TDIF

TCL Data Interface Framework

Presented to the 14th Annual Tcl Developer Conference by:

Sean Deely Woods

TABLE OF CONTENTS

TABLES AND FIGURES

INTRODUCTION

WHAT IS TDIF TRYING TO SOLVE?

TDIF is a uniform methodology for accessing external data within Tcl.

Its genesis stems from a need on my part to have the same software run under multiple platforms with different drivers to access MySQL. One distro of linux would ship with tclmysql. Another would ship with mysqltcl. Windows ships with tclodbc. And then there was the change in mysqltcl's interface between mysql_ and mysql::.

At the same time sqlite was coming into it's own, and I found myself replacing some MySql applications with it. I was also having to occasionally dump data out of MS Sql and MS Access. Being a lazy programmer, I developed a shorthand for all of the various database interactions I needed, and then wrote a suite of tools to convert that shorthand to the native interface of the storage engine.

TDIF is my attempt to adapt my own techniques, developed over time, into a formal interface.

TDIF PHILOSOPHY

TDIF caters to two different audiences.

One audience is simply looking for a consistent way in which to feed bare statements into a database engine. They want abstraction only so far as to ensure no matter what driver they use to access XSql, they get data returned in a predictable format. We will call this BMI: Bare Metal Interface.

This audience can pretty much skip to the chapter on Usage, most of this paper describes the full blown TDIF. Though the chapter on the Implementation may be an interesting read for those who wish to add new drivers.

The second audience does a lot of work accessing data from multiple data sources at once. For them, optimal database statements are a secondary concern. Their principle need is to transparently load and save data regardless of the database engine used. If indeed, the data is coming from a database at all. This interface uses all of the capabilities of TDIF: Tcl Data Interface Framework. TBMI is still available for the occasional optimization, or function that TDIF does not provide for.

In TDIF tables, records, and columns are abstracted into containers, elements, and properties (though I still call them 'fields' or 'columns' out of habit). This interface reduces all storage engines to a port in which to drop key/value lists.

I use TDIF as a framework on which to build much more complex systems. Reducing everything to key/value lists means that a webserver application no longer needs to worry about formulating queries. A user migration script can simply dump a list from one engine to another. An application writer can switch storage engines as a customization rather than a full-blown port.

TDIF CONCEPTS

Node

A node is simply an object within TDIF this is otherwise not defined. Structurally, tdif.node is a class that contains common operating elements of all the other object types. Method names use "node" because most of the time TDIF does not know or care what type of object is referenced.

Containers

While it us useful to think of a container as a data table, don't wrap yourself up in it. A data table is only on kind of container in TDIF. Standalone files are another type. A bank of instruments could be another container. In short, a container is any object in TDIF that acts as a grouping for other objects.

Containers can, in fact, contain other containers. Connectors are treated as a type of container. Instead of spawning database record objects, they spawn container objects for tables and property handling objects for columns.

A container can also spread it's data over connectors, or multiple tables within a connector, or conceivably multiple tables over multiple connectors. The point is, a developer simply writes and reads data to the container object. The container object works out all of the details on where data is stored and how.

Connectors

Connectors are a special kind of container that goes out and directly communicates with an outside data store. Other containers use Connectors to interact with the outside.

Database connectors spawn off other container objects. File connectors spawn off the records they contained with the file. Undoubtedly even more exotic arrangements will pop up as we go along.

The important takeway is that a connector is a container that has a few extra methods for interacting with the data store. Sql connectors also provide some creature comfort utilities for containers to formulate queries with.

Elements

Elements are objects that take data stored in records, and make them come alive within a TDIF application. Unlike containers, they do not contain other objects. Though that can link to other objects. Elements can also have data that spans multiple containers. (Though I suppose these should rightly be called **compounds**.) An element can also be present in more than one container. The key thing is that it deals with all of the data storage issue. All the application writer has to do is read and write key/value lists.

Properties

Properties are a special kind of node that controls how data being interacted with is formatted and validated. Think of them as objects that represent the columns in a database. If a container or an element want to generate a form, they pass the current value for each field to the property object, and it returns an Html input snippet. If the application wants to check if a new value is in a valid range, it can ask the property.
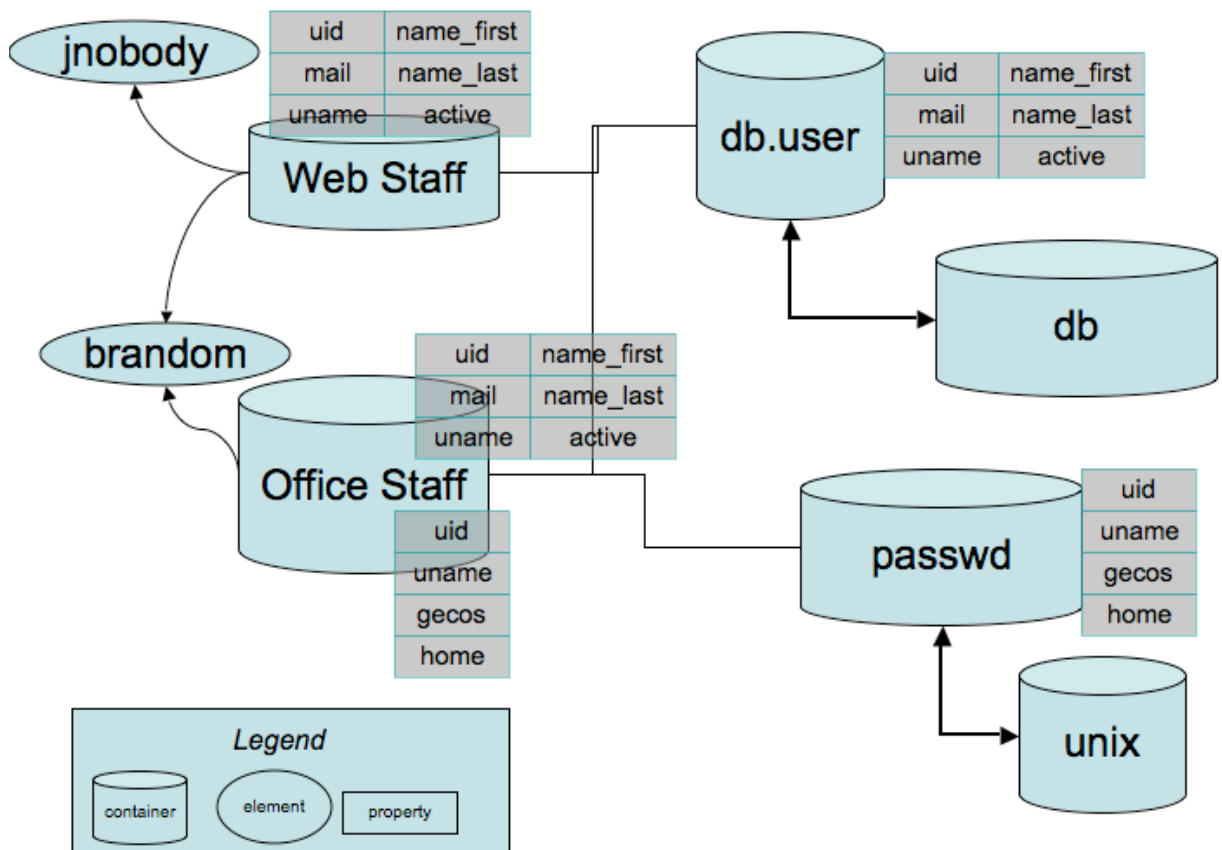
Putting it all together

This diagram shows all of the different concepts in action:

Figure 1 - TDIF in Action

The above example is a system in which we have different types of people. One is our office staff, who in addition to an entry in the online stafflist, also have an entry in the password table in unix. We also have frontline staff who come and go, but have no need (nor would we wish to grant them access to) our Unix system. A frontline staff element has all of the properties from the **db.user** table. Office staff have the properties from the **passwd** table in addition to the **db.user** table. Our element understands how changes to a field on one system need to reflect changes that take place in another.

For instance, if we change someone's last name on the staff directory, we also need to update the GECOS field in Unix if they are to be displayed properly. And of course some admins would find that writing directly to the *passwd* file to be a bit repugnant. So the **passwd** object makes all of it's updates through the *usermod* function. Also, no engineer worth his salt would put his security system at the mercy of his website, so the system call is done on the webserver's behalf by an another program. How the **passwd** container does its job is immaterial to us. That is the joy of abstraction.

USAGE

Let us building on our hypothetical staff directory/unix security bridge. We have a user "Betty Random." Betty marries a man with the surname "Stochiastic." Betty, being an old fashioned kind of

gal, changes her name to Betty Stochiastic. Of only to avoid writer's cramp from trying to sign Random-Stochiastic on checks. Because this sort of thing happens a lot in any vibrant organization, we want to design a simple way for our non-technical users in HR to be able to regularly update both the staff list and the name that appears on our unix hosted file server and domain controller.

A snippet of TDIF code to perform this update would look like this:

```
# Create the connector objects
tao::create_object maildb [lappend $dblogin class ::tdif::connector::mysqltcl]
tao::create_object passwd [class unixEtcPasswd]

# Create the staff directory container, and link it
tao::object_create staff [class staff_directory dbCon maildb passwdCon passwd]
tdif Container staff staff

# Spawn an object to handle brandom's account
set uObj [tdif spawn_object staff brandom]
$uObj displayName
Betty Random
$uObj Get
uid 1337 gecos {Betty Random} name_last Random name_first Betty
uname brandom active 1 mail brandom@fi.edu
$uObj Set {name_last Stochastic}
# Note that the name field has ALSO been updated
$uObj Get
uid 1337 gecos {Betty Stochastic} name_last Stochastic name_first Betty
uname brandom active 1 mail brandom@fi.edu
# Send changes back up to the containers. In the process, destroy the object
$uObj renew
passwd nodeGetField brandom gecos
Betty Stochastic
```

Example 1 - Using TDIF

The example above glosses some important details:

- How did we write the classes that drive this example?
- How does the object decide what fields go to which container?
- Which methods are standard TDIF, and what have my examples created?
- What does that "tdif Container" step do?
- What is that "spawn_object" step?
- Why does the table have the name **staff** in some places and **staffObj** in others?

All of these questions, and more will be answered in the course of this paper. We will begin with TCL interface, and work our way into creating new objects and finally creating new connectors.

Bare Metal

Many developers are not going to want TDIF to handle their table objects, nor provide transparent updates to things. What they do want is a consistent face with which to communicate to their database of choice. For these users we provide a subset of functions called the "BMI", the Bare Metal Interface. Essentially if you have the classes for the connectors already defined, you can call them into being and be on your merry way without all this talk about containers, elements, properties and whathaveyou.

What you do get is a consistent interface for any database the Tcl has a driver for, and someone has taken the time to code a TDIF wrapper. Here is a simple example of BMI in action:

```
# Call the connector object into being
tao::class maildb [class tdif.connector.mysqltcl \
     db_user smtpd db_pass password database maildb db_host localhost]
# Cry havoc an let loose the dogs of war
maildb query_flat "select email from user where username='swoods'"
swoods@fi.edu
maildb cmnd "update user set name 'Evil Twin Skippy' where username='swoods'"
```

Example 2 - Bare Metal Use if TDIF

Every TDIF connector exposes the following methods. They assume you know the native language of the database. Results are returned as a list, or a flat list. Doesn't sound all that thrilling, but it's actually the first feature that led me down the road to developing TDIF in the first place:

Method	Description
cmnd *statement*	Send a statement to the database that does not return data
query *statement*	Send a statement to the database the returns data, and format it as a list. Each row is one list item
query_flat *statement*	Send a statement to the database the returns data, and format it as a flattened list. Each column is one list item. Useful for feeding into foreach statements.
describe *table*	Using whatever means available, returns a dict that lists all columns and indexes for a table object
sqlfix *string*	Escape out any special characters in a string that could be interpreted as part of the statement
sqlprep *string*	Format a string complete with quotes to mark it as a value to be inserted as a column

Table 1 - Bare Metal Connector Methods

Connector objects are free to add additional methods to exploit the special features of their particular database engine, provided they do not overwrite one of the currently meaningful keywords. Keep in mind, connectors ARE containers, so in addition to the above 6, so all of the Container keywords also apply. See the appendix for a complete (as of the date of this writing) list of methods.

By convention, if your connector contains other sub-storage elements, container operations interact with meta-data, tables, columns, etc. If your connector goes right to a table, or some other flat data storage, container operations interact with its records.

containers

Containers are a corpus of data. Developers use them to provide a consistent interface to data tables and other logical storage units. Containers do not technically store data. They simply pass the messages between the nodes that use the data and the engine where the information is stored. They are also responsible for calling **element** and **property** nodes into being, as well as cleaning them up when the container is destroyed.

Some containers need to speak through a connector object. Others are tied directly to the data store. The difference is immaterial to the programmer. Their job is essentially to read and write key/value lists to a node identified by the nodeId.

A container in action (note we are re-using some of the objects from example 2):

```
# Create the table object using a class that reads the schema from the connector
::tao::object_create staffObj [class sqltable.static.autodetect \
    table maildb.user sqlObj maildb]
# Associate staff as a tdif container
tdif Container staff staffObj
staffObj nodeGetField 1104 name
{Evil Twin Skippy}
# Send an update through the container
staffObj nodeSet 1104 {name {Sean Woods}}
# See that the change has actually gone through
maildb query_flat "select name from user where username='swoods'"
{Sean Woods}
staffObj nodeGetField 1104 name
{Sean Woods}
```

EXAMPLE 3 – BASIC CONTAINER USAGE

Containers expose the following methods to developers:

Method	Description
nodeSet *nodeId keyValueList*	Update set contents of *nodeId* with the values in *keyValueList*
nodeGet *nodeId*	Return the entire contents of *nodeId* as a key/value list
nodeGetField *nodeId fieldName*	Return the value of *field* stored in *nodeId*
nodeGetFields *nodeId fieldList*	Return the value of each *field* in *fieldList* stored in *nodeId* as key/value list
nodeDelete *nodeId*	Delete the node stored at *nodeId*
nodeClass *nodeId*	Return the class that defines how the node at *nodeId* should be generated
nodeAlias *string*	Return the address of a node that is associated with the alternate identification of *string*
train *nodeId*	Return a dict that is used by the object manager to create a spawned node object. Must include a value for **class**, **globalName**, **node_id**, and **containerObj**
tdifAttach	Method called when a container is associated with tdif.
/container	If this container is a child of another container, return the object handle of the parent.
/node *nodeId*	Return a node object to represent the data in *nodeId*
/property *propertyName*	Return an object handle for the **property** node that validates and represents the field **propertyName**.
spawnedNodes	Return a list of all nodes this container has spawned

TABLE 2 – BASIC CONTAINER EXPOSED METHODS

SQL Containers add a few additional methods.

Method	Description
/sql	Return the connector object of the Container
tdifAttach	Method called when a container is associated with tdif. SQL table objects use this method to populate the connector object with meta-data
schema	Return a dict with sub-elements that define the columns, indexes, and primary key of the table
nodeNext	Return the next available *nodeId*
nodeMatch *keyValueList ?like?*	Return a list of *nodeIds* that match the properties given by *keyValueList*. If a 1 is given for "like", the system matches using a pattern match instead of literal value.

TABLE 3 – SQL CONTAINER EXPOSED METHODS

Container Handles

One may ask, why the manual step of running *tdif Container*? Truth be told, there is nothing keeping you the developer from adding that step to an object's constructor. The reason I include it in the example is to show that **tdif** is designed to play well with other object systems. It also is a nice way for me to mention the global record addressing system that **tdif** uses.

Suppose you wish to link records in two different containers. Yes, yes, if they are on the same SQL system you can create foreign keys and whatnot. But in script, say we have an entry in MySQL we wish to have associated with an entry in Sqlite. There is no realistic way to bind these things natively in the data store, so **tdif** has to do it in script.

To facilitate this kind of linking, **tdif** associates each container with a handle. During setup, the handle is associated with the object that is the actual container. A container can answer to many handles, but each handle can only be associates with on container at a time.

When we link records within the same container, we simply list the record ID numbers. (Or whatever the primary key is.) However, if we link to a element that is in another container, we add the handle of the other container. Because this link is stored in the actually data backend, and because we have no idea what the actual object that defines the container will be called in future implementations, we refer to it by this made up handle.

What handle a container answers to is entirely the responsibility of the developer to maintain. Though one convention I've developed for my webserver is to associate **users** with whatever container stores my user list, and **groups** with the grouplist. In my access control system, I can simply store links to **users-1104** and rest assured that however we are accessing the user list, as long as I keep a global number paired with an employee id, I'll be referring to user 1104, aka swoods, better known as me.

I've used this system for 6 years, and through countless technology changes. It works, even if the object systems change, tables move, and data technologies change. In my case the user list has moved from tfi.staff to mailbd.user, and will probably be an entry in LDAP pretty soon.

The other reason is to uncouple container objects from the same container system used to define TDIF. Because all of the interactions take place through object methods, there is nothing to keep a container (or any other object for that matter) defined in SNIT or XoTcl from working with TDIF. (TDIF itself is written in Tao). So not only could **users-1104** refer to database element in a completely different data store, the class that defines it could be written in an entirely different object system.

So why the convention of internally links be just the id, and external links be the container-nodeId? Because handles can change. If I decide to start referring to **users** as **staff**, how will I maintain all of the links that have already been created? What if a container is both **users** and **staff**? You start having to check all names for the container past, present, and possibly future just to find simple links between records. Not a happy state of affairs.

It doesn't hurt that a lot of db admins also just store record numbers in indexes, so this convention lets me rip link data direct from the database.

elements

Elements are packets of data brought to life as objects. While they express the greatest variety of all objects in TDIF, they have the simplest interface. Elements are almost never created directly. They are spawned by their containers.

Here is a short example of elements in action. Again, this example builds on the previous example and makes use of the containers and connectors already created:

```
# Spawn an object
set recordObj [staffObj /node 1104]
$recordObj globalName
staff-1104
$recordObj Get name
{Sean Woods}
$recordObj Set {name {Evil Twin Skippy}}
$recordObj renew
staffObj nodeGetField 1104 name
{Evil Twin Skippy}
```

Example 4 – Basic element usage

In the above example, we spawn off an element object, write a value to it, and then save the changes back to the container. **renew** is an inside joke with me. It uploads changes and then calls the **carosel** method, that puts the object in line to be destroyed on the next pass of the garbage collector. (It's a reference to the movie *Logan's Run*.)

Why does upload lead to the destruction of a element object? It's the only way to consistently handle elements that change class based on their contents. At least in my experience.

Take a work order, for instance. It starts off as a problem report. After it is reviewed, it becomes a work order. That work order is then assigned to someone. Once the person assigned the work order has completed the task, the work order is marked completed. Once the supervisor is satisfied that a work order has been completed, it is closed.

In each phase of it's life cycle, the element responds differently to stimulus. While one *could* simply devise a grossly complicated all-seeing/all-dancing class that deals with all of the states through conditional statements... it's been my experience that these systems are a gory mess. Much better to define each state as a different class of element, and then use inheritance to keep the exploit the similarities.

To be a TDIF element, an object must respond to the following methods:

Method	Description
trashRecord	Signal to delete this node's data from the parent container on destruction of the object
carosel	Signal to the garbage collector to destroy the object on the next pass
renew	Upload the internal representation of the object to the container, and then **carosel**
NodeId	Return the 'primary key' of the element in the container
globalName	Return the fully qualifed name of this object: *containerHandle-nodeId*
Get *?field?*	Return the complete contents of the element as a key value list if not field is given OR Return the value of a specific property of the element
Set *keyValueList*	Update the internal representation of the object with the values in *keyValueList*
Input *keyValueList*	Validate/Format all of the values in a keyValueList (Does not change the internal state). Throws an error if a value is out of bounds. On success it returns a key/value list with values reformatted for the native store.
/property *propertyName*	Return an object handle for the **property** node that validates and represents the field **propertyName**. Note, this object will be the same whether you call /property from an **element** or it's **container**
/container	Return the object handle of this elements container

Table 4 – Basic Element Exposed Methods

Properties

Properties are the oddest part of the TDIF implementation. Early in the design, it became clear that it was useful to think of columns as objects unto themselves. Rather than store data, they were the voice of reason that understood all of the validation rules for the column. The column object knew what the max size for a field was. It would throw an error if we tried to store an integer in a string field, or vice verse. When I implemented an HTML display engine, the column object generated the individual user interface elements for each field, be it an input box, a dropdown menu, etc.

Because it is handy to have an **element** call for a **property**, we provide a mechanism for it to do so. Likewise, if we don't yet have an **element**, but we want to create a form to generate one, it becomes useful to call up a **property** from the **container**. Thus, both object types can call up **property** objects on demand.

While **properties** do not store element data, the *do* store their own information. Tidbits like the size and description for the field. This information is stored in the **connector** object. On nodes that are both the connector and the container, it's assumed the developer will be able to figure out which nodes are records and which are properties.

Containers have the option to generate all of their **properties** during *tdifAttach*, or to simply call them into being in response to a */property* call. Most elements redirect */property* calls to their container for this reason. **elements** are also welcome to generate their own properties. The key is that the object spelled out should be consistent whether the call is made to the container or the element, and that the the first thing the */property* method should do is check to see if a property exists before creating a new one.

Properties are not subject to the same garbage collection as elements. It is assumed that they will stick around until their container is destroyed.

Here is an example of property object usage. Again, this build on the objects we've created in previous examples.

```
# Spawn an object
set nameObj [$recordObj /property name]
$nameObj Display [$recordObj Get name]
Evil Twin Skippy
$nameObj htmlEntry [$recordObj Get name]
<input name=maildb.user.name value="Evil Twin Skippy" size=40>
$nameObj Input "An Increadibly Long String that is sure to exceed the size …"
An Increadibly Long String that is sure t
```

Example 5 – Basic property usage

Method	Description
element	Return the fully qualified path of this column (for use in forms and database statements)
Label	Return the description of the property
Display *value*	Render *value* into human readable form for the current display engine
Export *value*	Render *value* into human readable plain text
Input *value*	Check the validity of *value*, and convert an inputted value into native format of the storage engine
Entry *value*	Return a string, block, or script that defines a UI input form for this property
Search value	Return a string, block, or script that defines a UI search form for this property
Hidden *value*	Return a string, block, or script that defines a hidden UI element form for this property
Dump	Return the value entered into the form generated by Entry in a tk window
Update *value*	Update the display widget in tk window with the new *value*
Options	For boolean and select fields, return a list of valid options

Table 5 – Basic Property Exposed Methods

EXTENDING TDIF

The reference implementation for TDIF is written in my own object system, Tao. Tao currently requires either Tcl 8.5, or Tcl 8.4 with either the Dict or sqlite package. There is no earth shattering technology in Tao that would prevent TDIF from being re-written in any other object system.

TDIF also does not care what object system a node (be it a connector, container, element, or property) is written in. It will be perfectly happy passing data back and forth through an incr Tcl, Tao, snit, or plain old namespace. Simple implement the method as outlined in the usage section.

Writing in Tao would only be required to extend or enhance the existing library of connectors, containers, elements, and properties.
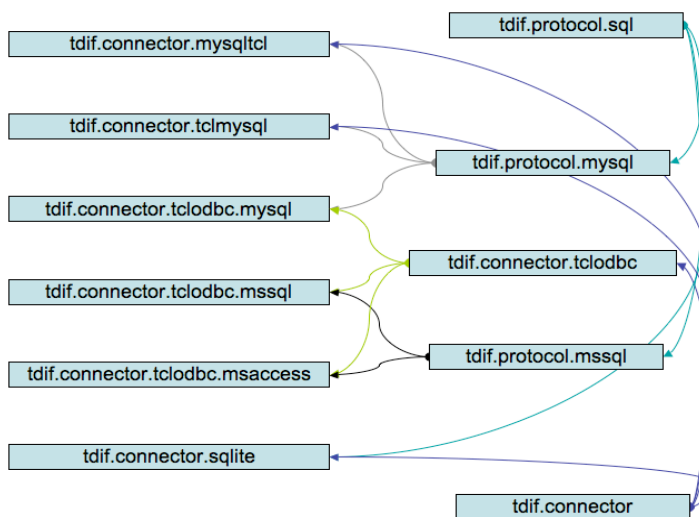
connectors

Connectors written in Tao have two parts:

➢ Data Language Interface – Translates between the TDIF and the specific SQL dialect of the database server.

➢ Data Driver Interface – Translates between TDIF and the Tcl interface of the database connection driver.

Why two? Basically the dividing line between the Protocol and the Connector is pretty blurry. To the right is a simplified relationship between the connectofs in TDIF.

TclOdbc, TclMysql and MySqlTcl all have a radically different connection driver. They are a raw pipe into MySql, though.

TclODBC doesn't just talk to MySql. It can talk to everything from Microsoft SQL, to access, to .txt files. It can also talk to PostGres, MySql, BerleyDBs, and so on. While the driver is identical, the language and features available depend on what type of database you are connecting to.

Then you have Sqlite, which is it's own language, its own driver, and it's pointless to try to separate the two. In other cases, our "driver" is really a raw socket talking to a relay. Once you abstract, it's really hard to think of connectors as monolithic things anymore.

Because we have no control over the data languages, and it will take some years for the driver writers to catch up with a standard interface, we need to accept their diversity as a fact of life. Each of system has its own capabilities, hindrances, and design panache. Sometimes a function a function we are looking for is in the language.

And then we have LDAP. Which I briefly tried to integrate, but frankly don't understand enough about to sensibly formulate policy. It did send me down the path of abstracting out database command statements though.

Writing connectors

Essentially a connector needs to implement every method that a container object would need to call to access data. This includes abstracting out common queries and language constructs. Why? Well

some language provides a very nice way to INSERT OR REPLACE. Which saves a lot of work on updates, since we don't have to check for the existence of a record. On others, we check first, if a record exists we INSERT, otherwise we UPDATE. And, depending on your SQL dialect, INSERT and UPDATE have very different syntaxes.

In other cases we have features like full-text searching. It's nice if you have it. But many database systems don't. So we have to fake it. I've prepared a table of all of the methods that my Containers call and my Connectors implement. There will undoubtedly be more as we move along.

Method	Description
stmt_insert {keylist valuelist dtable}	Return an insert statment
stmt_exists {keylist dtable}	Return a 1 if the record in keylist exists in table, 0 otherwise
stmt_replace {keylist valuelist dtable}	Formulate an INSERT OR REPLACE where it's available, Otherwise call stmt_exists and return an INSERT statement or an UPDATE statement as appropriate
stmt_select {keylist fieldlist dtable}	Return a select statement
stmt_update {keylist valuelist dtable}	Return an update statement
stmt_where {keylist {forbid_null 0}}	Return a where statement, optionally yell if we try to enter a null value
stmt_delete {keylist dtable}	Return a DELETE statement
table_qualify {rawtable}	Convert a **db.table** to just **table** on the braindead systems that don't handle it properly. In the systems that DO handle **db.table**, make sure we add **db** if we just give **table**. Essentially we always want to call a table by the same name
column_qualify {rawcolumn {table {}}}	A similar process to table_qualify for columns. On systems with no concept of alternate databases, make the reference: **table.column**. On those that do, make it **db.table.column**. And by the way, make it relative to *table*
table_exists table	Return 1 if a table exists, 0 if it doesn't. And use the best mechanism you have.
searchStmtPrim {var op val}	Return a comparison statement in the native form for **op** the database. Valid **op**s in TDIF are: match notlike isnot = > < Also convert **null** and **now** in the val field to either a native representation, or some tcl calculated value. Not too useful for SQL, but essential when we go to add non-sql data languages
searchStmtJoin {joinop stmtl}	Connect several conditions created by searchStmtPrim into a compound conditional. Again, SQL we all know how to do. This is primarily for other languages.
searchStmtNest stmtl	Nest a compound statement into a single conditional. (In SQL we use parentheses. Other systems do it differently)
table_create {table schemaDict}	Create a table, if it doesn't already exist, to the spec defined in schemaDict.
stmt_create_column {field infoDict}	Return a column creation statement for a column named *field* using the information in *infoDict* as a guide.
stmt_create_index {idxtable idxname infoDict}	Return a statement that will create and index of *idxname* on *idxtable* according to the specs in *infoDict*
stmt_create_table {table columns indexes {table_type {}}}	Generate a statement that will create a table with the columns defined by the **columns** dict, the **indexes** dict, and of type **table_type**. At present, *temporary* is the only alternate type TDIF understands.
searchFullText {value {columns {}}}	Generate a fulltext search statement. If fulltext is not available, fake it.

TABLE 6 - CONNECTOR LANGUAGE ABSTRACTION METHODS

WRITING CONTAINERS

Containers would seem to be the hard part. There are so many, with so many different shapes. From a TDIF perspective, though, they are quite simple. And if you looked at the section on writing connectors, you'll see that most of the heavy lifting has actually been abstracted out to the connector.

Sql connectors are available with the stock release of TDIF, and all that is really required to customize them is to supply the proper schema. In most cases, I either dump data from the connector

object, or I provide a hard-coded dict to return through the schema method. There really isn't any magic going on behind the scenes.

A default container stores information as an in-memory dict. SQL containers redirect the nodeGet and nodeSet commands to pull data from the connector instead. And even there, they don't build sql statements. They farm that out to the connector.

The only trick is keeping track of what you spawn off in the way of **element** and **property** nodes, so you remember to destroy them when you destroy the container. Now containers *do* get to be monstrously complex. But the complexity is all driven by the application. The needs of TDIF are simple and few.

WRITING ELEMENTS

Elements are only as complicated as your application. Beyond the basic methods outlined in the usage chapter, there are not magic interactions. You only need to be aware that garbage collection does take place, and *elements* are not permanent

WRITING PROPERTIES

Like elements, the methods that TDIF provides are minimal for inter-operability. How they are used, and what additional methods you chose to implement are your own.

WRITING IN DIFFERENT OBJECT SYSTEMS

TDIF is blind to what object system you are actually using, assuming you are using an object system at all. TDIF does make a few small requests if you do decide to strike out on your own and write objects that interact with TDIF in a system other than Tao:

Pay the Ferryman

If you write elements in your own object system, implement a "**scythe**" method for the garbage collector to call. **scythe** should return a procname to evaluate which will destroy your object. The garbage collector calls: "[$object scythe] $object" internally. If your object system has an ensemble command "object destroy $object", just define a proc that takes care of the details. For instance:

```
proc itclScythe objname {
   ::itcl::delete object $objname
}
```
Example 6 – Scythe Implementation for [[ncr Tcl]

When a temporary object (especially elements) come into being, register with the garbage collector *thatanos*. All that's required to add an object to it's accounting is a call to *thanatos alloc $object*. Once thanatos has it allocated, it will delete temporary objects whenever you call: *thanatos cleanup*.

By convention, if you want to prevent the garbage collector from destroying an object, return an empty list when **scythe** is called.

For web-engines, I call thanatos cleanup at the end of every page view. TK based systems should probably call it after a data entry screen closes. Thanatos will only delete objects that have been "kissed." Tao calls *thanatos kiss $object* at the end of the *carosel* method.

Thantos also destroys objects if they have been idle for 60 seconds. To mark an object as not idle, call *thantos touch $object*. This will grant it another 60 second lease on life. To change the lease length for all objects, set a new value for ::thanatos::kill_time. Kill_time takes integer values, in seconds.

The complete interface to *thanatos* is:

Method	Description
alloc *object*	Add an object to automatic garbage collection
free *object*	Remove an object from automatic garbage collection
touch *object*	Extend an object's lease on life
kiss *object*	Trigger an object's destruction on the next call to *cleanup*
knock	Check to see if any objects are set to be destroyed. (If zero, cleanup is not called)
cleanup	Run through and dispatch any object that has been *kissed* or whose lease has expired

TABLE 7 – GARBAGE COLLECTOR INTERFACE

Thanatos expects to see the following methods in any object it interacts with:

Method	Description
scythe	Proc to call to destroy an object

TABLE 8 – METHODS TO EXPOSE FOR GARBAGE COLLECTOR

I would also like to point out that "thanatos kiss" should not be called during an object's destructor. Odd's are if the destructor has been called, the object has already been kissed. What you may want to do is "thantos free" the object. This will remove if from garbage collection no matter if it self destructs or is removed through another mechanism.

Stay with the times

TDIF includes a built in schedular object *chronos*. Chronos is called periodically to run tasks. Rather than invent your own periodic task use TDIF's. Chronos requires no exposed methods in objects.

The MySql connector objects presented a troublesome problem for me because the idle/reconnect feature in the drivers is broken, wounded, or missing. So I had to implement my own in script, thus the genesis of Chronos. To use chronos:

```
set jobId [::chronos::JobCreate name "Test Job  " \
                        interval 60 script "puts {Hi There}"]
Hi There
Hi There
….
  ::chronos::JobKill $jobId
```

EXAMPLE 7 – CRONOS USAGE

Format for Schema Dicts

Containers report their schema as a dict. The table_create function in the connector objects ALSO takes data in as a dict. What is the format for this dict? The dict has four main entries: **column, index, primary_key,** and **type**.

```
staffObj schema
primary_key uid
column {
   uid {type intkey}
   username {type char length 32 required 1}
   name {type string desc {Full Name}}
   name_last {type string desc {First Name}}
   name_first {type string desc {Last Name}}
   mail {type char length 64 required 1 desc {Email Address}}
   active {type boolean options {1 0} states {0 no 1 yes} default 1 \
      desc   {Display on Stafflist}}
   password {type password crypt sha1 length 64 sqltype char \
      desc {Password}}
}
index {
   uuname {type unique columns username}
   uemail {type unique columns email}
   maildx {type index columns {
      uid
      {username direction ascending}
      {email length 16 direction ascending}
   }
   nsearch {type fulltext columns {name name_last name_first}}
}
```

Example 8 – schema dict

Column

The column dict lists each column in a table in the format: *columnName {key value …}*. This dict is used to generate **property** nodes. The table of values used by TDIF are as follows:

Property	Description
type	What type of property (see table)
sqltype	Native storage type (optional)
desc	Property/Column description
length	Length of the field (longer values are truncated on Input)
width	Display size of the field
default	The default value for the column
options	For enum and select types, the range of valid values
states	Mapping of stored values to human readable values
required	Allow null values
collate	Colation to use on column. Valid values: binary, nocase

Table 9 – TDIF Column properties

TDIF Type	MySql Representation	MSSql Representation	Sqlite Representation	Description
string	tinytext	char(*length*)	string	Generic String
int, integer	int	int	int	Integer
intkey	…	…	…	Auto Incrementing Primary Key (implementations differ between data engines)
char, varchar	varchar(*length*)	char(*length*)	string	String of fixed width. Note that the **collate** option will select char or varchar in MySql to emulate the collate functionally in sqlite.
text	bigtext	text	string	Large block of text
select	enum(*options*)	char(*length*)	string	Enumerated list of values
boolean	tinyint	tinyint	int	1/0 or Y/N or True/False value
Table 10 – TDIF Column Types				

Index

The index portion simply lists all of the indexes, by name, as well as a type and columns affected.

Field	Description
type	Index Type: index, unique, fulltext
storage	Native storage engine to use (if supported, ignored otherwise). In MySql available values are: BTREE and HASH
columns	Dict describing the columns indexed
Table 11 – TDIF Index properties	

Field	Description
length	Integer, length of the field to index
direction	Sort direction to index by this column. Valid values: asc, ascending, desc, descending
collate	Colation to use (if supported by database, ignored otherwise) For sytems
Table 12 – TDIF Index column properties	

TDIF Type	MySql Representation	MSSql Representation	Sqlite Representation	Description
index	index	index	index	Generic index
unique	unique index	unique index	unique index	Uniquely constrained index
fulltext	fulltext index	*emulated*	*emulated*	Fulltext search index
Table 13 – TDIF Index Types				

CONCLUDING REMARKS

TDIF is a work in progress. My intent is to at least start a conversation on how we in the Tcl community can begin adopting a formal way of interfacing with external data. Why is such a framework desirable?

Newcomer Friendly

For starters, it makes the system friendlier to newcomers. I won't pitch this as a major reason for adoption, but it is at least a nice side effect. Few can argue though that providing a consistent interface for all database connections will make documenting simpler.

Interoperability

A stronger reason than consistency for consistency's sake is that software developed for one system can be more easily integrated into another if they can at least agree on how to access the database. By abstracting most of the SQL statement generation, one also allows software originally developed for one environment to be turned around and used in another with little to any modification.

Optimization

We are not all SQL experts. By abstracting out common functions to a system that can be customized for the individual database engine, we can exploit performance enhancing tricks. Take for instance using the "INSERT OR REPLACE" statement in Sqlite, instead of the conventional two step process required for other databases.

Safety

Abstracting out SQL statement generation also allows us to catch common mistakes. Say a "DELETE FROM TABLE" or "UPDATE" with no WHERE condition. It also ensures that all of the data from Tcl is escaped properly, and won't cause a bizarre interaction if someone manages to input a value that just so happens to be the name of a column, and they JUST so happen to have not put the value in quotes.

It doesn't just work for SQL

By abstracting the concept of tables out to containers, we open ourselves to an entire world of data storage systems. We can write a handler that wraps around a flat file. We can transparently relay database calls to another machine.

Code can work in multiple operating environments

A case in point, I have libraries of scripts the run from a shell environment on various server as well as withing my tclhttpd base intranet. Depending on the computer, it may or may not have access to the mysqltcl package. If it does not, I create an object, with the same name, that takes all of the calls that *would* have gone to the mysqltcl connector, and instead relays them to a helper daemon running on the database server.

My software doesn't know, or care, which is which.

For all of these reasons, and more, I hope that TDIF will spark a conversation, if not a change in thinking, on how we treat outside data. Feel free to contact me with suggestions on how TDIF can be improved.

You can download a reference implementation of TDIF, as well as Tao (the object system it is written in) at:

http://www.etoyoc.com/tao

FURTHER READING

"The Tcl Architecture of Objects", Sean Woods: http://www.etoyoc.com/tao/

ABOUT THE AUTHOR

Sean Deely Woods is the Senior Network Engineer at the Franklin Institute Science Museum in Philadelphia, PA. He has been writing in Tcl professionally since 1997. His claims to fame include a serial port driver under MacOS classic, and the famous "Hypnotoad" presentation at the 13th tcl conference.

Sean can be reached via email at: yoda@etoyoc.com

He maintains an independent website at: http://www.etoyoc.com/

He can be found on the Tclers chat under the pseudonym **hypnotoad**

Tclers Wiki entries by him are normally signed **SDW**

CREDITS:

Graphics and Art

Cover Graphic, "Game Console", Username MachZero, ConceptArt.org: http://www.medievalfx.com/machzero/graphics/050420a.jpg

Fonts Used:

Text: Eurostile
Title: MAINFRAME
Headings: MAINFRAME, METRODF, Monoglyceride
Captions: Aerovias Brasil NF
Code: Monoco CE