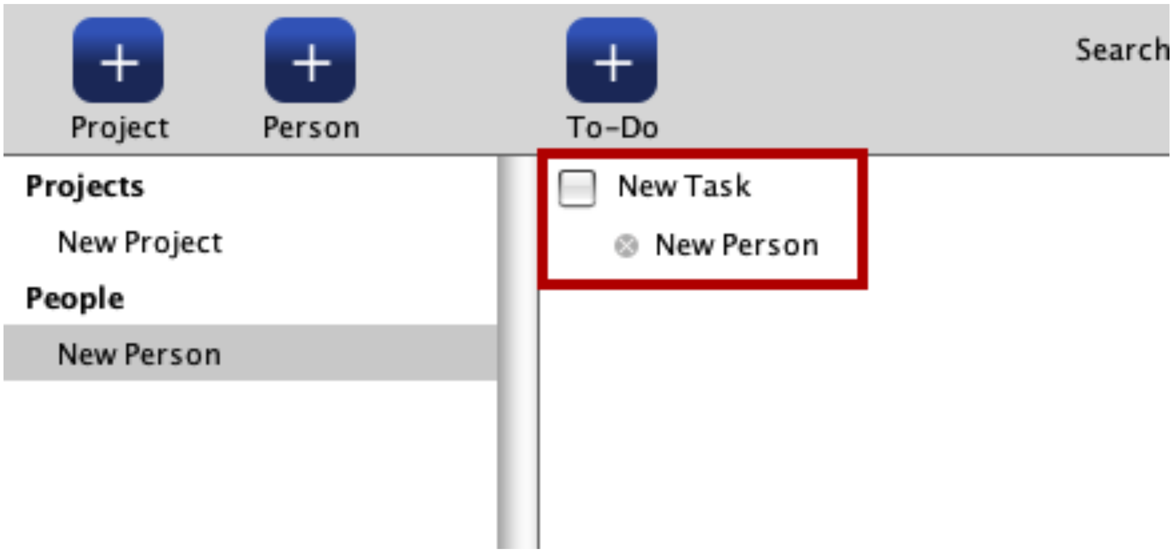
3) Call sqlrecord_link. SQL Yoga inserts the necessary values into the people_todo table.

Note that the call to sqlrecord_link is wrapped in a try/catch block. SQL Yoga throws an error if a SQL Query fails. The error is prefixed with **sqlyoga_executesql_err,** and includes the error message returned from the database after the comma. The To-Do application database does not allow a person to be linked to the same to-do item twice so the database will return an error if you try to do so. The try/catch block allows you to silently ignore this error and is adequate for the purposes of this sample application.

Test

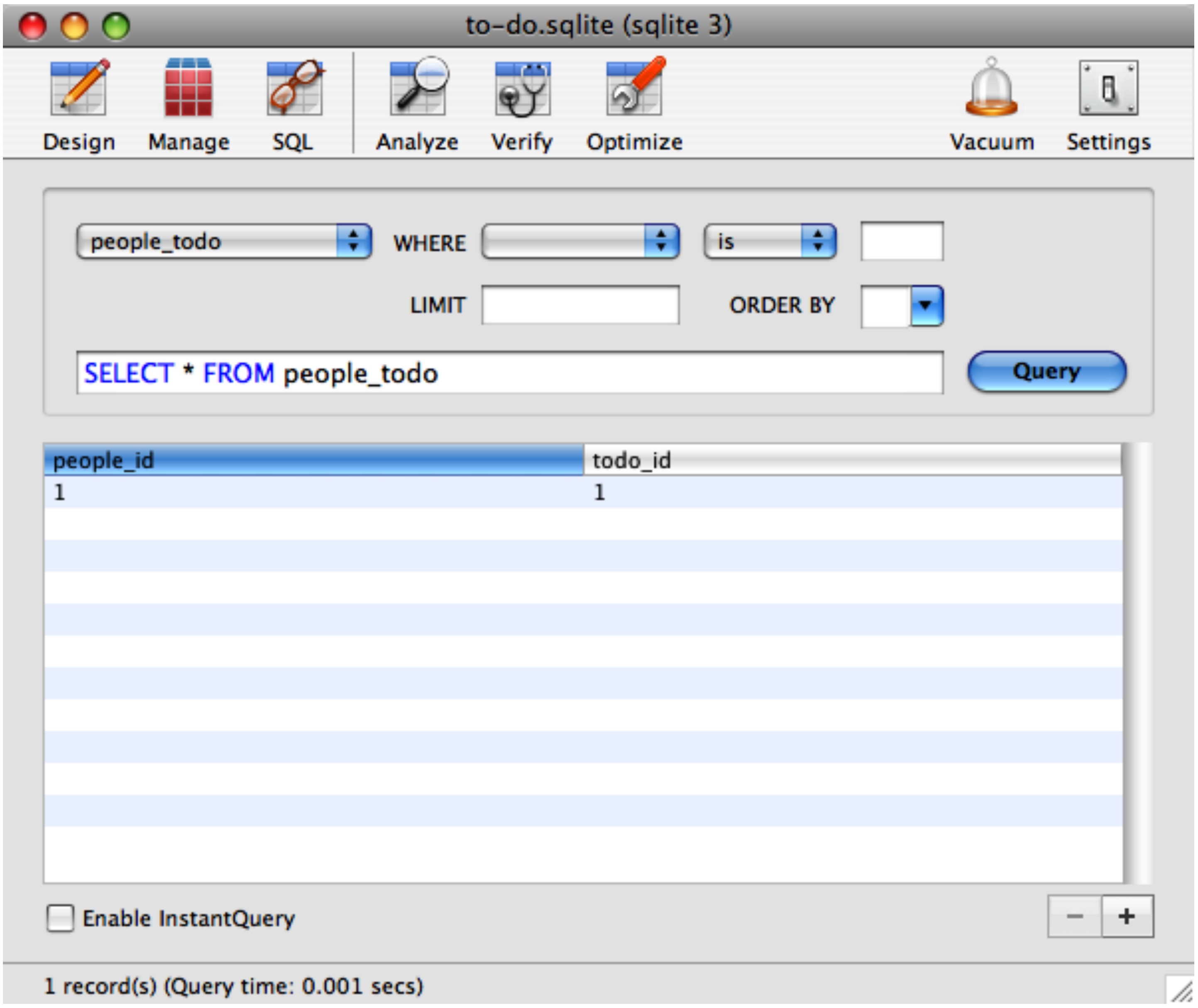


With **New Project** selected in the left column drag **New Task** onto **New Person** in the left column.



Clicking on **New Person** in the left column will now show the person associated with the task.

Verify That Database Record Was Created



```
SELECT * FROM people_todo
```

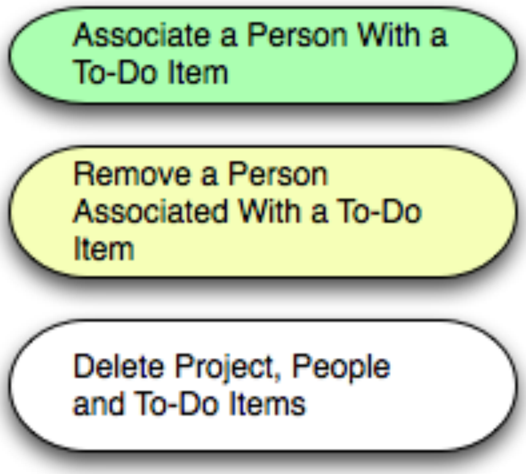
people_id	todo_id
1	1

1 record(s) (Query time: 0.001 secs)

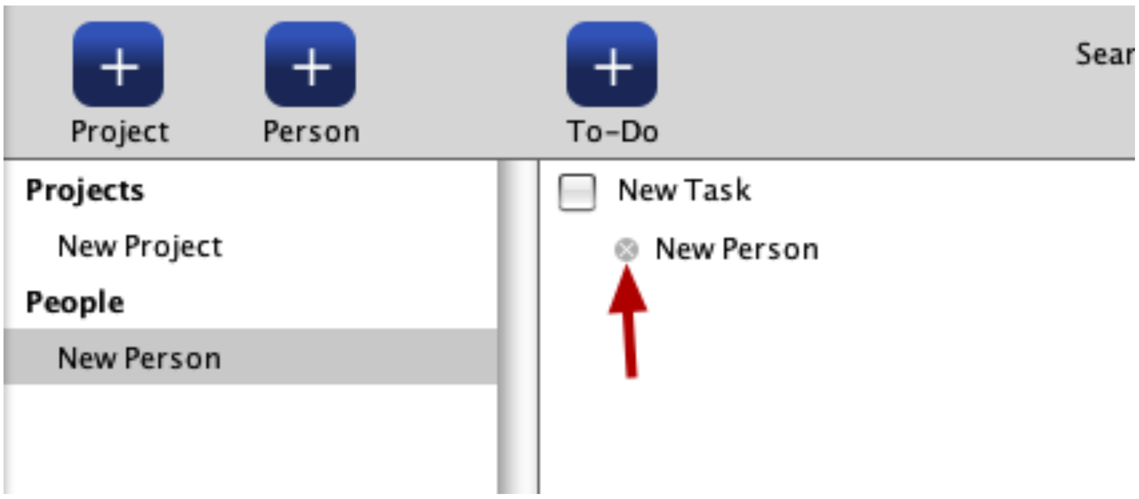
Look at the records in the **people_todo** table and you should see a new entry linking the task and the person.

Removing a Person Associated With a To-Do Item

We are now going to look at how to remove an association between a person and a to-do item.


Associate a Person With a To-Do Item
Remove a Person Associated With a To-Do Item
Delete Project, People and To-Do Items

Removing a Person from a To-Do Item



In the UI there is a little circle with an "x" in it that is used to remove a person from a task. Clicking this icon calls the uiRemovePersonFromToDo handler in the card script.

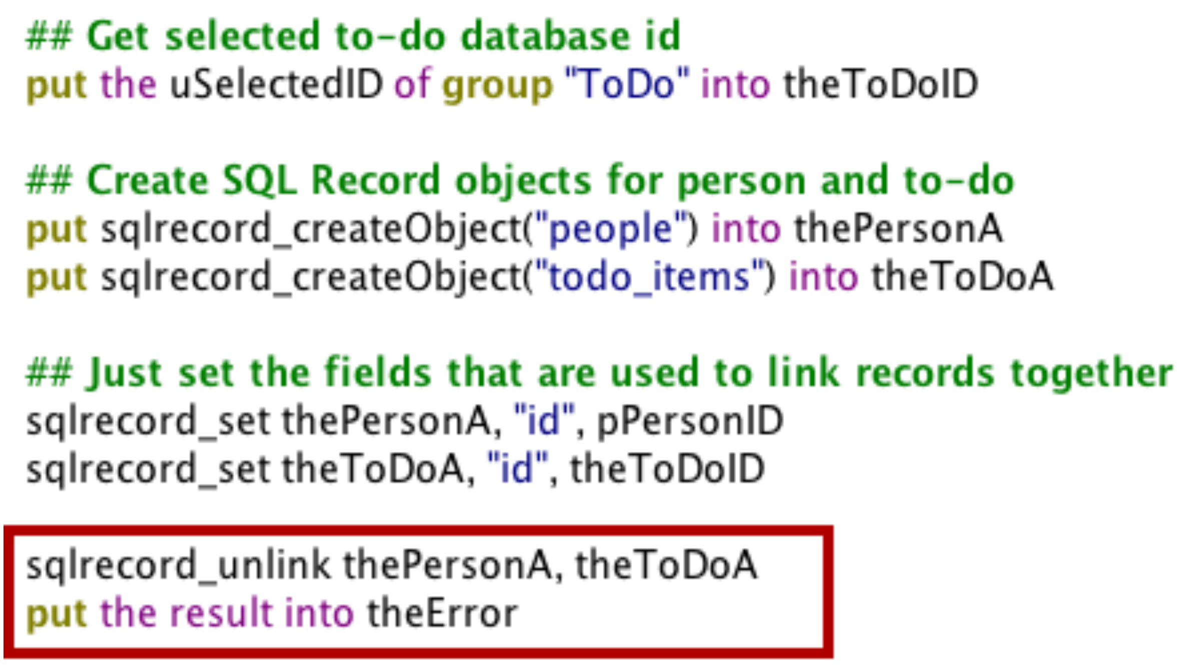
Update the uiRemovePersonFromToDo Command

Replace the empty uiRemovePersonFromTodo command in the card script with the following RevTalk code. After you insert the code we will go through the relevant parts.

Copy & Paste The Following Code

```
command uiRemovePersonFromTodo pPersonID
   ## Get selected to-do database id
   put the uSelectedID of group "ToDo" into theToDoID

   ## Create SQL Record objects for person and to-do
   put sqlrecord_createObject("people") into thePersonA
   put sqlrecord_createObject("todo_items") into theToDoA

   ## Just set the fields that are used to link records together
   sqlrecord_set thePersonA, "id", pPersonID
   sqlrecord_set theToDoA, "id", theToDoID

   sqlrecord_unlink thePersonA, theToDoA
   put the result into theError

   if theError is empty then
      ## Refresh list
      RefreshToDoList
   end if

   if theError is not empty then
      answer "Error removing person from to-do item:" && theError & "."
   end if
end uiRemovePersonFromTodo
```

Removing the Link Between Two Database Records

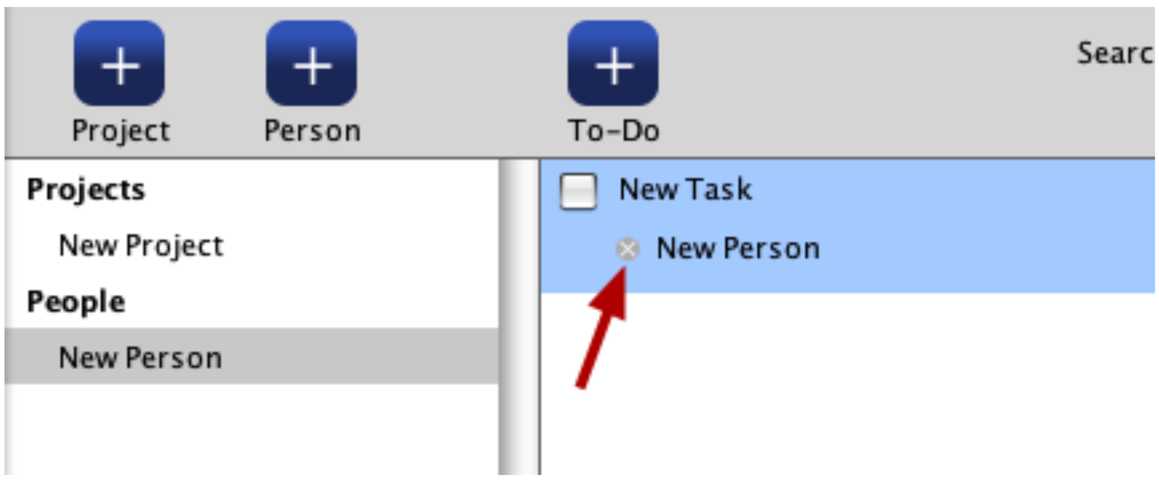


```
## Get selected to-do database id
put the uSelectedID of group "ToDo" into theToDoID

## Create SQL Record objects for person and to-do
put sqlrecord_createObject("people") into thePersonA
put sqlrecord_createObject("todo_items") into theToDoA

## Just set the fields that are used to link records together
sqlrecord_set thePersonA, "id", pPersonID
sqlrecord_set theToDoA, "id", theToDoID

sqlrecord_unlink thePersonA, theToDoA
put the result into theError
```

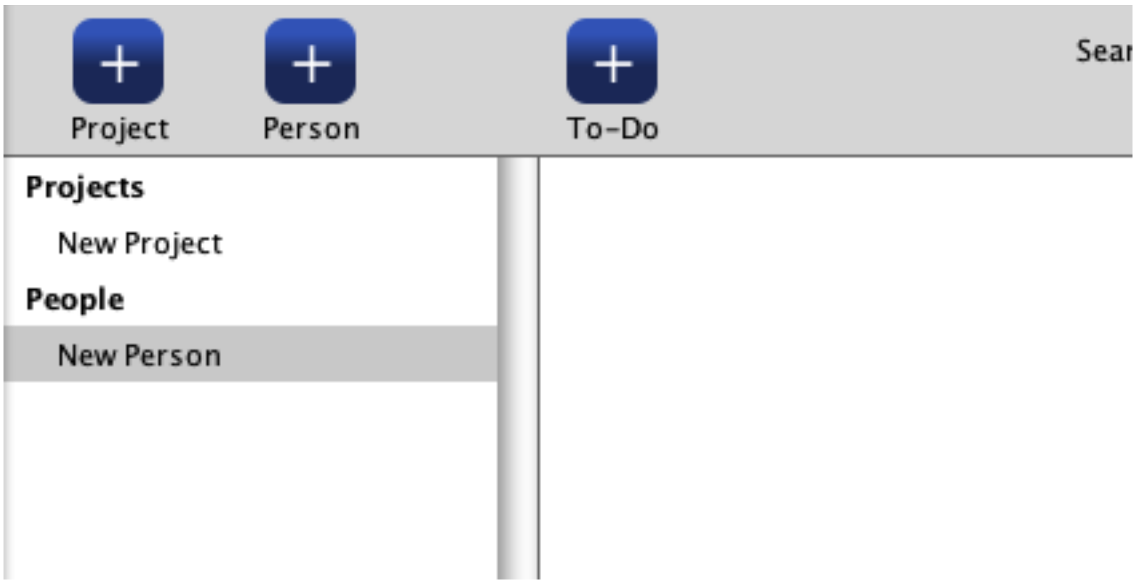
Removing the associated between two records is just as easy as associating two records. The only difference is that you call **sqlrecord_unlink** rather than **sqlrecord_link**.

Since the only errors that can be thrown when deleting an existing record are ones we are interested in knowing about there is no need to wrap the call in a try/catch block like we did for sqlrecord_link.

Test the UI

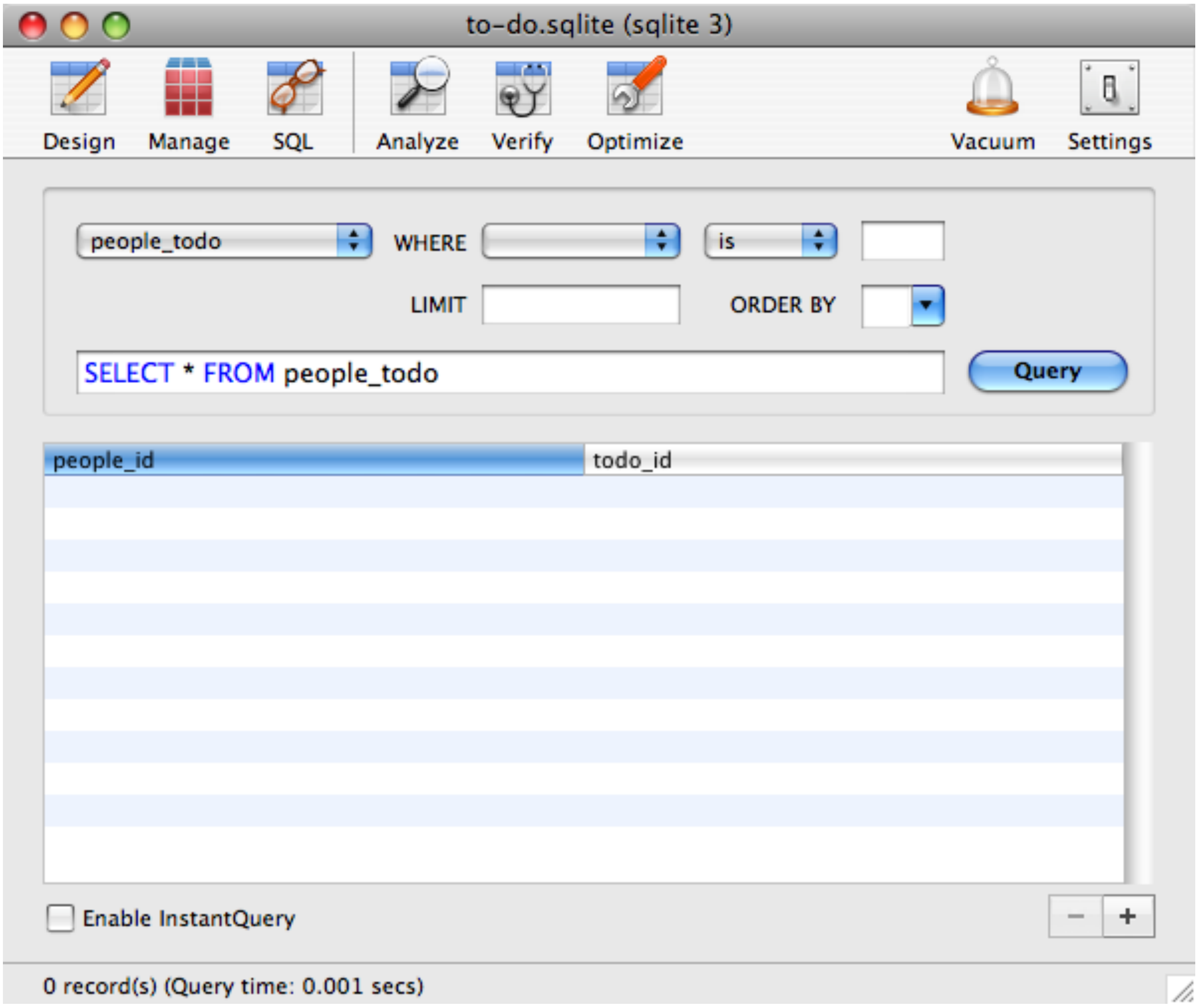


Click on the "x" next to **New Person** to trigger the uiRemovePersonFromTodo command.



The to-do list will refresh and you will see that the to-do item is no longer associated with **New Person**.

Verify in the Database



people_todo

```
SELECT * FROM people_todo
```

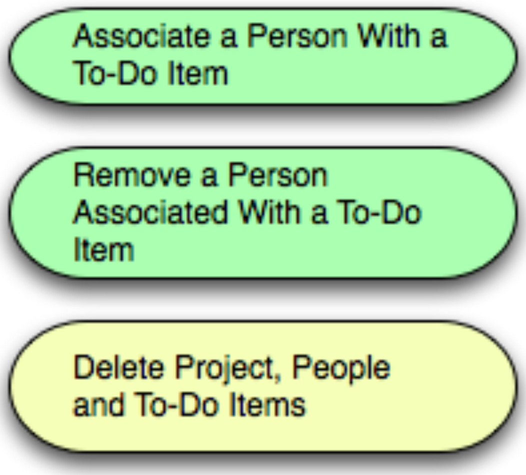
people_id	todo_id

0 record(s) (Query time: 0.001 secs)

If you look at the records in the people_todo table you will see that there are no longer any.

Deleting Projects, People and To-Do Items

Deleting records from a database using SQL Yoga is really easy. Let's quickly go through the code that deletes a project, person or to-do item.


Associate a Person With a To-Do Item
Remove a Person Associated With a To-Do Item
Delete Project, People and To-Do Items

Update the uiDeletePerson Command

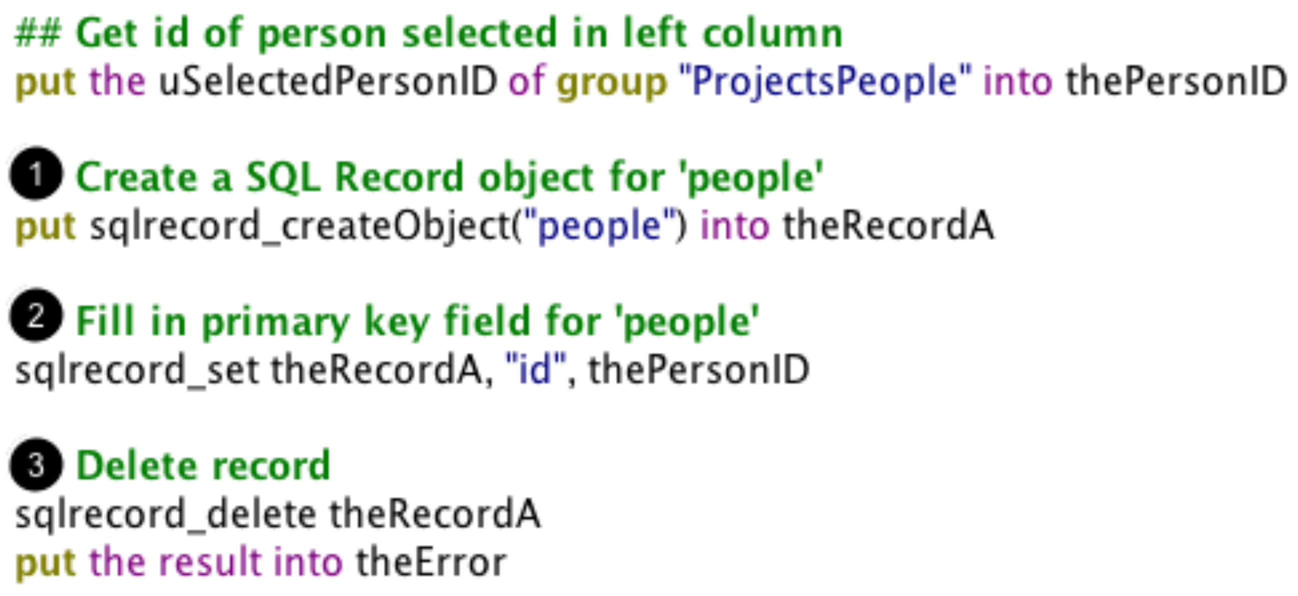
Replace the empty uiDeletePerson command in the card script with the following RevTalk code.

Copy & Paste The Following Code

```
command uiDeletePerson
   ## Get id of person selected in left column
   put the uSelectedPersonID of group "ProjectsPeople" into thePersonID

   ## Create a SQL Record object for 'people'
   put sqlrecord_createObject("people") into theRecordA

   ## Fill in primary key field for 'people'
   sqlrecord_set theRecordA, "id", thePersonID

   ## Delete record
   sqlrecord_delete theRecordA
   put the result into theError

   if theError is empty then
```

```
      ## Refresh list
      RefreshProjectsPeopleList
   end if

   if theError is not empty then
      answer "Error deleting person:" && theError & "."
   end if
end uiDeletePerson
```

Deleting a Person

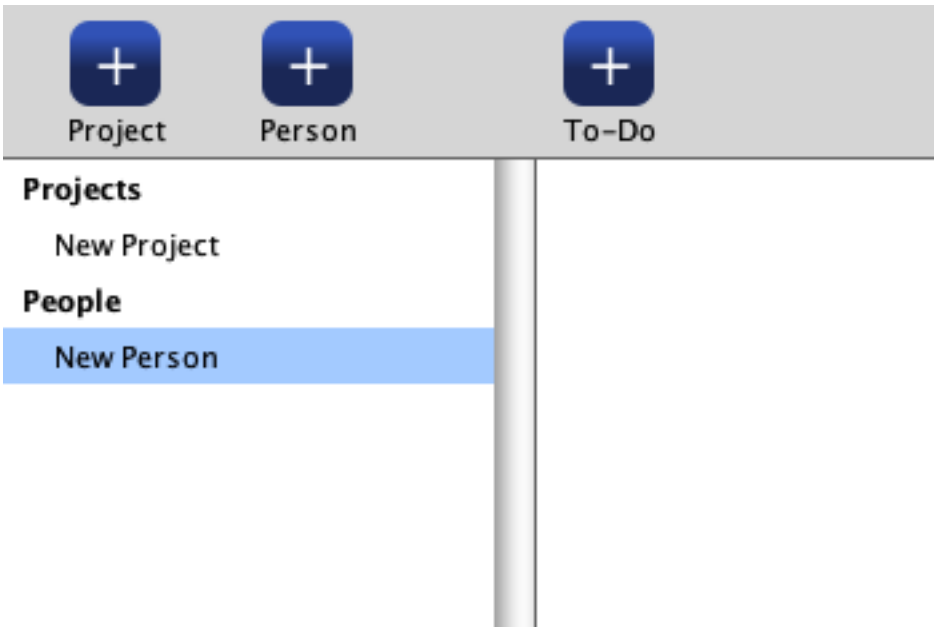


```
## Get id of person selected in left column
put the uSelectedPersonID of group "ProjectsPeople" into thePersonID

1 Create a SQL Record object for 'people'
put sqlrecord_createObject("people") into theRecordA

2 Fill in primary key field for 'people'
sqlrecord_set theRecordA, "id", thePersonID

3 Delete record
sqlrecord_delete theRecordA
put the result into theError
```

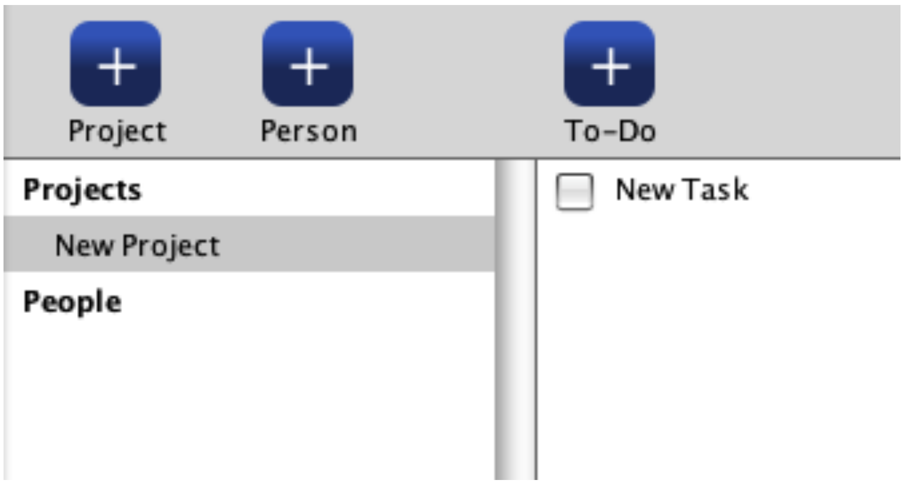
To delete a person from the database you can use a SQL Record object (1). You just need to fill in the primary key field for the table which is 'id' (2).

sqlrecord_delete will then generate the SQL to delete the record from the database (3).

Test



Select **New Person** in the UI and press the **delete** key or **backspace** key.



New Person will be removed.

Deleting Projects

Replace the empty uiDeleteProject command in the card script with the following RevTalk code. You can then use the delete or backspace key to delete a project.

Copy & Paste The Following Code
----

command uiDeleteProject

```
   ## Get id of project selected in left column
   put the uSelectedProjectID of group "ProjectsPeople" into theProjectID

   ## Create a SQL Record object for 'projects'
   put sqlrecord_createObject("projects") into theRecordA

   ## Fill in primary key field for 'projects'
   sqlrecord_set theRecordA, "id", theProjectID

   ## Delete record
   sqlrecord_delete theRecordA
   put the result into theError

   if theError is empty then
      ## Refresh list
      RefreshProjectsPeopleList
   end if

   if theError is not empty then
      answer "Error deleting project:" && theError & "."
   end if
end uiDeleteProject
```

Deleting To-Do Items

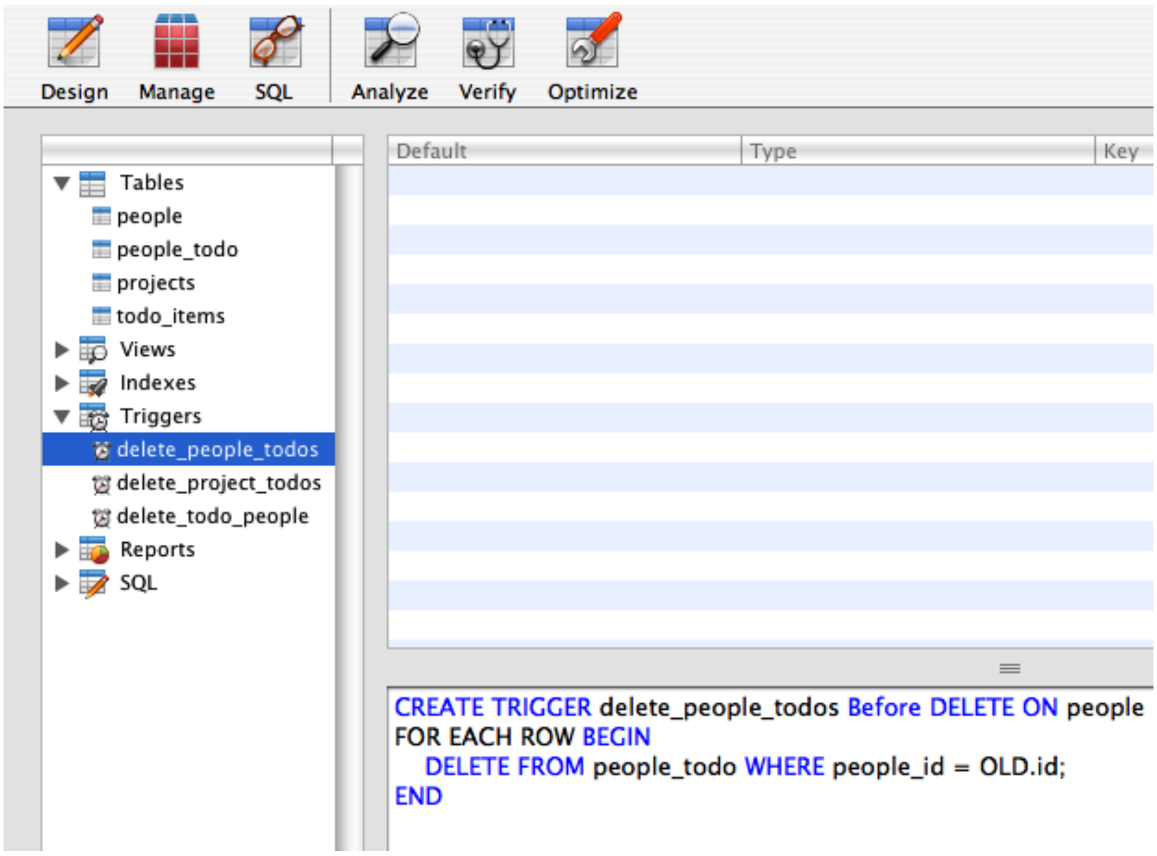
Replace the empty uiDeleteToDo command in the card script with the following RevTalk code. You can then use the delete or backspace key to delete a to-do item.

Copy & Paste The Following Code

```
command uiDeleteToDo
   ## Get id of project selected in left column
   put the uSelectedID of group "ToDo" into theToDoID

   ## Create a SQL Record object for 'projects'
   put sqlrecord_createObject("todo_items") into theRecordA

   ## Fill in primary key field for 'projects'
```

```
    sqlrecord_set theRecordA, "id", theToDoID

    ## Delete record
    sqlrecord_delete theRecordA
    put the result into theError

    if theError is empty then
        ## Refresh list
        RefreshToDoList
    end if

    if theError is not empty then
        answer "Error deleting to-do item:" && theError & "."
    end if
end uiDeleteToDo
```

A Short Note on Triggers and SQLite



```
CREATE TRIGGER delete_people_todos Before DELETE ON people
FOR EACH ROW BEGIN
    DELETE FROM people_todo WHERE people_id = OLD.id;
END
```

When relationships exist between two tables in a database you sometimes need to delete records in other tables when deleting a record from a related table. For example, if you delete a person from the database the link between that person and any to-do items needs to be removed and vice versa.

If you delete a project from the database you need to delete any related to-do items.

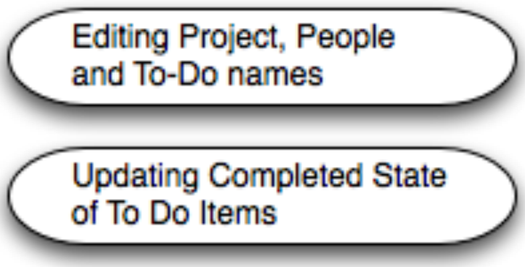
While some databases will take care of deleting the records for you, SQLite will not. This is where Triggers come into play. A Trigger is a way of performing operations in a database when certain events occur. If you look at the Triggers for the to-do application database you will see that there are three of them. This triggers delete to-dos when a project is deleted and remove the link between people/to-dos when either one is deleted.

You could mimic this logic in the application code but by incorporating the logic into the database you can edit records in a database manager and know that your data does not become corrupted (i.e. to-dos linked to projects that don't exist).

Updating Database Records

Chapter Overview

Here are the major subjects we are going to cover in this chapter.

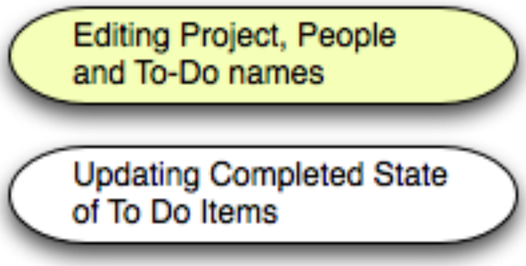


Editing Project, People and To-Do names

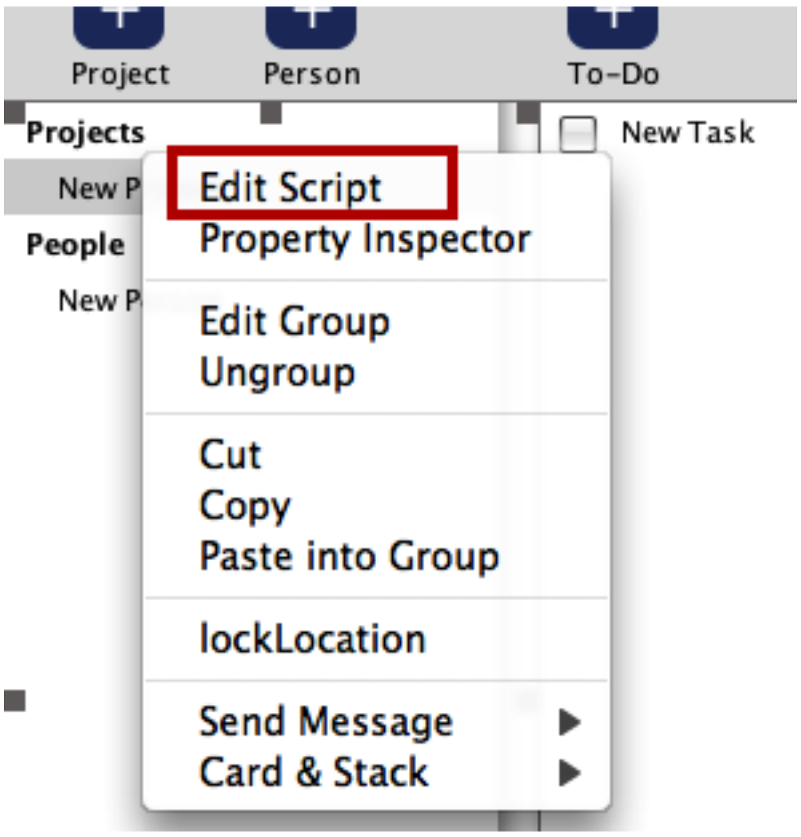
Updating Completed State of To Do Items

Editing a Project, Person or To-Do Name

So far we have looked at creating records in and retrieving and deleting records from the database. We haven't looked at how to update records yet, however, so let's take a look at that now.

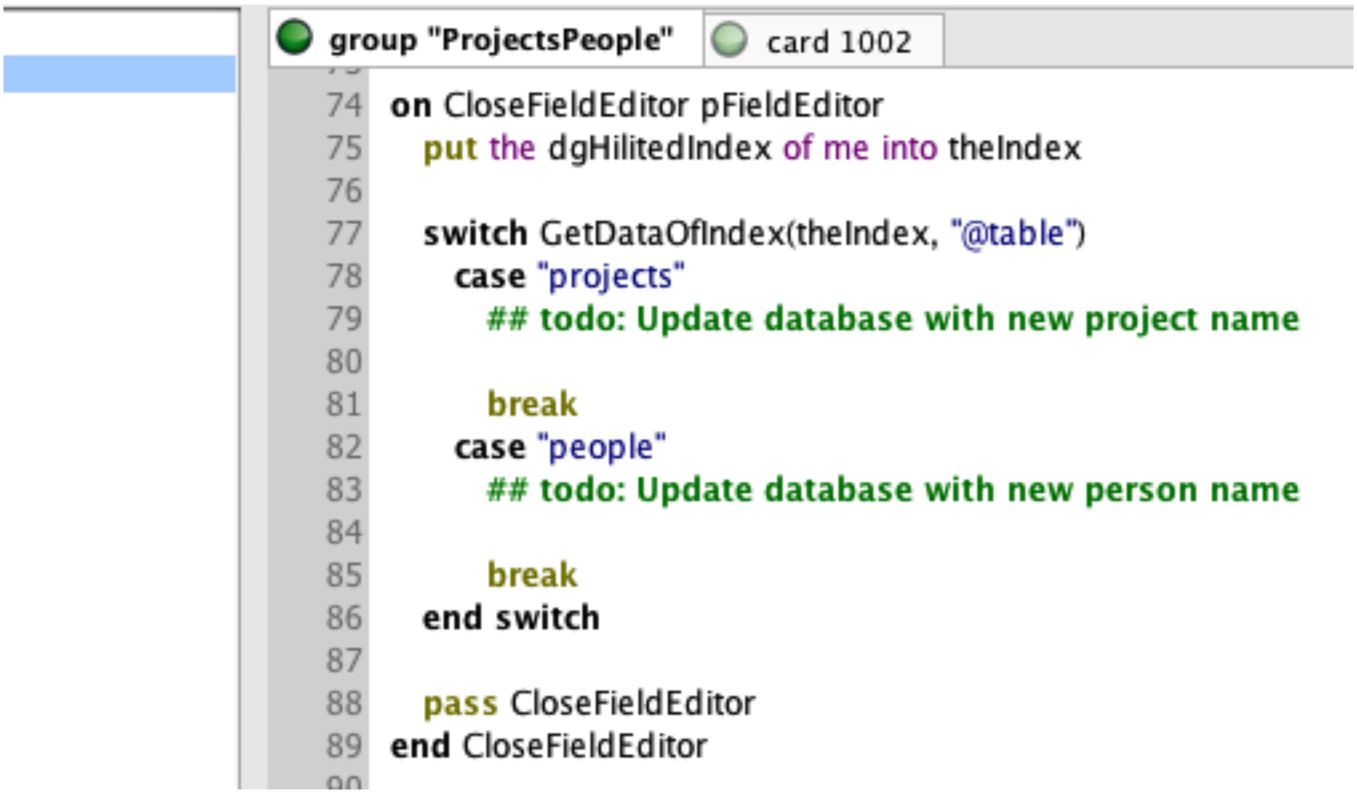

Editing Project, People and To-Do names
Updating Completed State of To Do Items

Edit ProjectsPeople Data Grid Script



Right-click on the left column Data Grid (named "ProjectsPeople") and choose **Edit Script**.

Locate the CloseFieldEditor Handler



```
on CloseFieldEditor pFieldEditor
   put the dgHilitedIndex of me into theIndex

   switch GetDataOfIndex(theIndex, "@table")
      case "projects"
         ## todo: Update database with new project name

         break
      case "people"
         ## todo: Update database with new person name

         break
   end switch

   pass CloseFieldEditor
end CloseFieldEditor
```

You should have a CloseFieldEditor handler in the script that looks like this. When the user double-clicks on a project or person name a field is opened that allows the user to edit the text. CloseFieldEditor is a message that is sent to the Data Grid when the user presses the enter or return key and the content has changed.

Update the CloseFieldEditor Script

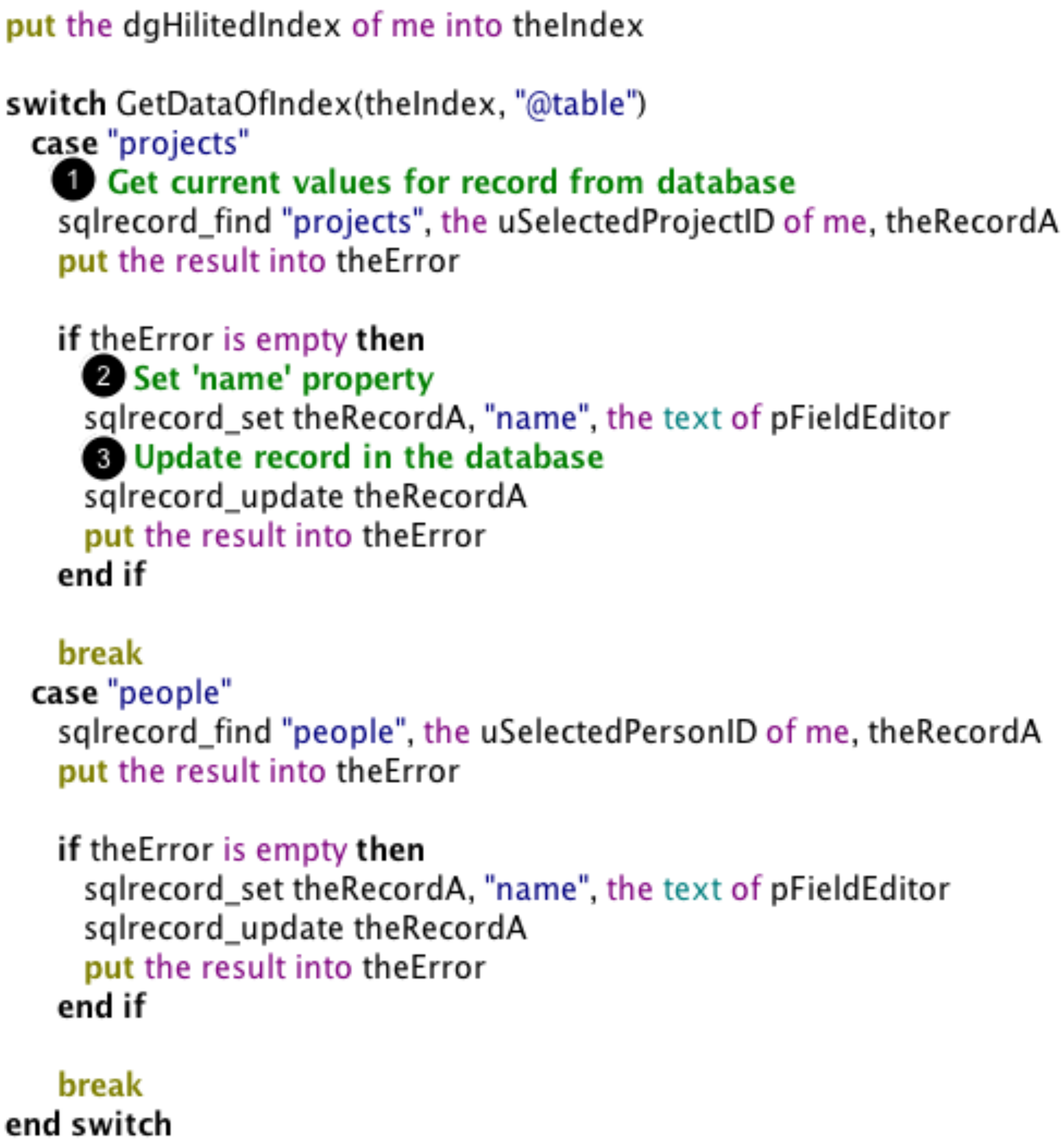
Replace the CloseFieldEditor command in the ProjectsPeople Data Grid group script with the following RevTalk code.

Copy & Paste The Following Code

```
on CloseFieldEditor pFieldEditor
   put the dgHilitedIndex of me into theIndex

   switch GetDataOfIndex(theIndex, "@table")
      case "projects"
         ## Get current values for record from database
         sqlrecord_find "projects", the uSelectedProjectID of me, theRecordA
```

```
        put the result into theError

        if theError is empty then
           ## Set 'name' property
           sqlrecord_set theRecordA, "name", the text of pFieldEditor
           ## Update record in the database
           sqlrecord_update theRecordA
           put the result into theError
        end if

        break
     case "people"
        sqlrecord_find "people", the uSelectedPersonID of me, theRecordA
        put the result into theError

        if theError is empty then
           sqlrecord_set theRecordA, "name", the text of pFieldEditor
           sqlrecord_update theRecordA
           put the result into theError
        end if

        break
  end switch

  if theError is empty then
     pass CloseFieldEditor
  else
     answer "Error saving name:" && theError & "."
  end if
end CloseFieldEditor
```

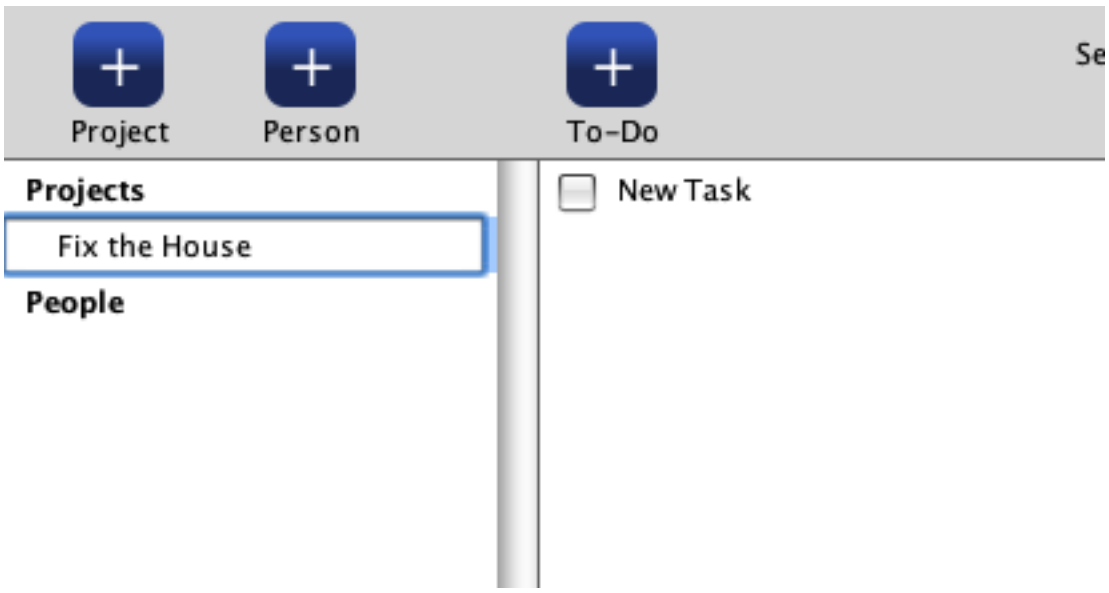
Updating Data in the Database



```
put the dgHilitedIndex of me into theIndex

switch GetDataOfIndex(theIndex, "@table")
  case "projects"
    1 Get current values for record from database
    sqlrecord_find "projects", the uSelectedProjectID of me, theRecordA
    put the result into theError

    if theError is empty then
      2 Set 'name' property
      sqlrecord_set theRecordA, "name", the text of pFieldEditor
      3 Update record in the database
      sqlrecord_update theRecordA
      put the result into theError
    end if

    break
  case "people"
    sqlrecord_find "people", the uSelectedPersonID of me, theRecordA
    put the result into theError

    if theError is empty then
      sqlrecord_set theRecordA, "name", the text of pFieldEditor
      sqlrecord_update theRecordA
      put the result into theError
    end if

    break
end switch
```

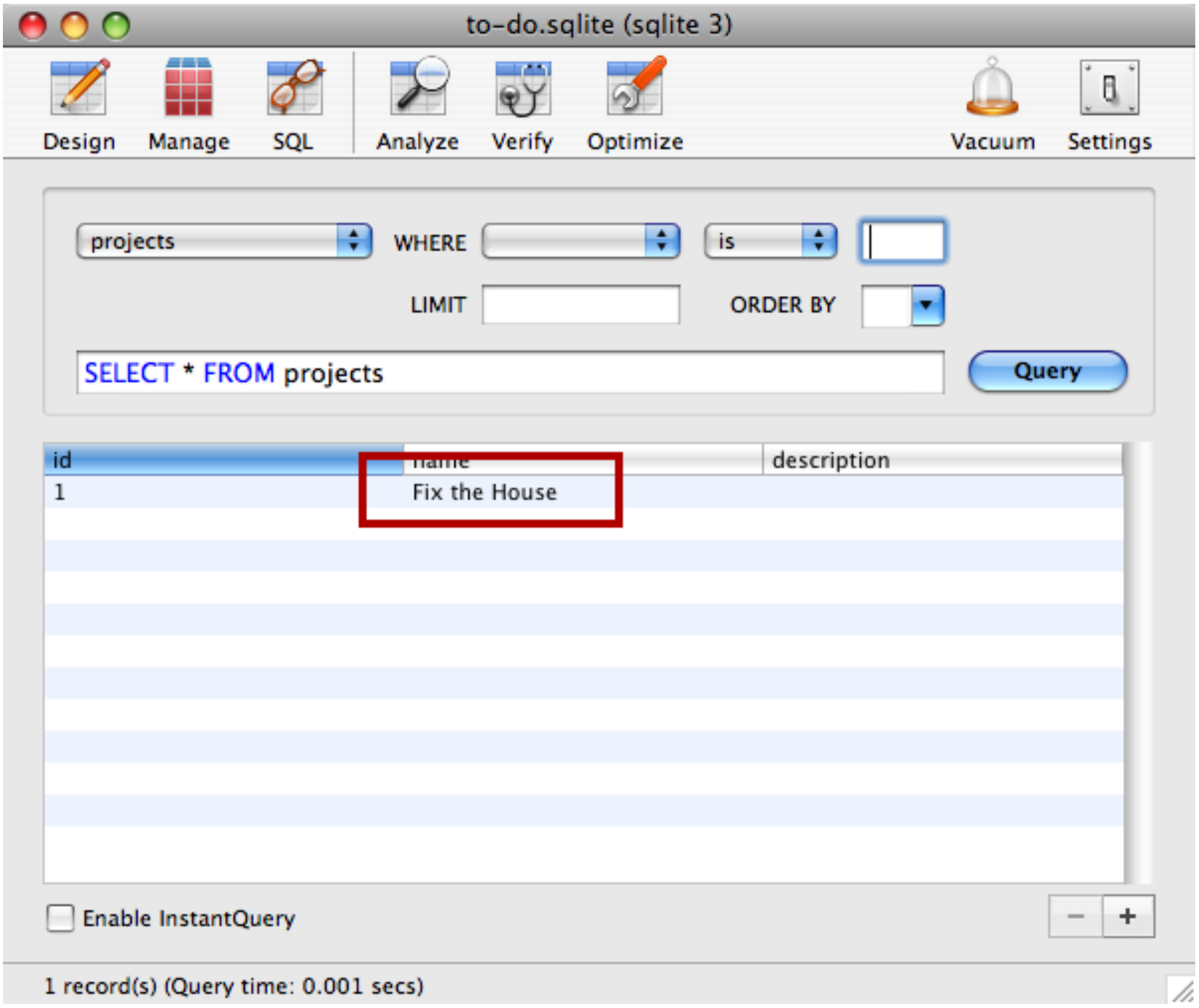
Here is what the completed handler will look like. This example introduces the **sqlrecord_find** command (1). This command searches a database table and returns a SQL Record object. This is how we get all of the fields from the database for a project or person before upating the name. After locating the record you can update the **name** (2) and call **sqlrecord_update** (3).

Test the UI



Double-click on **New Project** and edit the text. Press the **enter** or **return** key to save the changes.

Verify in the Database


to-do.sqlite (sqlite 3)

projects WHERE is

LIMIT ORDER BY

SELECT * FROM projects

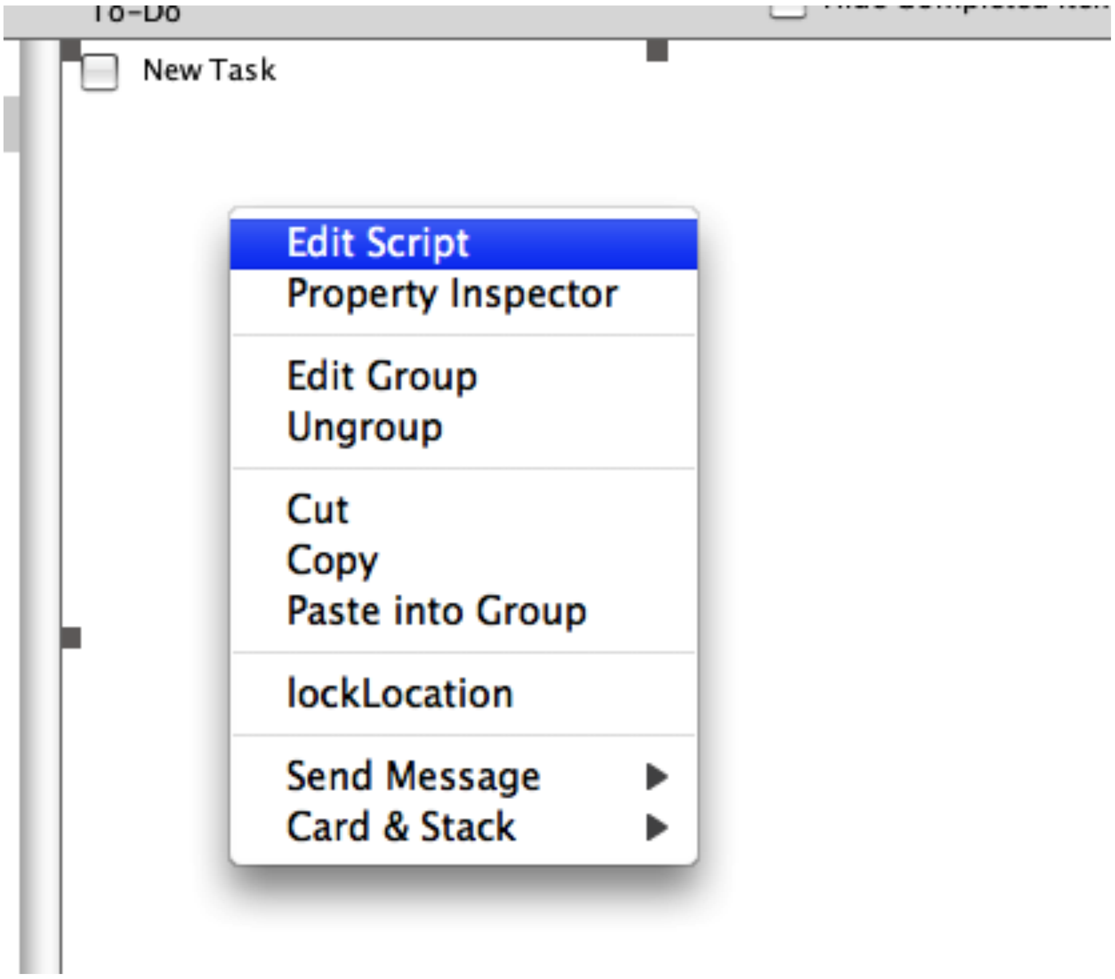
id	name	description
1	Fix the House	

Enable InstantQuery

1 record(s) (Query time: 0.001 secs)

If you look at the records in the **projects** table you will see that the name has been updated.

Edit the ToDo Data Grid Script



Now let's update the code in the ToDo Data Grid. Select the ToDo Data Grid and edit the script.

Update Data Grid Group Script

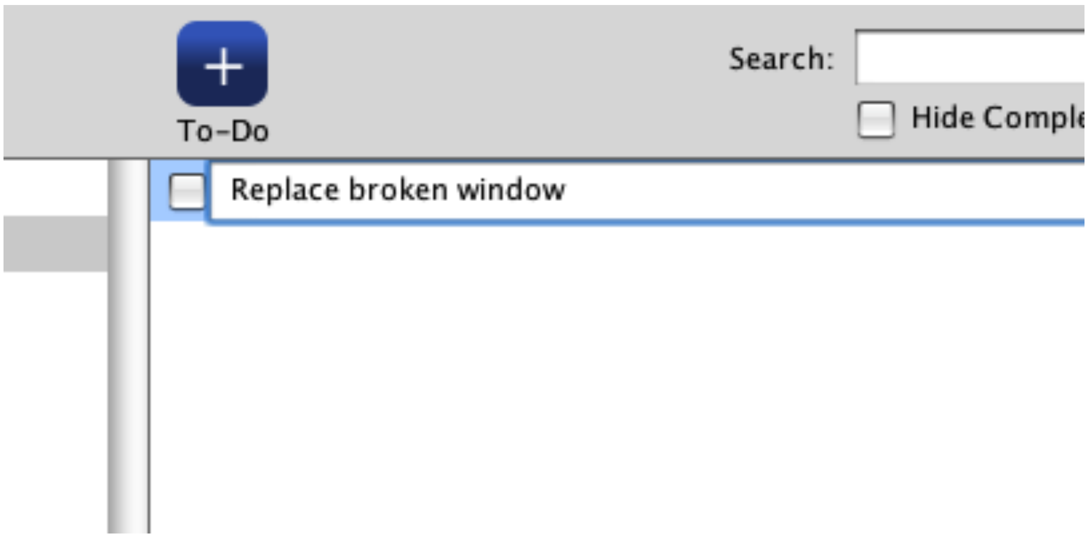
Replace the CloseFieldEditor command in the ToDo Data Grid group script with the following RevTalk.

Copy & Paste The Following Code

```
on CloseFieldEditor pFieldEditor
   ## Get current values for record from database
   sqlrecord_find "todo_items", the uSelectedID of me, theRecordA
   put the result into theError

   if theError is empty then
```

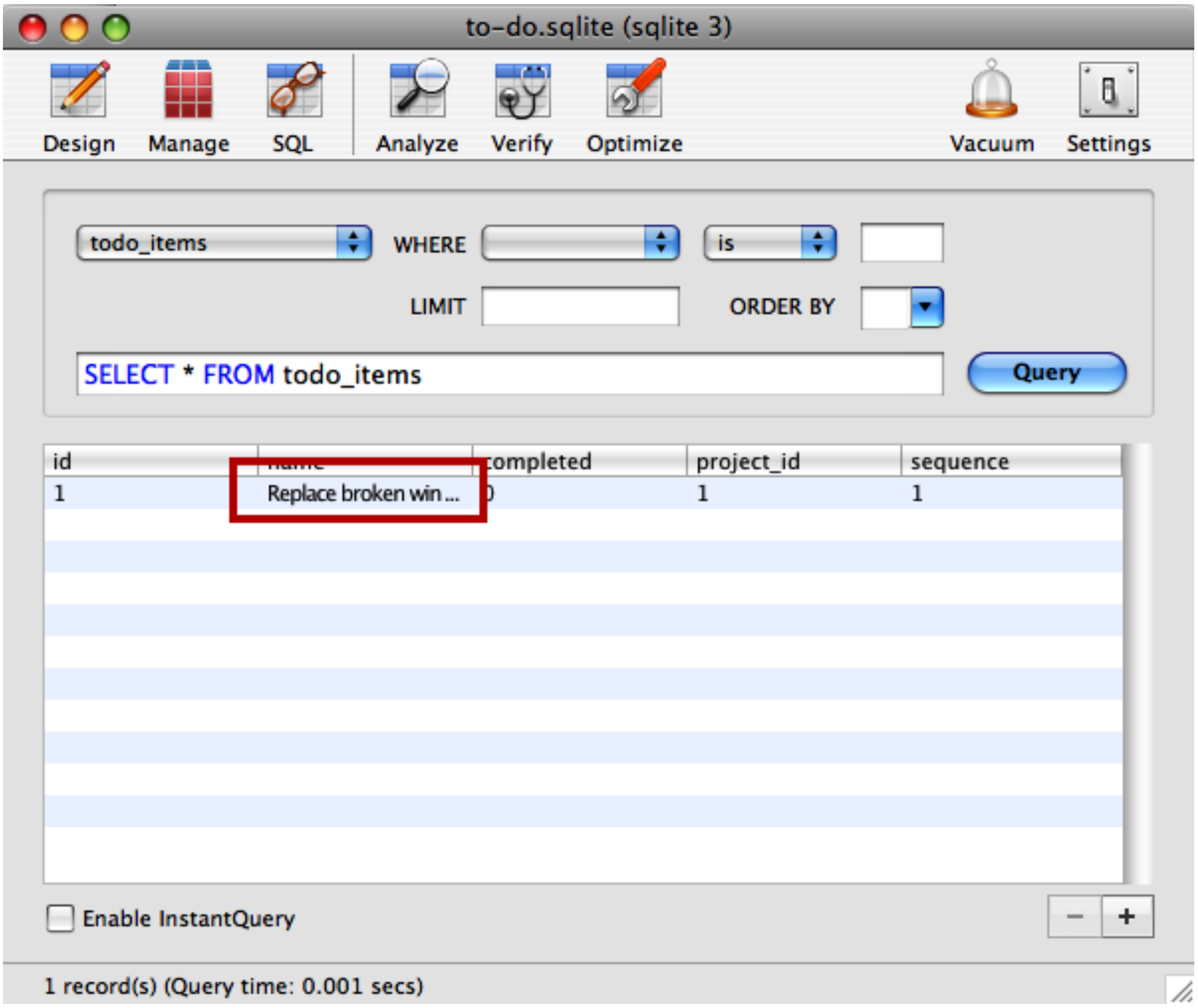
```
        ## Set 'name' property
        sqlrecord_set theRecordA, "name", the text of pFieldEditor
        ## Update record in the database
        sqlrecord_update theRecordA
        put the result into theError
    end if

    pass CloseFieldEditor
end CloseFieldEditor
```

Test the UI



Double-click on **New Task** and update the name. Press the **return** or **enter** key to save the changes.

Verify in the Database



todo_items

```
SELECT * FROM todo_items
```

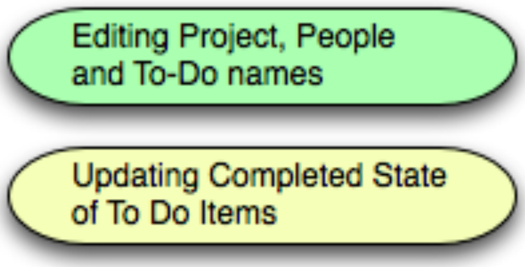
id	name	completed	project_id	sequence
1	Replace broken win...	0	1	1

1 record(s) (Query time: 0.001 secs)

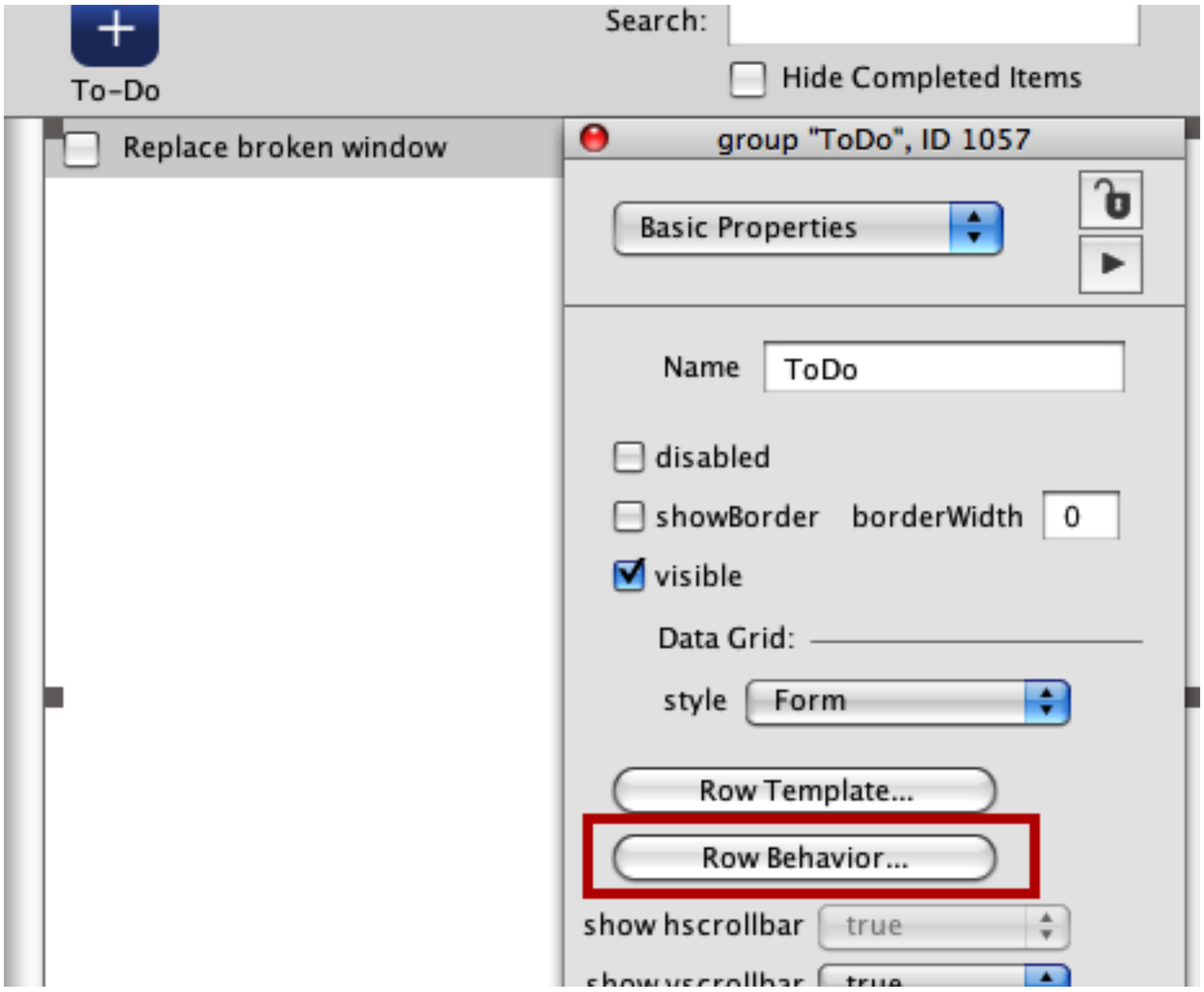
The to-do record in the databae will be updated.

Updating a To-Do Items Completed State

We are going to look at one more example of updating a record before moving on. This lesson will show how to update the **completed** property for a to-do item and explains how boolean values work with SQL Record objects.


Editing Project, People and To-Do names
Updating Completed State of To Do Items

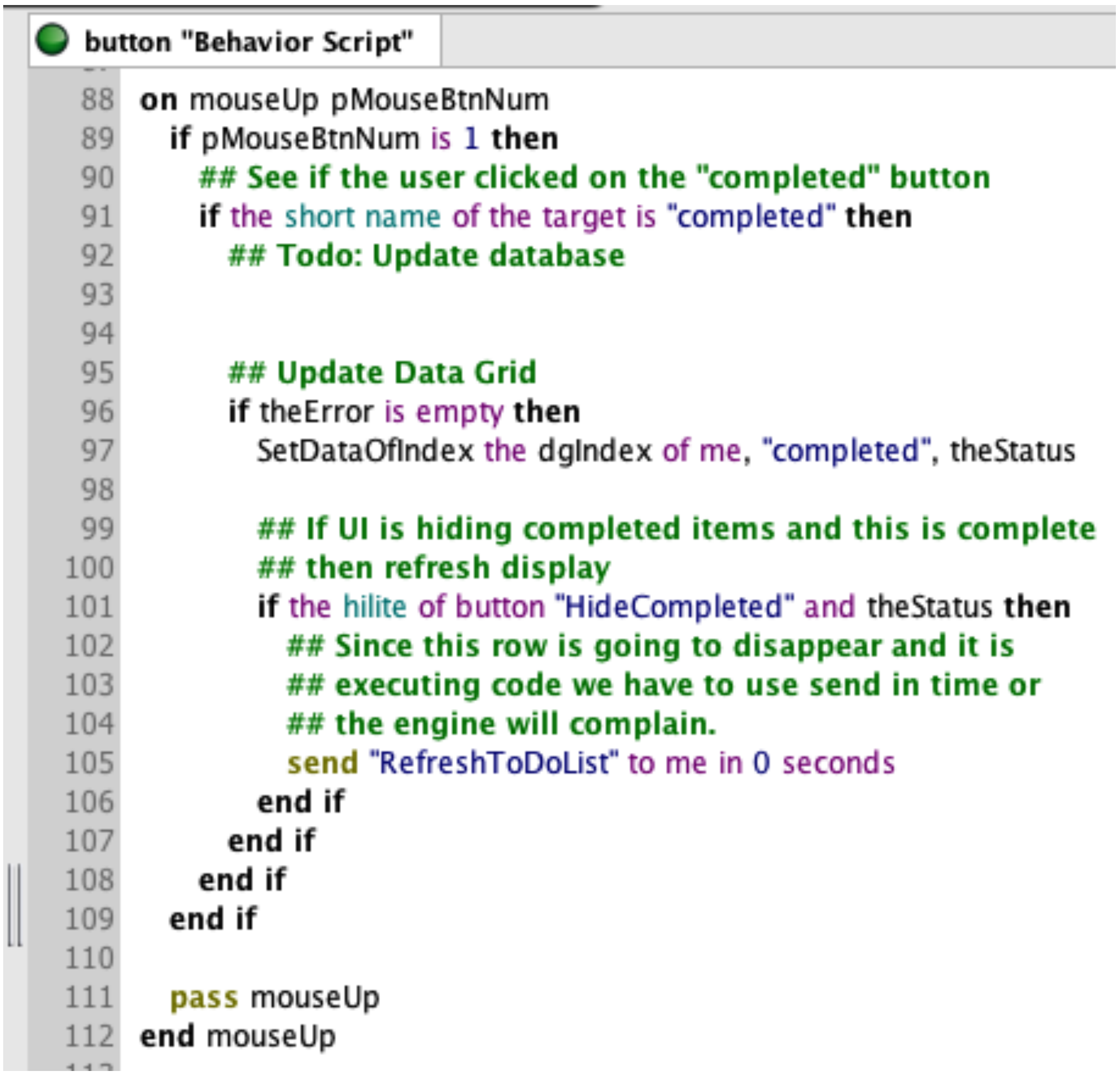
Edit ToDo Data Grid Behavior



Name: ToDo
disabled
showBorder borderWidth 0
visible ✓
Data Grid:
style: Form
show hscrollbar: true
show vscrollbar: true

Select the ToDo Data Grid and open the Object Inspector. Click on the **Row Behavior...** button to open the behavior script for the Data Grid.

Locate the mouseUp Handler



button "Behavior Script"

```
on mouseUp pMouseBtnNum
  if pMouseBtnNum is 1 then
    ## See if the user clicked on the "completed" button
    if the short name of the target is "completed" then
      ## Todo: Update database


      ## Update Data Grid
      if theError is empty then
        SetDataOfIndex the dgIndex of me, "completed", theStatus

        ## If UI is hiding completed items and this is complete
        ## then refresh display
        if the hilite of button "HideCompleted" and theStatus then
          ## Since this row is going to disappear and it is
          ## executing code we have to use send in time or
          ## the engine will complain.
          send "RefreshToDoList" to me in 0 seconds
        end if
      end if
    end if
  end if

  pass mouseUp
end mouseUp
```

You will find a mouseUp handler in the script that looks like this. Basically the Data Grid looks for clicks on the checkbox button and updates an internal Data Grid value to the current value of 'the hilite' of the button. This is where we will add the SQL Yoga code.

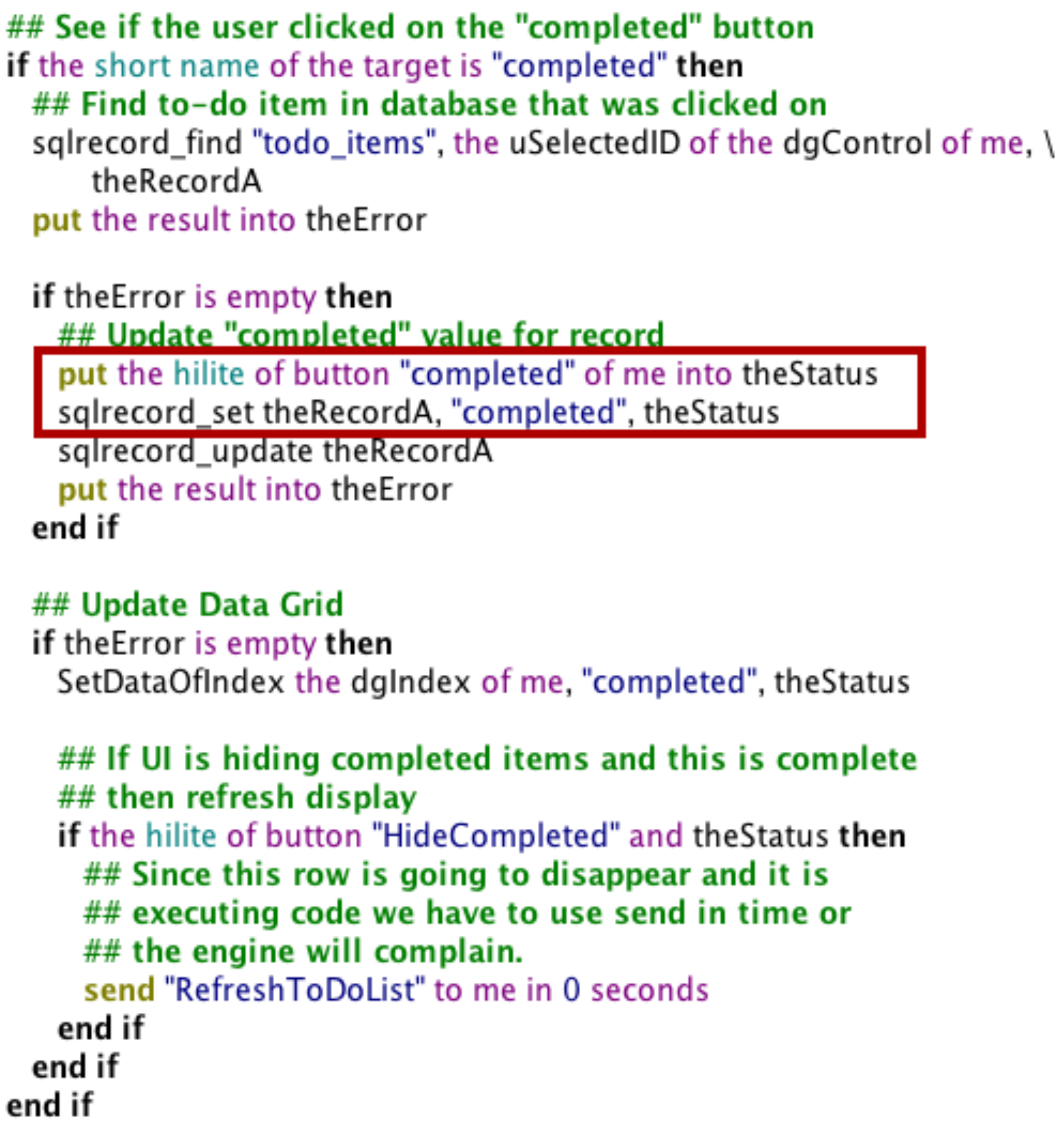
Update the mouseUp Handler

Replace the mouseUp handler in the behavior script with the following RevTalk code.

Copy & Paste The Following Code

```
on mouseUp pMouseBtnNum
   if pMouseBtnNum is 1 then
      ## See if the user clicked on the "completed" button
      if the short name of the target is "completed" then
         ## Find to-do item in database that was clicked on
         sqlrecord_find "todo_items", the uSelectedID of the dgControl of me, \
               theRecordA
         put the result into theError

         if theError is empty then
            ## Update "completed" value for record
            put the hilite of button "completed" of me into theStatus
            sqlrecord_set theRecordA, "completed", theStatus
            sqlrecord_update theRecordA
            put the result into theError
         end if

         ## Update Data Grid
         if theError is empty then
            SetDataOfIndex the dgIndex of me, "completed", theStatus

            ## If UI is hiding completed items and this is complete
            ## then refresh display
            if the hilite of button "HideCompleted" and theStatus then
               ## Since this row is going to disappear and it is
               ## executing code we have to use send in time or
               ## the engine will complain.
               send "RefreshToDoList" to me in 0 seconds
            end if
         end if
      end if
   end if

   if theError is not empty then
      answer "Error updating completed status:" && theError & "."
   end if

   pass mouseUp
end mouseUp
```

The Code

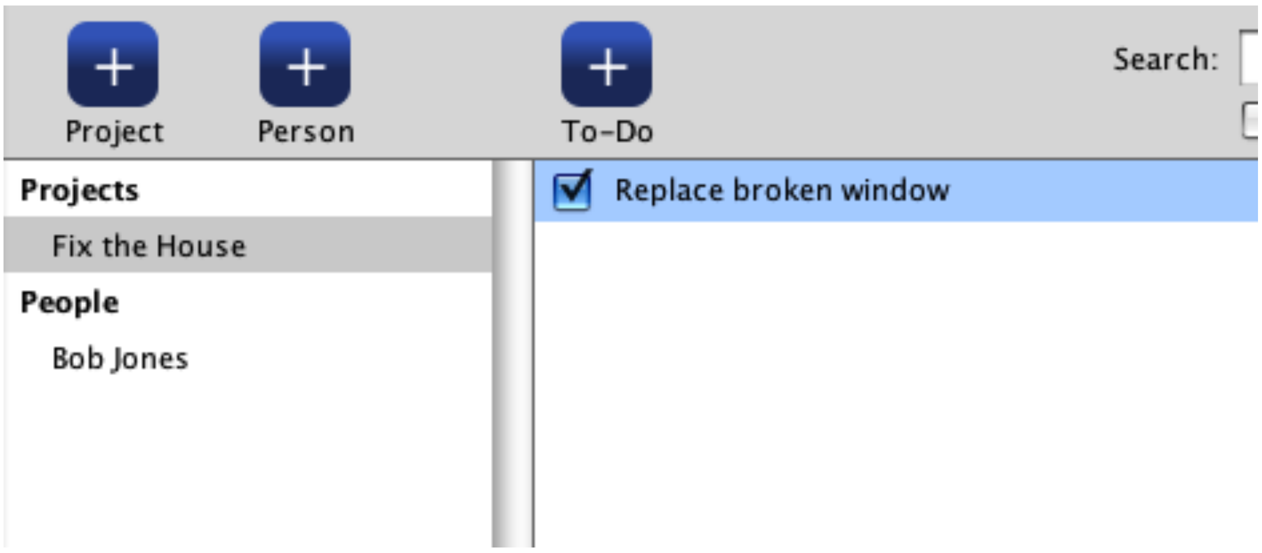


```
## See if the user clicked on the "completed" button
if the short name of the target is "completed" then
  ## Find to-do item in database that was clicked on
  sqlrecord_find "todo_items", the uSelectedID of the dgControl of me, \
       theRecordA
  put the result into theError

  if theError is empty then
    ## Update "completed" value for record
    put the hilite of button "completed" of me into theStatus
    sqlrecord_set theRecordA, "completed", theStatus
    sqlrecord_update theRecordA
    put the result into theError
  end if

  ## Update Data Grid
  if theError is empty then
    SetDataOfIndex the dgIndex of me, "completed", theStatus

    ## If UI is hiding completed items and this is complete
    ## then refresh display
    if the hilite of button "HideCompleted" and theStatus then
      ## Since this row is going to disappear and it is
      ## executing code we have to use send in time or
      ## the engine will complain.
      send "RefreshToDoList" to me in 0 seconds
    end if
  end if
end if
```

The only thing I want to point out in the code you will paste in is in relation to how Boolean values are handled when you work with SQL Record objects.

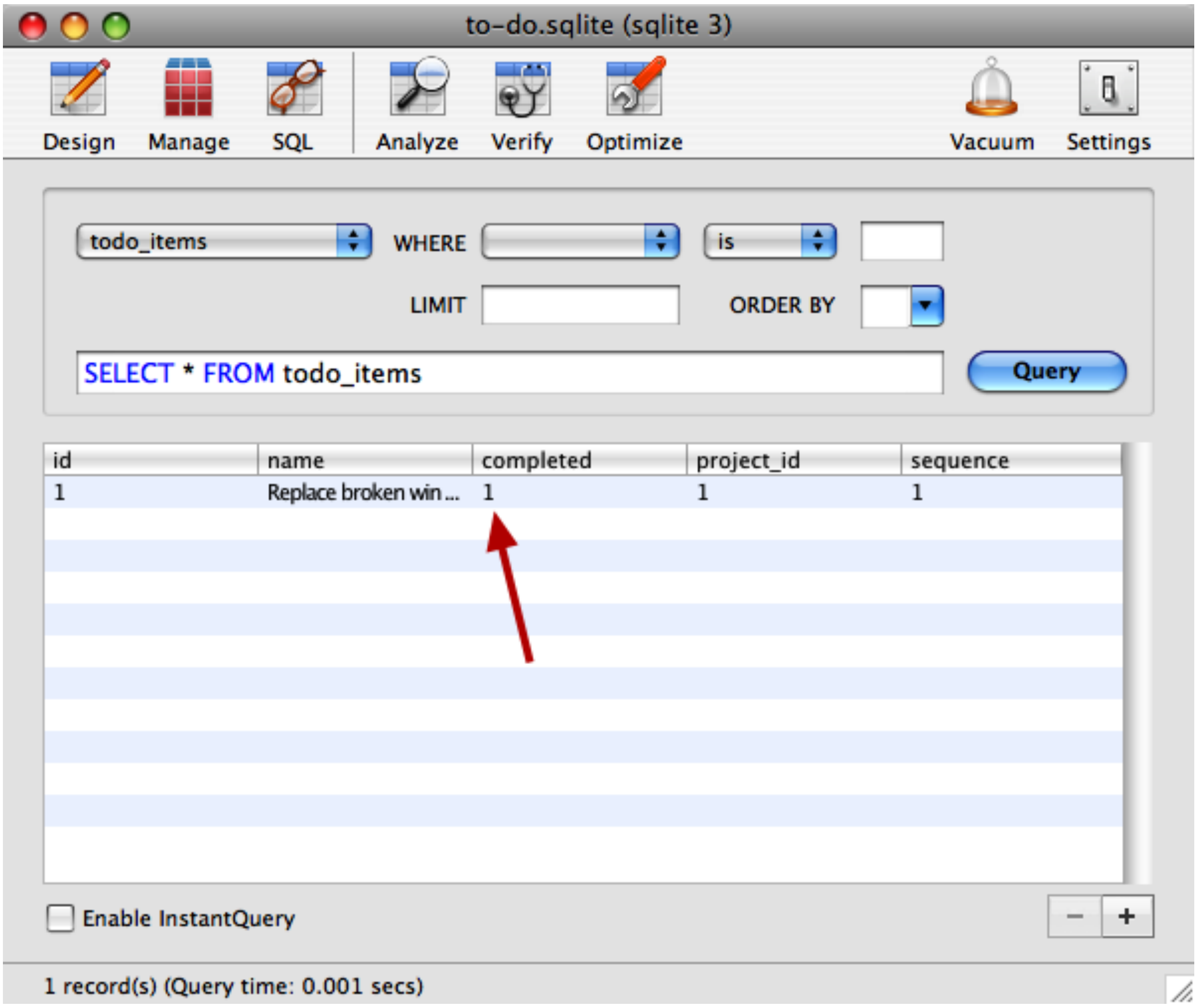
In the database the **completed** field in the **todo_items** table is of type Boolean and is stored as 1 or 0. Note, however, that the code sets the "completed" field to the hilite of button "completed". The hilite property of a button returns 'true' or 'false'. When modifying records in the database SQL Yoga will convert 'true' or 'false' to 1 or 0 respectively. When retrieving records from the database SQL Yoga will convert values to 'true' or 'false'. This is done because Revolution uses the true/false strings for boolean values. Note that this is only done when working with SQL Record objects.

Test the UI



Click on the checkbox next to the to-do item.

Verify in the Database



```
SELECT * FROM todo_items
```

id	name	completed	project_id	sequence
1	Replace broken win...	1	1	1

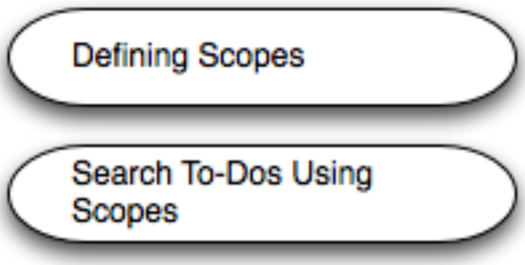
1 record(s) (Query time: 0.001 secs)

Look at the todo_items records in the database manager. **completed** should now have a value of '1'.

Adding Search Functionality

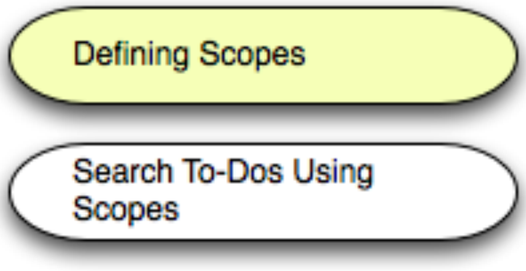
Chapter Overview

Here are the major subjects we are going to cover in this chapter.

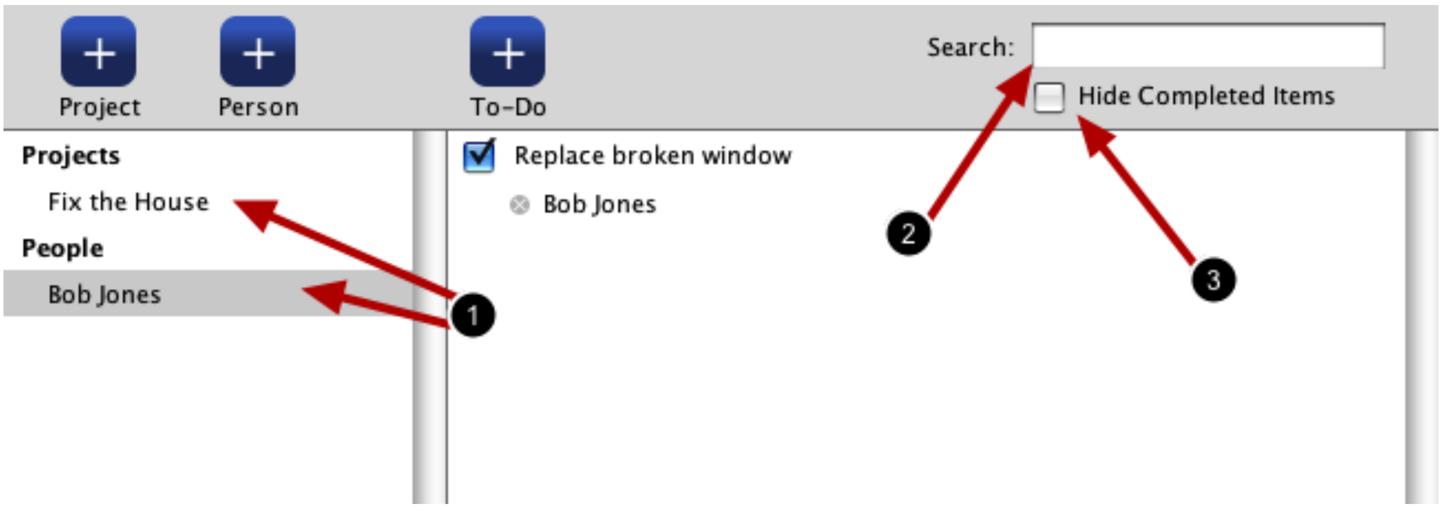


Defining Scopes To Make Aggregating Search Filters Easier

To finish up this tutorial we are going to take a look at how SQL Yoga makes managing search filters really easy.



Search Filters To Take Into Account



When displaying to-do items the application has to take three possible search filters into account.

1) Are to-do items being filtered by project or person?

2) Are there any search terms the user wants to filter the list by?

3) Should all to-do items be shown or just those that aren't completed?

The normal approach would be to have a number of conditional statements that built up a SQL WHERE clause. These tend to be hard to read and hard to debug as the number of search conditions grows.

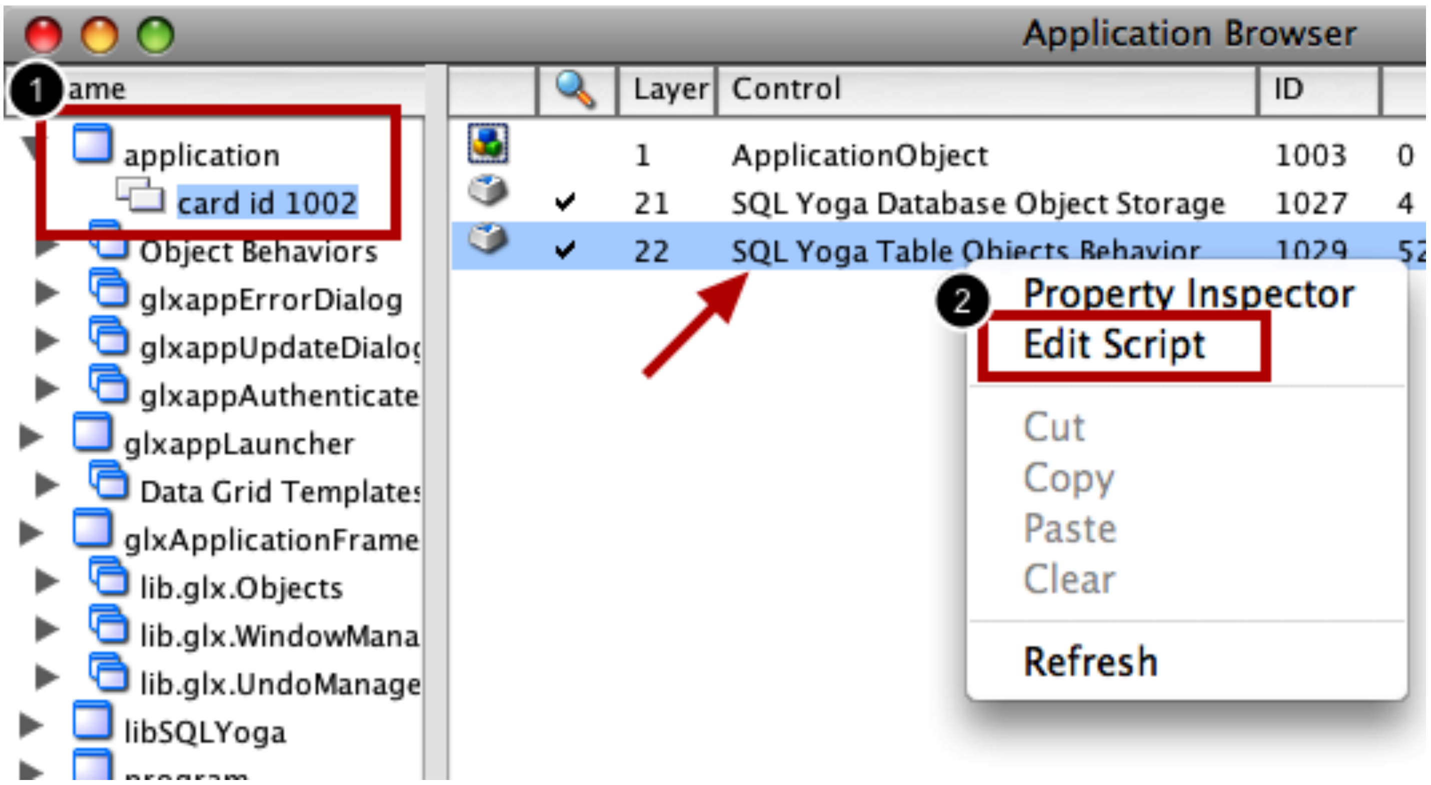
Scopes to the Rescue

SQL Yoga provides a Scope object that makes working with search filters much simpler.

A Scope object allows you to define a very narrow search condition that can be used with a SQL Query object. Scopes are useful when you have a number of search conditions that you want to dynamically apply to a search. You can easily add in the scopes you need to the SQL Query object based on user input and SQL Yoga will generate a complete search condition for you based on all added Scopes.

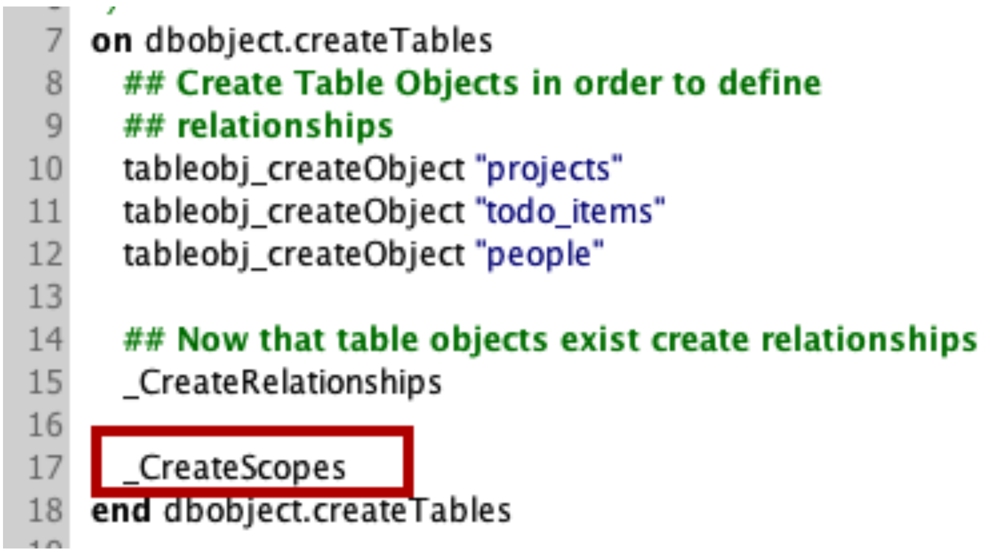
Let's look at how to create Scope objects that will help us display to-do items based on the 3 criteria.

Defining A Scope Object



You define Scope objects in the **table objects behavior** script where you created Table and Relationship objects earlier. Edit the behavior script by viewing the card controls of the **application** stack (1) and editing the script of the **SQL Yoga Table Objects Behavior** button.

Add _CreateScopes Call



```
on dbobject.createTables
  ## Create Table Objects in order to define
  ## relationships
  tableobj_createObject "projects"
  tableobj_createObject "todo_items"
  tableobj_createObject "people"

  ## Now that table objects exist create relationships
  _CreateRelationships

  _CreateScopes
end dbobject.createTables
```

The first thing you need to do is add a call to _**CreateScopes** in the **dbobject.createTables** message. You can add it just after the _CreateRelationships call.

Update Behavior Script

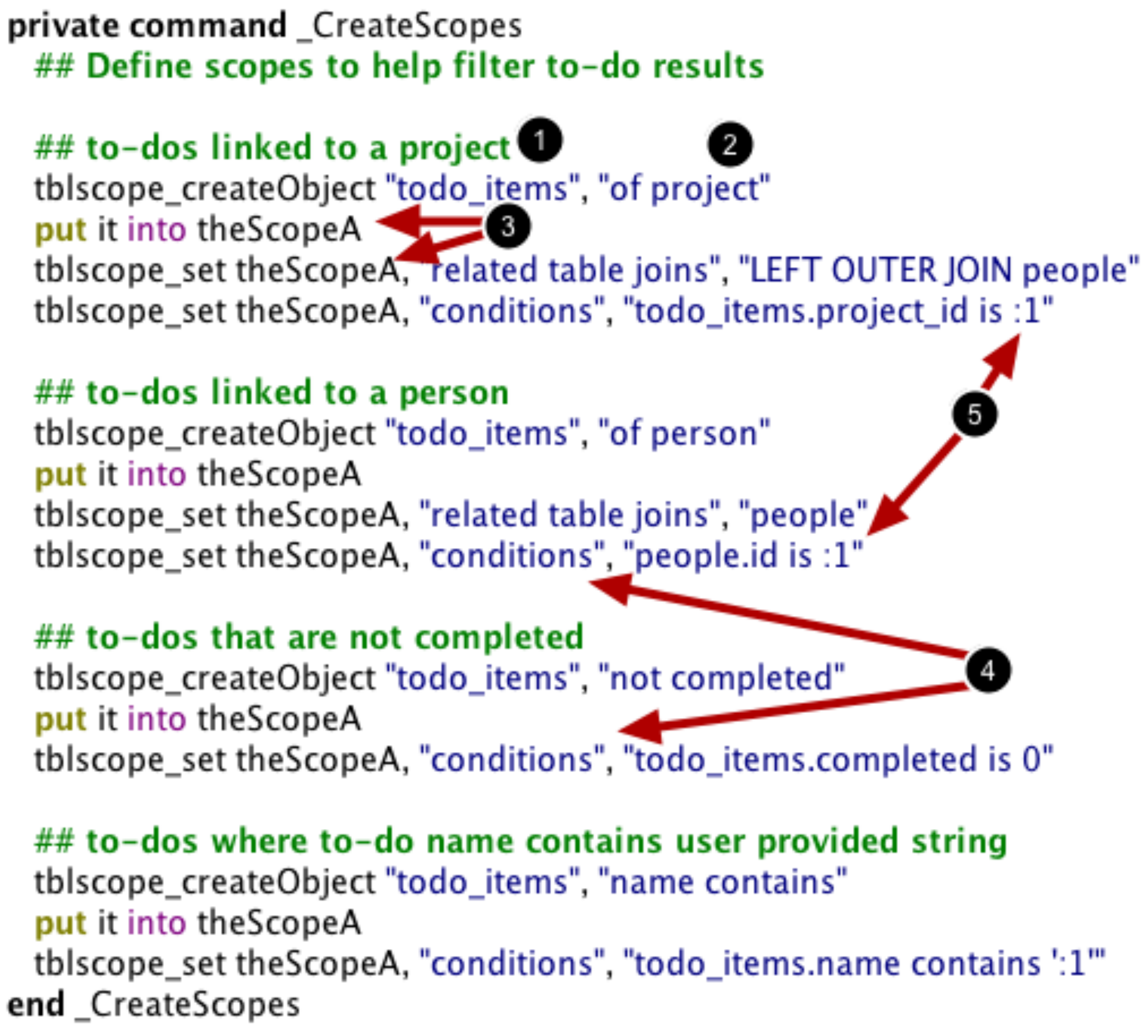
Paste the following RevTalk code at the end of the behavior script.

Copy & Paste The Following Code

```
private command _CreateScopes
  ## Define scopes to help filter to-do results

  ## to-dos linked to a project
  tblscope_createObject "todo_items", "of project"
  put it into theScopeA
  tblscope_set theScopeA, "related table joins", "LEFT OUTER JOIN people"
  tblscope_set theScopeA, "conditions", "todo_items.project_id is :1"

  ## to-dos linked to a person
  tblscope_createObject "todo_items", "of person"
  put it into theScopeA
  tblscope_set theScopeA, "related table joins", "people"
  tblscope_set theScopeA, "conditions", "people.id is :1"
```

```
    ## to-dos that are not completed
    tblscope_createObject "todo_items", "not completed"
    put it into theScopeA
    tblscope_set theScopeA, "conditions", "todo_items.completed is 0"

    ## to-dos where to-do name contains user provided string
    tblscope_createObject "todo_items", "name contains"
    put it into theScopeA
    tblscope_set theScopeA, "conditions", "todo_items.name contains ':1'"
end _CreateScopes
```

Defining Scope Objects for a Table Object



```
private command _CreateScopes
  ## Define scopes to help filter to-do results

  ## to-dos linked to a project
  tblscope_createObject "todo_items", "of project"
  put it into theScopeA
  tblscope_set theScopeA, "related table joins", "LEFT OUTER JOIN people"
  tblscope_set theScopeA, "conditions", "todo_items.project_id is :1"

  ## to-dos linked to a person
  tblscope_createObject "todo_items", "of person"
  put it into theScopeA
  tblscope_set theScopeA, "related table joins", "people"
  tblscope_set theScopeA, "conditions", "people.id is :1"

  ## to-dos that are not completed
  tblscope_createObject "todo_items", "not completed"
  put it into theScopeA
  tblscope_set theScopeA, "conditions", "todo_items.completed is 0"

  ## to-dos where to-do name contains user provided string
  tblscope_createObject "todo_items", "name contains"
  put it into theScopeA
  tblscope_set theScopeA, "conditions", "todo_items.name contains ':1'"
end _CreateScopes
```

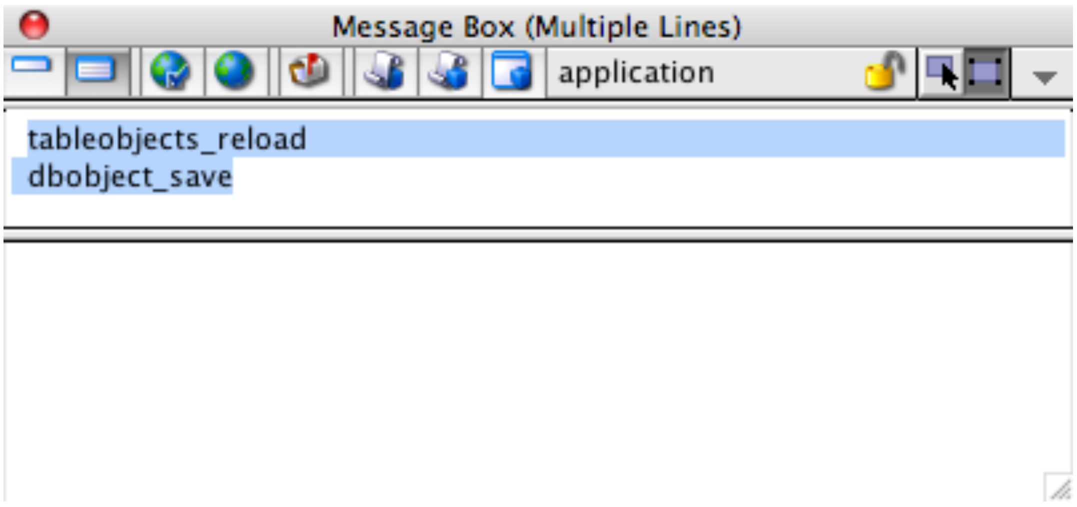
A Scope object is attached to a Table object. You create a Scope object by calling **tblscope_createObject** and passing in the name of the table (1) and a unique name (unique for that particular table) for the scope (2).

tblscope_createObject returns a reference to the Scope object that was created (3). You use this reference when calling tblscope_set.

After creating the Scope object you define the conditions of the scope (4). Four Scopes have been created above and notice that each one has a very specific search condition. By combining various Scope objects the exact search the user has requested can be performed.

Notice how most of the Scope object conditions using binding variables (the :1 in the conditions string) (5). The actual value that will be searched for is usually specified when the Scope is added to the SQL Query object.

Call tableobjects_reload

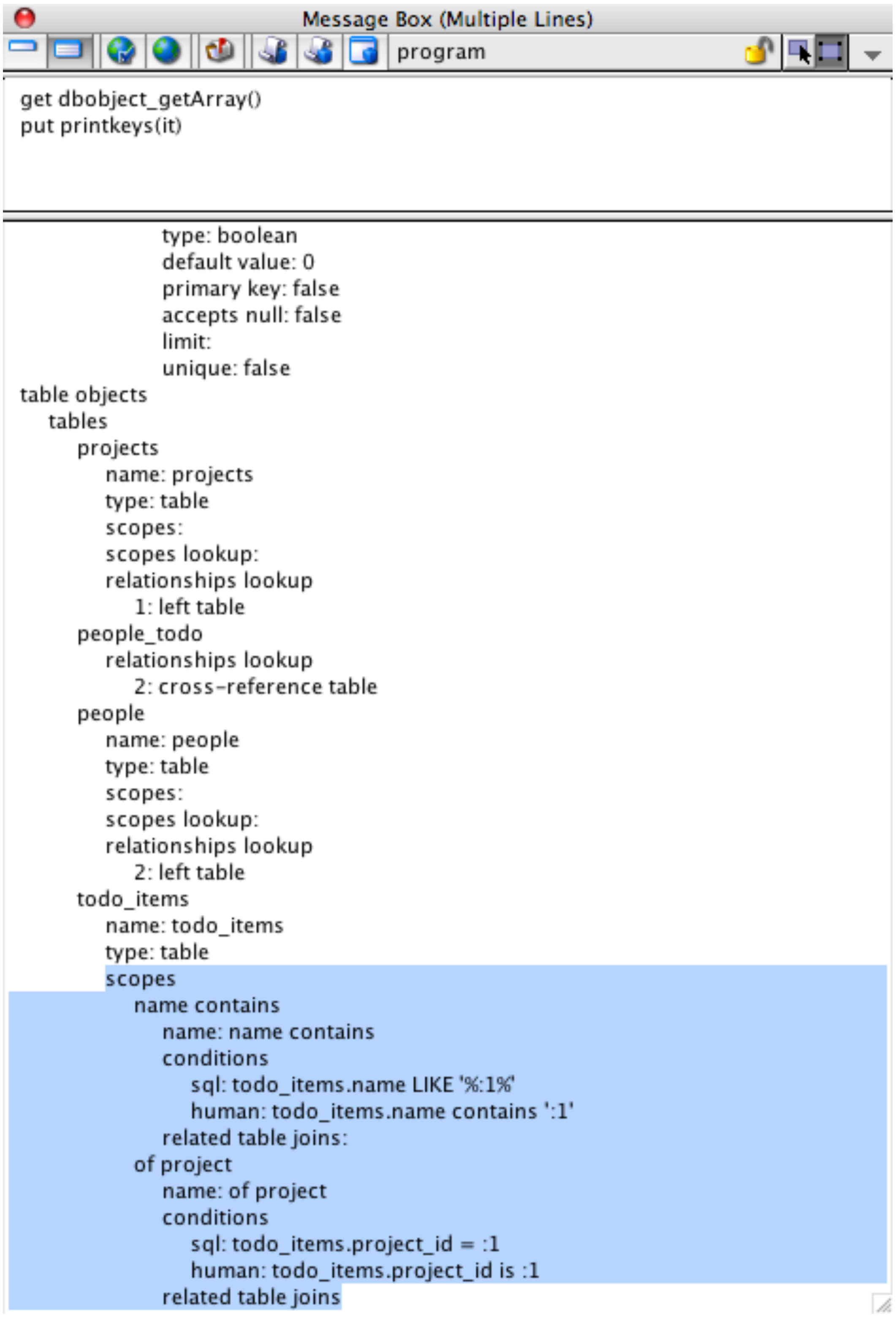


```
tableobjects_reload
dbobject_save
```

Since you are modifying the objects in the Database object you need to call **tableobjects_reload** so that the dbobject.createTables message is called.

Make sure and call **dbobject_save** and then **save the application stack** to disk as well to make the changes permanent.

Confirm Your Work



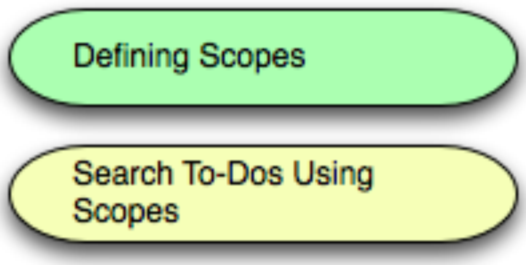
```
get dbobject_getArray()
put printkeys(it)
```

```
          type: boolean
          default value: 0
          primary key: false
          accepts null: false
          limit:
          unique: false
table objects
  tables
    projects
      name: projects
      type: table
      scopes:
      scopes lookup:
      relationships lookup
        1: left table
    people_todo
      relationships lookup
        2: cross-reference table
    people
      name: people
      type: table
      scopes:
      scopes lookup:
      relationships lookup
        2: left table
    todo_items
      name: todo_items
      type: table
      scopes
        name contains
          name: name contains
          conditions
            sql: todo_items.name LIKE '%:1%'
            human: todo_items.name contains ':1'
          related table joins:
        of project
          name: of project
          conditions
            sql: todo_items.project_id = :1
            human: todo_items.project_id is :1
          related table joins
```

To confirm that you updated the behavior script correctly check the printout of the Database object array. You should see an entry for **scopes** under the todo_items table object.

Search for To-Dos Using Scopes

Now that we have defined Scope objects we are going to revisit the uiPopulateToDos command in the card script. We will use Scope objects to rework the existing functionality as well as add support for the search field and the 'Hide Completed Items" checkbox in the UI.



Update the uiPopulateToDos Command

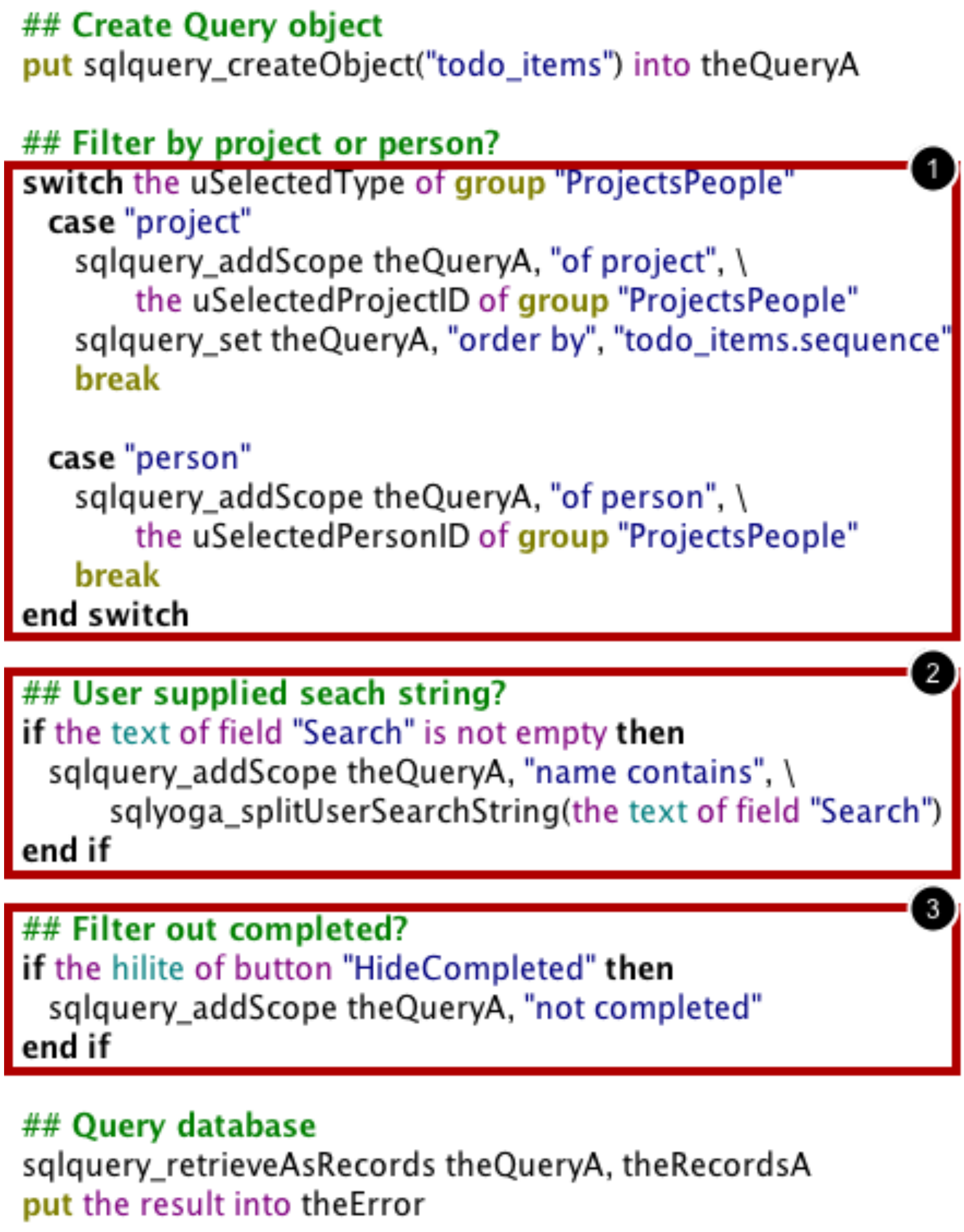
Replace the existing uiPopulateToDos handler in the card script with the following RevTalk code.

Copy & Paste The Following Code

```
command uiPopulateToDos
   local theRecordsA,theError

   ## Create Query object
   put sqlquery_createObject("todo_items") into theQueryA

   ## Filter by project or person?
   switch the uSelectedType of group "ProjectsPeople"
      case "project"
         sqlquery_addScope theQueryA, "of project", \
               the uSelectedProjectID of group "ProjectsPeople"
         sqlquery_set theQueryA, "order by", "todo_items.sequence"
         break

      case "person"
         sqlquery_addScope theQueryA, "of person", \
               the uSelectedPersonID of group "ProjectsPeople"
         break
```

```
    end switch

    ## User supplied search string?
    if the text of field "Search" is not empty then
        sqlquery_addScope theQueryA, "name contains", \
                 sqlyoga_splitUserSearchString(the text of field "Search")
    end if

    ## Filter out completed?
    if the hilite of button "HideCompleted" then
        sqlquery_addScope theQueryA, "not completed"
    end if

    ## Query database
    sqlquery_retrieveAsRecords theQueryA, theRecordsA
    put the result into theError

    if theError is empty then
        set the dgData of group "ToDo" to theRecordsA
    end if

    if theError is not empty then
        answer "Error populating to-do items:" && theError & "."
    end if
end uiPopulateToDos
```

uiPopulateToDos Reworked To Use Scopes



```
## Create Query object
put sqlquery_createObject("todo_items") into theQueryA

## Filter by project or person?
switch the uSelectedType of group "ProjectsPeople"
  case "project"
    sqlquery_addScope theQueryA, "of project", \
         the uSelectedProjectID of group "ProjectsPeople"
    sqlquery_set theQueryA, "order by", "todo_items.sequence"
    break

  case "person"
    sqlquery_addScope theQueryA, "of person", \
         the uSelectedPersonID of group "ProjectsPeople"
    break
end switch

## User supplied seach string?
if the text of field "Search" is not empty then
  sqlquery_addScope theQueryA, "name contains", \
       sqlyoga_splitUserSearchString(the text of field "Search")
end if

## Filter out completed?
if the hilite of button "HideCompleted" then
  sqlquery_addScope theQueryA, "not completed"
end if

## Query database
sqlquery_retrieveAsRecords theQueryA, theRecordsA
put the result into theError
```

Here is the meat of the uiPopulateToDos handler reworked to use the Scopes we just created. We add Scopes in order to provide the search filters that have been specified by the UI.

1) Are the to-do items linked to a project or person?

2) Is the user searching for a particular string in a to-do name?

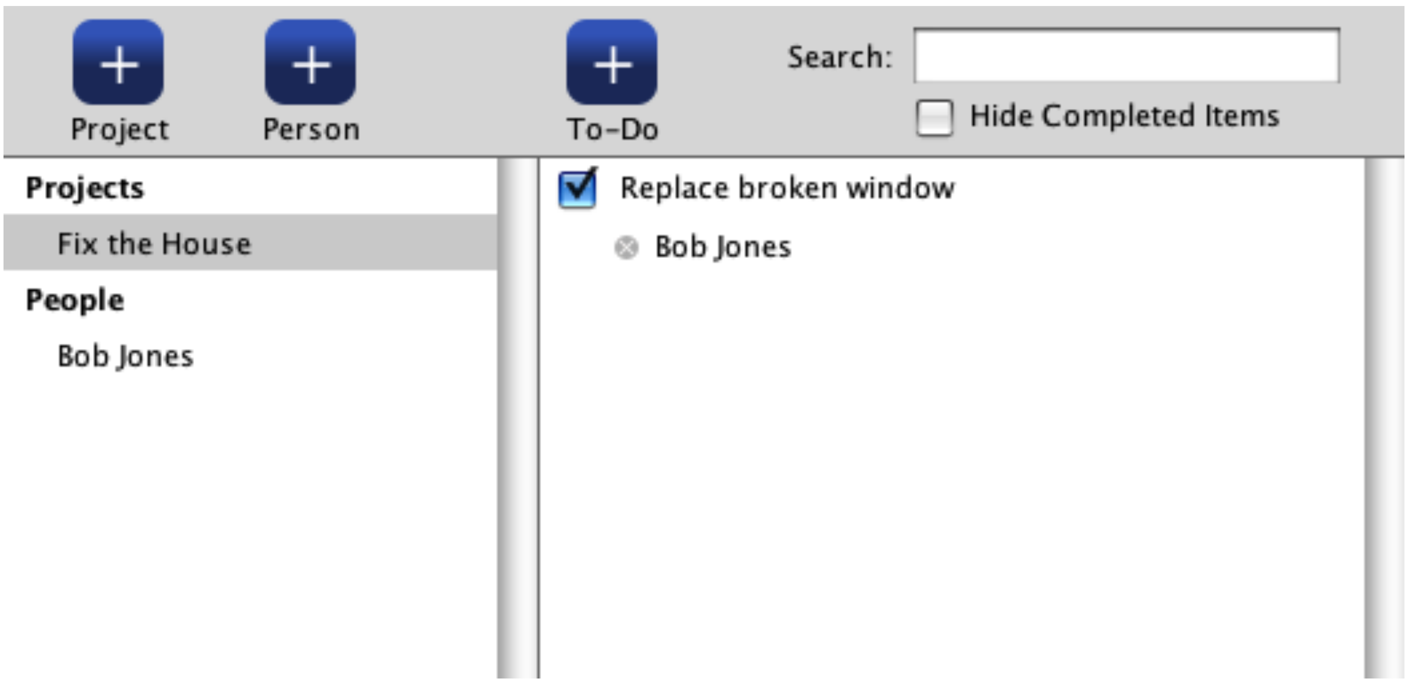
3) Should completed to-do items be displayed?

When you use sqlquery_addScope to add a Scope you pass in the name of the Scope object and then any values that will replace the binding variable in the Scope objects 'conditions' string.

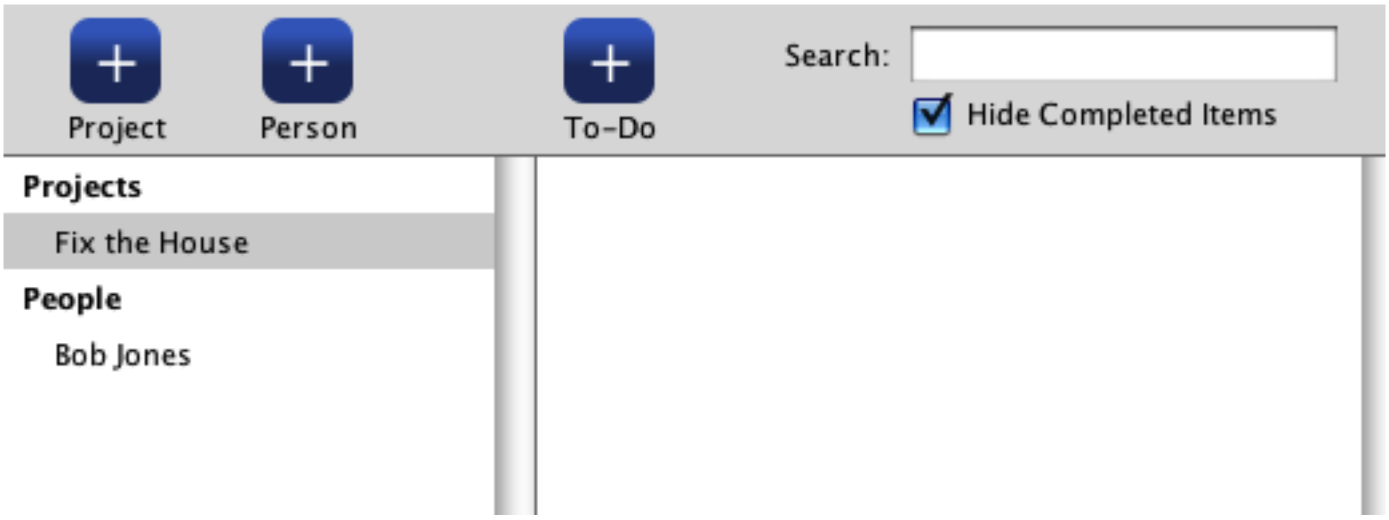
Tip: If you want to see the SQL query that the SQL Query object you just created will generate you can check the 'query' property. Try adding the following line right before sqlquery_retrieveAsRecords and you will see the query appear in the Message Box whenever the to-do list is populated.

```
put sqlquery_get(theQueryA, "query")
```

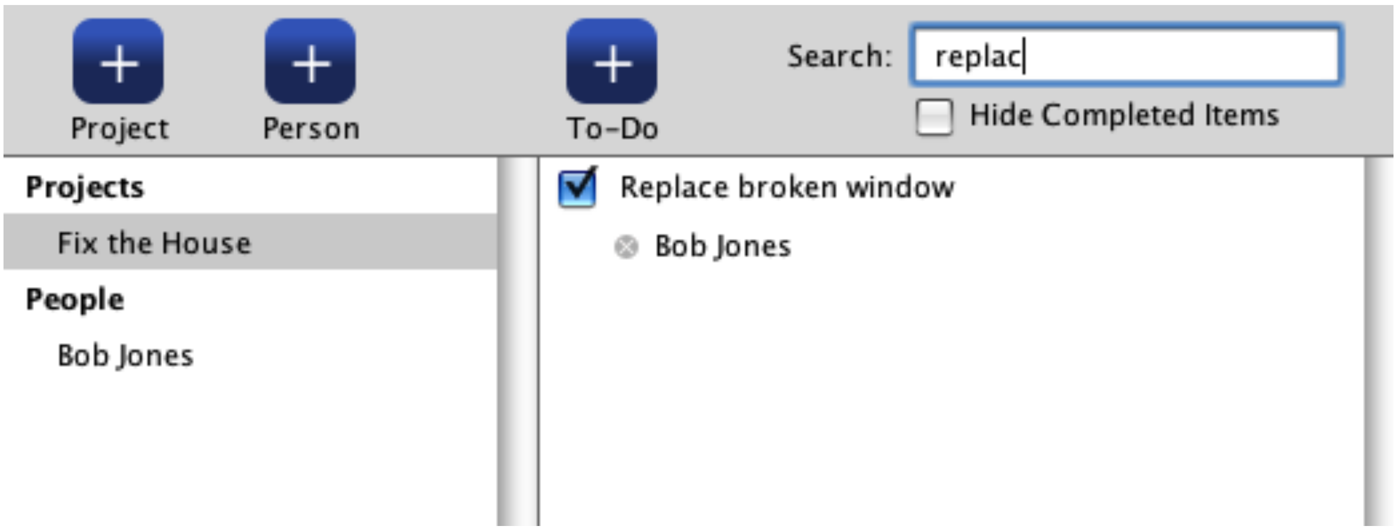
Test the UI



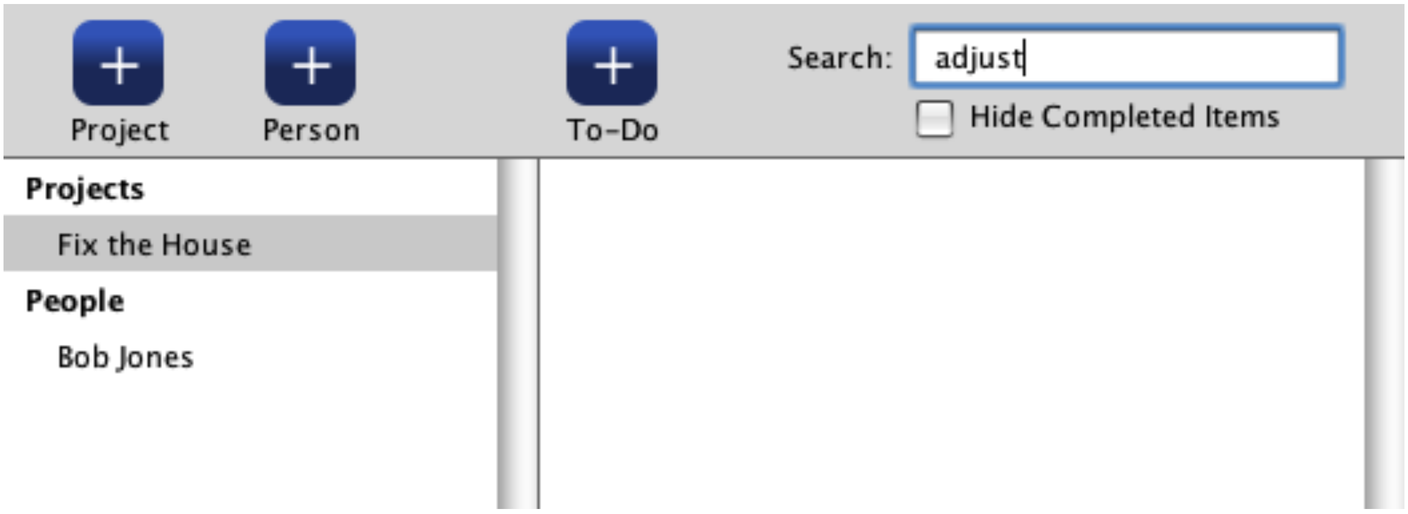
After updating the uiPopulateToDos handler you can test the UI. Make sure you have at least one project and one person create and that you have linked a to-do item to the person. Select a project or person.



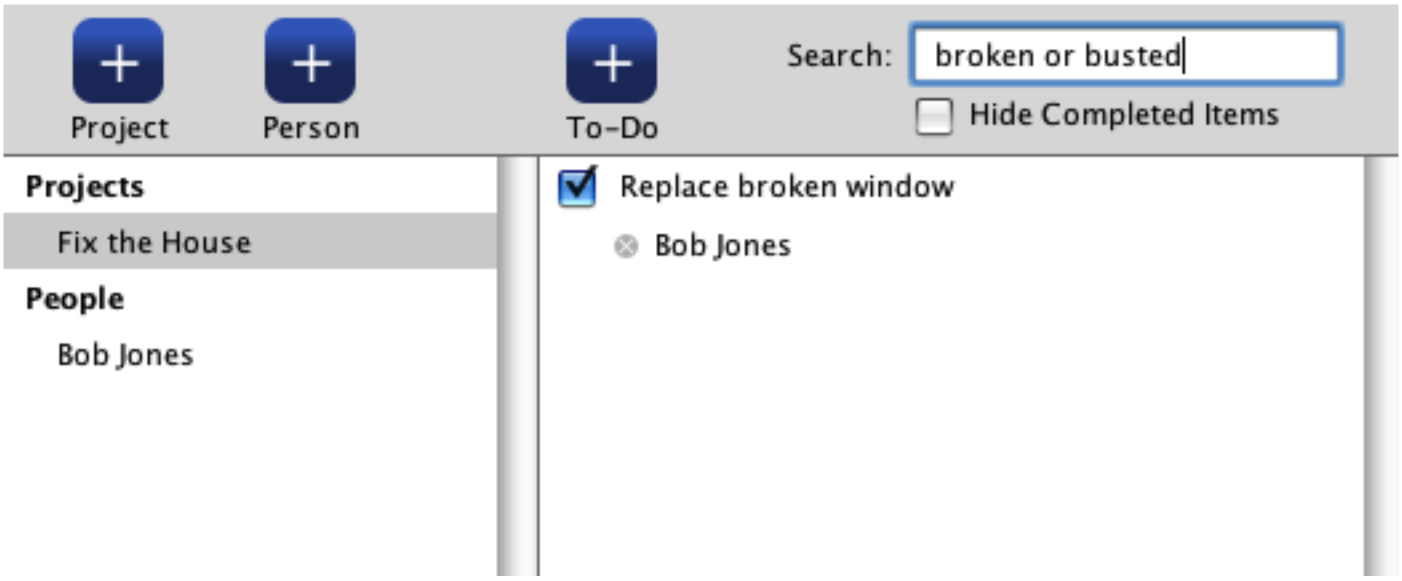
Check the 'Hide Completed Items' checkbox.



Enter a search term for a string that appears in a to-do item name and press the return key.



And then enter a search string that doesn't appear in a to-do item name and press the return key.



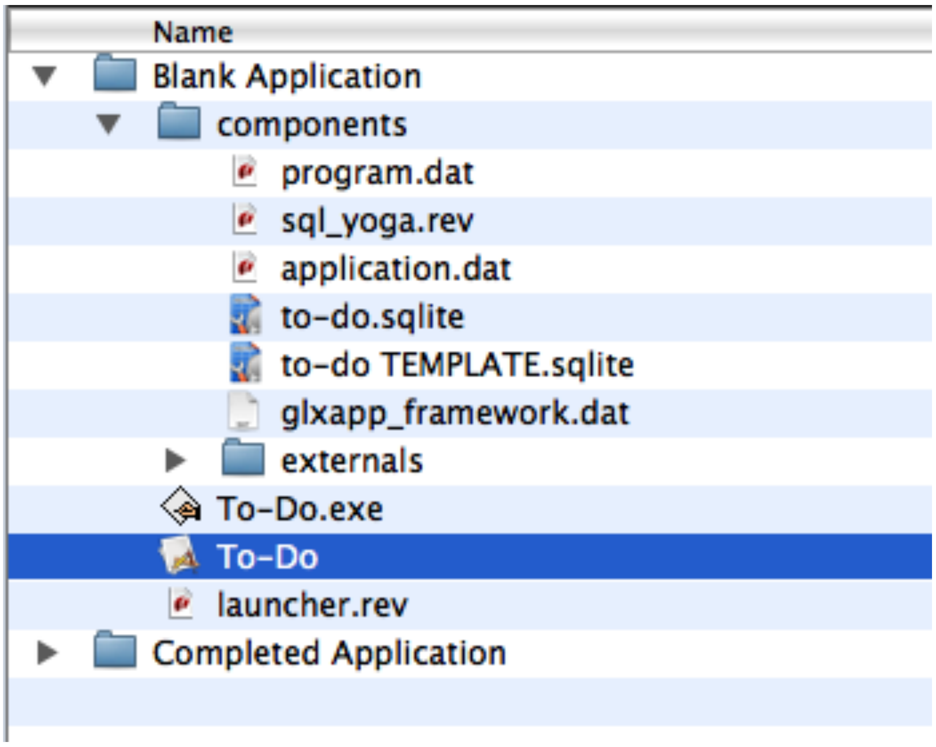
You can even enter search strings that use AND or OR operators. **sqlyoga_splitUserSearchString** will break the string up so that the records with the string 'broken' or the string 'busted' are returned.

Conclusion

Test The Standalone Version

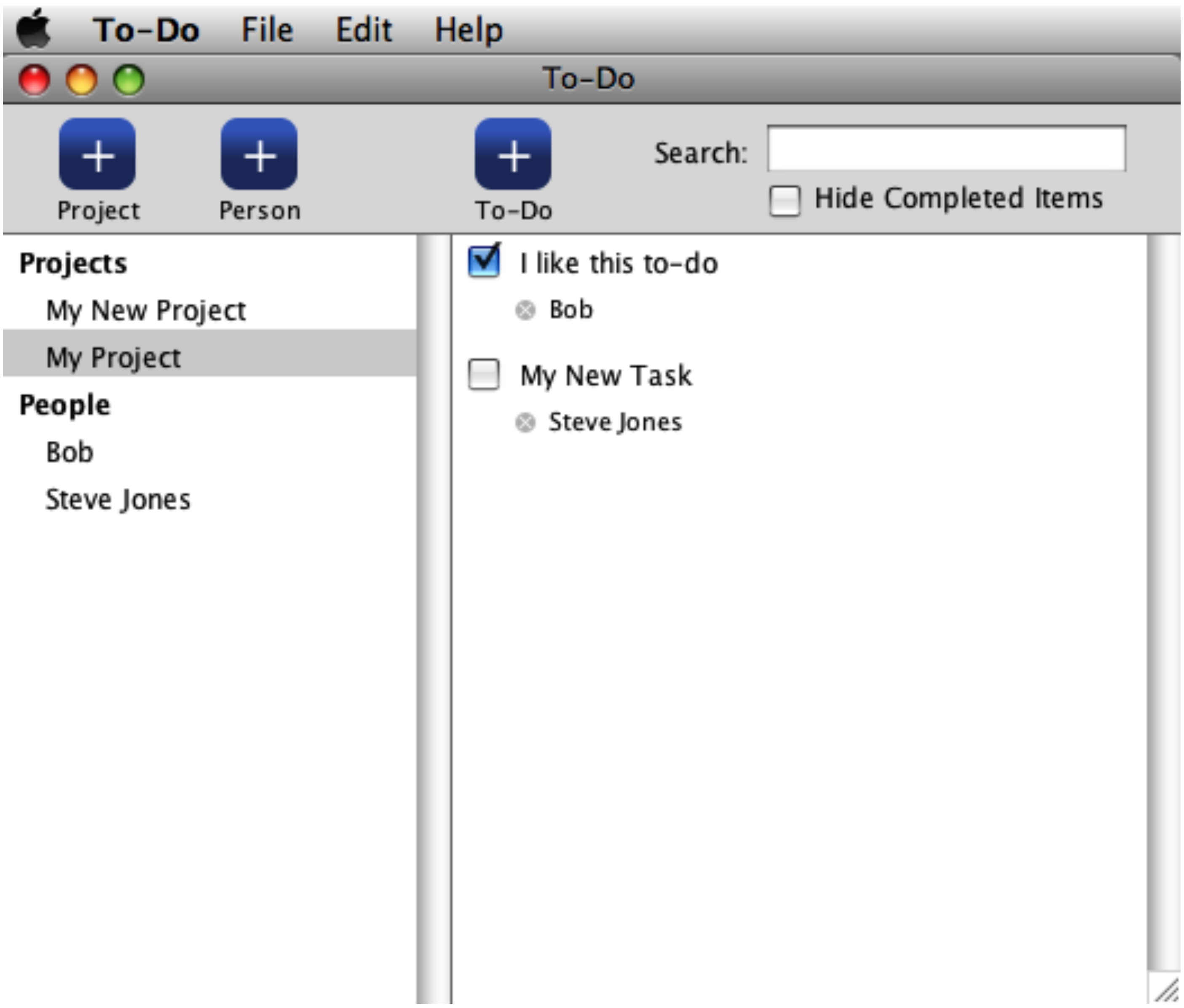
Now that you have put together the To-Do application you can launch the OS X or Windows standalone. Just make sure and save your work in the IDE first.

Locate To-Do Executable



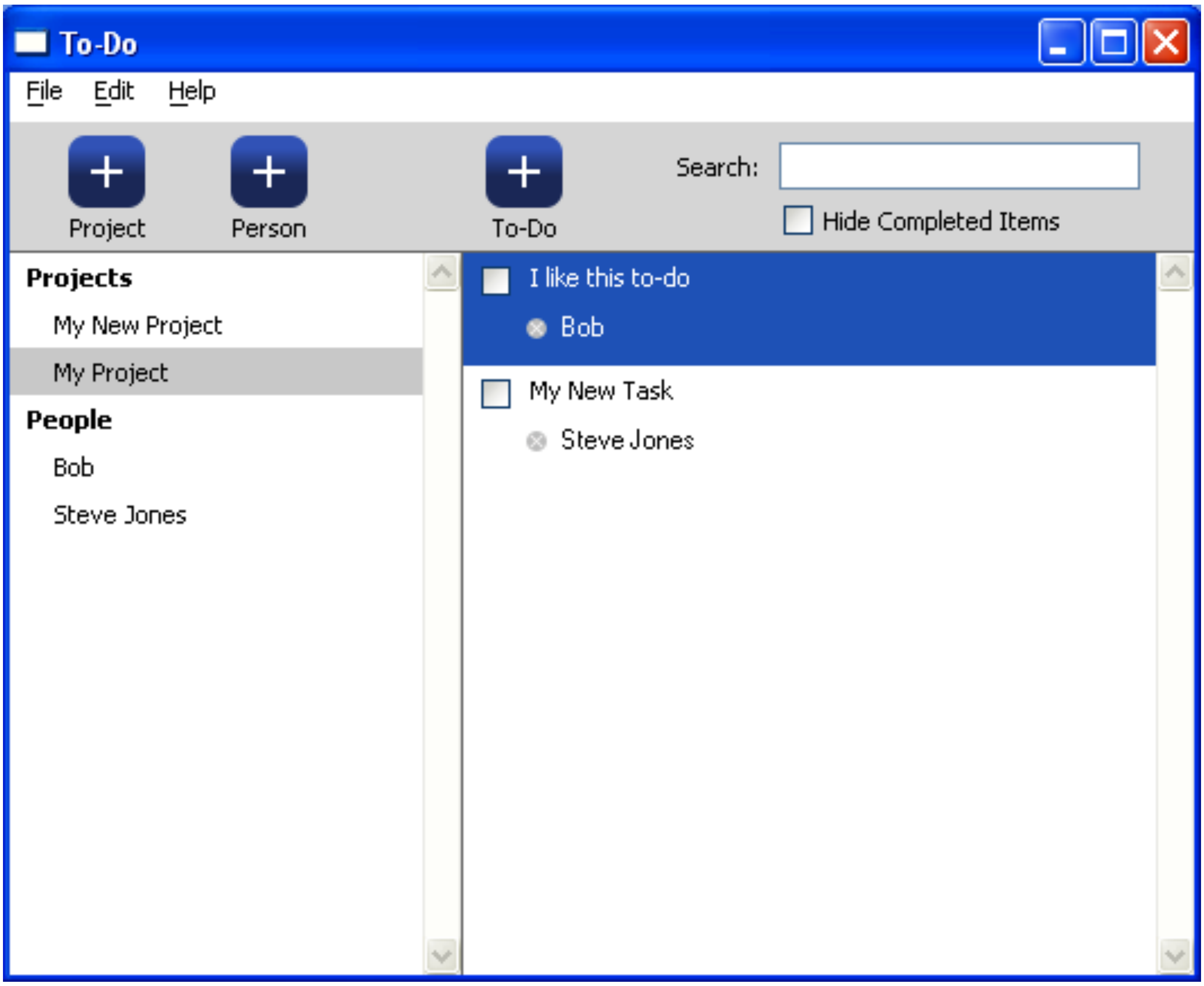
The tutorial distribution includes a To-Do executable for Mac and Windows. You can launch the executable for your platform to see the To-Do application work.

Running on OS X



- [✓] I like this to-do
  - Bob
- [ ] My New Task
  - Steve Jones

Running on Windows


To-Do
File
Edit
Help
Project
Person
To-Do
Search:
Hide Completed Items
Projects
My New Project
My Project
People
Bob
Steve Jones
I like this to-do
Bob
My New Task
Steve Jones