The Sobyk Binary Distribution

Presented at
The 26th Annual Annual Tcl Developer's Conference
(Tcl'2019)
Houston, TX
November 4-8, 2019

Abstract

ActiveState has stepped out of the business of actively developing Tcl. The side effect of their absence is that the core team no longer has a reference binary distribution where the impact of new developments in the core can be measured against existing applications and production software.

Existing kit architectures do a fairly decent job of keeping up to date with released packages and integrating them into a released core. The process requires constructing a lot of boilerplate code. Some is shell script. Some is C code. And the process is made difficult by a profound lack of integration
standards when it comes to package development for Tcl/Tk. This "boiler plate" actually ends up including a lot of institutional knowledge about the quirks of each package's library requirements (often varying by platform), build system limitations, and installer quirks. All of this was information that was tirelessly gathered by the staff at Active State.

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Introduction

Tcl is really two different communities trying to live under one tent. One group are the brilliant, talented, hard working core developers and want to advance Tcl in 9 directions at once. The other is the brilliant, talented, and hard working users who use Tcl to advance their own projects in 9 directions at once.

In between used to be ActiveState. They would take what the hardcore developers made and beat it into a shape the hardcore users could use. They did a brilliant enough job that both camps in our community didn't realize how important ActiveState's role was until they ceased to perform it.

2018-2019 war our first release (8.6.9) without ActiveState. And it was rough. I should know, I manage to wear both a hardcore Tcl Core Developer AND a hardcore Tcl Application Developer hat.

I'm the author of Tip430 (adding Zipfs to the core). I'm a maintainer of Tcllib. I'm one of the poor schleps who maintains the Tcl Extension Architecture's tcl.m4 macro.

I am also the lead developer for a tiny little company that uses Tcl to perform very advanced Naval Simulations. And in my non-existent spare time I'm trying to write a Tcl based game engine.

In this past year I have yelled, and been yelled at. My implementation of ZipFS while I did my best, had limitations that needed to be addressed on contact with real applications, and I heard about it. Multiple issues in 8.6.9 broke the application I develop at work, and I made sure I was heard. We've also had a major rumble over in Tcllib as two of the maintainers (and I was one of them) found ourselves on opposite sides of a definition of what "working code" meant. And the scars of those conversations reverberate to this day.

Yelling is what happens when talking stops working. And in every case yelling involved someone who had a legitimate need that was not being addressed by developer who was suddenly blindsided by a requirement they never knew existed.

How IT People Communicate

We, as IT people, don't tend to communicate complex ideas in human terms. The reason is that human language is inadaquate at describing complex technical ideas. We aren't anti-social, we aren't ineffective communicators, there is simply no decent way in prose to more precisely describe that 1+1=2.[1]

As such there is a certain amount of passing around a finished implementation, and allowing a fellow developer to "grok" it that must take place for an idea to be communicated. But this process tends to focus on technical issues of making the implementation function from a logic and syntax perspective. Problems that can't be communicated in code tend to be drowned out.

"*This application that I pay my mortgage with is crashing*" from a user isn't as compelling as a regression test on one's own machine that proves otherwise. "*This change to the core is causing an extension I depend on to stop compiling*" is not as compelling as how that how much simpler that chunk of the core is now that it doesn't need to deal with that ill-designed API.

I have found the only way to prove that a change is causing a problem to the community at large involves having them replicate the error. And as my problems tend to be pretty deep,

---

[1] And if the 360 pages of *Principia Mathematica* by Russell and Whitehead is any guide, ground-up discussions can be a tad long winded as well.

replicating that error in a particular environment. And because the other party's time is precious[2], providing a kit that allows them to replicate the environment on their machine.

That got me to thinking, what if we had a community resource which would allow developers to re-create an entire binary distribution of Tcl on their own machine?

Sobyk

Sobyk is a binary distribution for Tcl. It builds viable Tcl based applications for the Mac and Windows platforms. Two environments that are generally hostile to creativity, yet inexplicable popular with end users. Where possible, extensions are statically compiled for the platform. It utilizes a Tcl based build system to allow the same automation to work, unmodified, on platforms that either lack unix sh, or where the file path management for their sh is dodgy at best[3].

Sobyk can build an in-house application. It build a retail software title, such as a game or App store utility. The ultimate user of the code probably has no idea what Tcl is, or that their application is written in it.

The idea is that whoever is assembling the applications is, on some level, a Tcl developer. They probably have both Tcl and C code that need to be integrated. While talented and intelligent, they do need help with packaging, because binary Tcl extensions are utterly inconsistent in build system fulfillment. They also have users with their own crazy requirements that will likely involve custom software.

Sobyk works directly from Fossil and Git checkouts, and only resorts to Tarballs for non-tcl external dependencies. Sobyk can be given directions for how to work with code that hasn't been manually prepped for release. (Important for experimental builds of the Tcl core.) And wherever possible it keeps all of its markup and automation under one set of rules, and in a language that a Tcl developer of a Tcl based application are likely to know already: Tcl.

The name Sobyk was invented by an online name generator for fantasy games. It was selected for it's aesthetics from a pool of a few hundred candidates. I chose a random name because it means nothing. There is no Acronym. There is no mythical figure[4]. There is no computer science term. There is only Sobyk.

A distinct name is important, because names tend to get stamped all over a project. And the intended use for this project is to be cargo-culted and used to make other people's projects. A global search and replace can find "Sobyk" and replace it with "Wayland Yutani Power Shell", or "Throatwarbler Mangrove", without fear of accidentally renaming references to an external C function, or introducing comical typos in the documentation.

Core Team (Non)Involvement

The concept is that Sobyk will not be an officially sanctioned project on the part of the Tcl Core Team (TCT.) Instead, Sobyk is a fictitious "customer" who takes what the TCT releases, and figures out how to make those releases into viable products. How to adapt/update/enthrall packages into melding with the core to become finished products.

---

[2] My time is apparently free, however. 🙄

[3] (cough) Microsoft Windows.

[4] Well, there was an Egyptian God named Sobek, at least according to: https://en.wikipedia.org/wiki/Sobek. But I only discovered that after putting the string into a search engine to make sure it wasn't a sex act on Urban Dictionary or a swear word in a foreign language.

The distinct name gives a bit of "Plausible Deniability" to the core team. This allows Sobyk to make policies and design decisions independent of the TIP process, divorced from the official Tcl release cycle, and be answerable only to its user base.

Sobyk Basics

Sobyk is a fossil repository, which unpacks a build system I first described in *Practcl: A Tcl-Based Build Automation Tool for C Extensions*[5]. The concept of Practcl is the core of all of the tools needed to assemble a working kit are distributed as module in a single Tcl script file. What capabilities are not present in the library itself, or the Tcl core that is interpreting it, the library contains code to download and install.

The instructions to build a sobyk kit are thus (assuming you have the source code unpacked):

```
tclsh make.tcl all
```

That's it. The make.tcl script will go through the motions of cloning the SCM repositories of the core as well as libraries and extensions the kit will use. The script unpacks them, configures them, compiles them, and integrates them into a statically linked, self-contained executable.

Two executables, actually. One *sobyk* is a Tcl shell with all of the extensions that do not require Tk. *wishkit* is all of the products included in *sobyk*, plus Tk and extensions that rely on Tk.

Walkthrough

Now, I realize a one line script to generate a Tclkit is nothing new. So, let us review what ended up happening when we ran that script. Let us back up our example and go with a file scheme I actually use in practice:

```
mkdir -p ~/build/myproject/sobyk                        # Build our directory Structure
mkdir -p ~/build/myproject/build-macos
cd ~/build/myproject/sobyk
fossil open ~/.local/downloads/fossil/sobyk.fossil      # Unpack the sobyk source code
cd ~/build/myproject/build-macos                        # Enter the directory for the build
tclsh ../sobyk/make.tcl all                             # Run make.tcl from sobyk in our build directory
```

The make.tcl file assumes that whatever directory it is being run at is the directory you want the build product to go. (Much like ./configure.) After running you will see the following file system emerge under ~/build/myproject/sobyk

pkg/	Toplevel folder where binaries are built. Every folder is a different core component or project
PKGROOT/	Install location where non-Practcl projects deposit their contributions for the Kit's virtual file system
objs/	Folder where Practcl will deposit .o/.obj files generated during compilation.
build/	Directory where staged builds deposit products, and dynamically generated C and Tcl files are deposited.
password.txt	Cleartext password for cryptographically enciphering proprietary code through helper scripts
make.tcl	Script redirect to invoke the make.tcl to wherever the make.tcl file that constructed this directory actually resides in.

[5] Available Online: https://tcl.tk/community/tcl2016/assets/talk42/practcl-paper-v1.0.1.pdf

sobyk_bare	Tclkit with all of the binary components for Sobyk, but without the attached VFS
sobyk.vfs	The virtual file system for Sobyk's executable
sobyk	The completed Sobyk executable with Zip based VFS appended
wishkit_bare	Wishkit with all binary components for Sobyk, Tk, and additional Tk extensions
wishkit.vfs	The virtual file system for Wishkit's executable
wishkit	The completed Wishkit executable with Zip based VFS appended

Customizing Sobyk

Sobyk's build system is an object model. Object can contain other objects, and at all steps along the way object can spawn new objects.

Our recipe for a Tk-less kit contains segments like this:

```
my add_project tcl {
  class subproject.core
  name tcl
  tag release
  static 1
  fossil_url http://fossil.etoyoc.com/fossil/tcl
}
```

This snippet adds a new object to the kit "tcl" which represents the Tcl core. That description includes a class of behaviors, an SCM tag, and a URL to clone the SCM from. Sobyk understands fossil and git at present, so a Git-based project looks like:

```
my add_project rl_json {
  class subproject.binary
  git_url https://github.com/RubyLane/rl_json
  tag master
  install static
}
```

The fact the SCM is Git instead of Fossil is denoted by populating the git_url field instead of the fossil_url field. There is also a file_url for tarball/zip based distributions as well. A project can have multiple distribution types, and if multiples types are given, Practcl will try each in the following order: fossil, git, file.

Some sub-projects require customized behavior, and Practcl allows for existing methods to be replaced or new methods added by adding an extra argument:

```
my add_project tls {
  class subproject.binary
  fossil_url https://core.tcl.tk/tcltls
  tag trunk
  install static
  initfunc Tls_Init
  libfile tcltls.a
  pkg_name tls
} {
  # The tls package is a little weird, so we
  # have to be a little more explicit
  # about linker products because the
  # Makefile doesn't give us the normal hints
  # that a TEA makefile would
  method linker-products {configdict} {
    set srcdir [my define get builddir]
    set dat [::practcl::read_sh_file [file join $srcdir Makefile]]
    return " [dict get $dat LIBS] [file join $srcdir tcltls.a]"
  }
}
```

In this case, we treat TLS as a TEA extension, but because it's not really a TEA extension, we require a little magic to link it properly. So we replace the method that issues the linking instructions. You can also see we feed specifics about this project's init function and project name are populated in the configuration dictionary.

The "behavior customization" script is actually a call to `oo::objdefine`. Any statement valid in an `oo::objdefine` block is valid in an object behavior script. Thus, we get an easy to explain way of injecting custom code that can teach a novice Tcl a well documented trick from TclOO rather than attempt to wrap things in a domain specific language.

Sobyk attempts at all times to use facilities in Tcl wherever possible. You'll note the use of "my". `add_project` is a method of the object being manipulated to generate a child object. This script is invoked inside the namespace of the object we are spawning objects into. The method also works outside of the object's namespace. The objects produced are given names that are local to the parent object, and to look them up we provide linking and introspection tools. Which I will describe in a make file section of this paper.

Creating a Make.tcl File

Sobyk make.tcl files are Tcl scripts. While it has facilities for performing Make style directed graph dependency fulfillment, that is not its normal mode of operation. Make accepts a number of commands with arguments, and those commands and arguments are designed to perform the day-to-day interactions expected of a project that needs to compile, link, and sometimes even take control of one or more other projects.

For backward compatibility with TEA, Practcl extensions have a Makefile which translates "make all ; make install" into the equivilent invocation of the make.tcl file. The Make.tcl file in turn has facilities to do things that are, while not impossible, downright annoying to try to do with an existing makefile:

- Perform either a static or shared build on command
- Install the project to a specified location
- Perform and SCM update and recompile as needed. Or switch to a different SCM tag.
- Generate a zip file for distribution in teacup
- Install only one or more modules from a pure-tcl library
- Have autoconf rebuild the ./configure script for the package
- Re-Run ./configure with new flags

All of these actions can involved passing arguments, and arguments are awkward to pass to Make.

Each Makefile is a hybrid of standard Tcl script accepting arguments from the command line, and a dependency graph driven build fulfillment system. Each verb that is intended to take arguments must be given as the first argument. If the verb is not a special keyword, it is treated as a trigger for a dependency.

The first part of the Makefile is creating objects that represent the ultimate build products for our project:

```
set dat [::practcl::read_configuration $CWD]
::practcl::tclkit  create SOBYK $dat
SOBYK  define set name sobyk
SOBYK  source [file join $::SRCDIR tclkit.ini]
```

In this snippet, we use the `::practcl::read_configuration` command to sniff the build directory for autoconf/autosetup data, and return the pertinent bits as a dictionary. That dictionary is passed to the new object *SOBYK* who will represent the kit we ultimately wish to build. We also tell *SOBYK* to read additional instructions from a file. (The contents of which we discussed in the *Customizing Sobyk* section.)

Next, we build our dependency tree:

```
::practcl::target tcltk {
  depends {deps configure clean local-env}
  triggers {script-packages script-pkgindex}
  filename [file join $CWD config.tcl]
}
::practcl::target tclkit {
  aliases {example sobyk}
  depends {deps tcltk packages practcl}
  object  SOBYK
}
```

All of the targets in the dependency tree have a configuration dictionary. Several of these keys have special meaning:

aliases	Alternate names this trigger will answer to
depends	Indicates a target requires the fulfillment of another target
filename	The name of a file this target produces. Used as a check to see if it has been fulfilled
object	Used to associate a Practcl object with a trigger.
triggers	Indicates that the execution of this target should also activate other targets

Next, we process command line arguments:

```
switch [lindex $argv 0] {
  all {  ::practcl::trigger tclkit }
  default {
    ::practcl::trigger {*}$argv
  }
}
```

The ::practcl::trigger command walks the dependency graph and populates the global array make() with a boolean flag indicating which triggers have been activated, and which have not.

We can then test what portions of our build process need to be run, and because this is a Tcl script, we can mandate the order of events:

```
if {$make(clean)} {
  # Actions to cleanup build products
}

if {$make(tclkit)} {
  # Actions to compile, link, and wrap tclkit
}
```

The concept is that our script is a pipeline, and thus one trigger can have effects in several stages of the process.

Individual Practcl objects also have their own dependency trees. In fact, the `::practcl::trigger` command is just a convenience wrapper around a global object `::practcl::LOCAL.`

```
proc ::practcl::trigger {args} {
  ::practcl::LOCAL make trigger {*}$args
 foreach {name obj} [::practcl::LOCAL make objects] {
```

```
    set ::make($name) [$obj do]
  }
}
proc ::practcl::depends {args} {
  ::practcl::LOCAL make depends {*}$args
}
proc ::practcl::target {name info {action {}}} {
  set obj [::practcl::LOCAL make task $name $info $action]
  set ::make($name) 0
  set filename [$obj define get filename]
  if {$filename ne {}} {
    set ::target($name) $filename
  }
}
```

My early efforts with Practcl were heavily influence by smake[6]. And while I realize a global array with a really common name is not a great design paradigm in general, it does feel kind of right for small projects with a single deliverable. However, of your project already makes use of the *make* variable, and or you hate that sort of shortcut with the fire of 1000 suns in principle, Practcl is perfectly happy NOT operating in this mode. As you can see, `practcl::trigger`, `::practcl::depends`, and `::practcl::target` are all sugar coating interactions with the `::practcl::LOCAL` object's **make** method ensemble We could easily rewrite those sections of our make.tcl file without the global array:

```
::practcl::LOCAL make task tcltk {
  depends {deps configure clean local-env}
  triggers {script-packages script-pkgindex}
  filename [file join $CWD config.tcl]
}
::practcl::LOCAL make task tclkit {
  aliases {example sobyk}
  depends {deps tcltk packages practcl}
  object  SOBYK
}
switch [lindex $argv 0] {
  all {  ::practcl::LOCAL make trigger tclkit }
  default {
    ::practcl::LOCAL make trigger {*}$argv
  }
}
set makeobjs [::practcl::LOCAL make objects]
if {[[dict get $makeobj clean] do]} {
  # Perform clean actions
}
if {[[dict get $makeobj tclkit] do]} {
  # Actions to compile, link, and wrap tclkit
}
```

With kits, it is also useful to place make triggers on the Tclkit object itself. If our project supports two different kits. The *make* ensemble would work just as easily on our *sobyk* an *wishkit* objects as they would on *::practcl::LOCAL*. The advantage of *::practcl::LOCAL* is that it is guaranteed to exist as soon as practcl package is loaded.

Cross Compile Support

Sobyk looks in the build directory for a file named config.site. The contents of this file are identical to any other MinGW cross compile, and contain a series of environmental variables pointing the build system to the proper compiler, linker, etc, as well as flags to feed to each. Sobyk only supports one build target per build directory, but you can have any number of build directories open in what I call "the sandbox."

---

[6] Smake is a pure-tcl directed graph style build system: http://people.fishpool.fi/~setok/proj/smake/

Managing Multiple Projects

My build folder in my home directory looks a bit like this:

irm-trunk/build-macos	The folder where the MACOS build for IRM will be built. Because I am compiling on the mac, no config.site
irm-trunk/build-win32	Build folder for Win32 release. Contains a config.site with the compiler and flags for 32 bit windows
irm-trunk/build-win64	Build folder for the Win64 release. Contains a config.site with the compiler and flags for 64 bit windows.
irm-trunk/irm	The Fossil checkout for IRM
irm-trunk/tcl	Fossil checkout of the Tcl core for this project
irm-trunk/tk	Fossil checkout of the Tk core for this project
irm-trunk/*	Fossil/Git checkouts of all of the packages and libraries IRM depends on
sobyk/	Completely parallel file system to support Sobyk builds
irm-fsar/	A different branch if the IRM codebase to irm-trunk, which utilizes a different versions of the core and several projects

A core developer who wished to maintain a sobyk build for multiple versions of Tcl would probably have:

core-tcl-8-5-release	Whatever the most up-to-date version of 8.5 is at that point.
core-tcl-8-6-release	The last release version
active-tcl-8-6-6	A profile mimicking ActiveTcl's core and popular packages
core-tcl-8-7	A semi-up-to-date 8.7
core-tcl-trunk	Pull of everything currently checked into trunk
core-tcl-tipXXX	A checkout of a particular tip this developer is either creating or evaluating, along with its associated packages

The nice thing about Sobyk is that each of these profiles is maintained in its own file system. And the profile not only includes a specific Tcl/Tk core, but it also includes the proper version for each Tcl/Tk package that supports that core. Those can be core supported packages, external packages, little creature comforts that provide toys that suite the developer's fancy, etc.

Package Marketplace

Currently the Sobyk distribution only includes a handful of the most commonly used Tcl/Tk extensions. Moving forward, we need a place for developers of extensions to be able to keep users informed of bug fixes, new versions, and new extensions. We also need a develop a mechanism for identifying orphaned packages, and directing volunteer efforts to maintaining those packages. This marketplace is not about binary builds, it is about managing the build process. This site needs to capture the vitality of a package, any workarounds required for certain platforms, and up-to-date build instructions for every profile the Sobyk supports.

I will be soliciting ideas at the US Conference this year on what this website should actually look like, as well as what controls should be in place to ensure quality while at the same time allowing the marketplace to grow and evolve in a timely manner. For the interim, I will be creating a separate fossil repository titled "sobyk-marketplace" which will be centrally controlled by myself and my appointed deputies. That repository will contain a branch for every profile that sobyk supports. The file system for every branch will contain on file per package with a machine readable description of how Sobyk can download and integrate that package into a kit. That file will also include a machine readable description of all of the metadata and provenance that a human developer may wish to introspect about the project.

The Sobyk Bazaar starts with the concepts I introduced in the *shed* system[7], which in turn built on the foundation of the *Franklin Artifacts Database*[8]. In this database, while there are nodes and links, the focus of the system is on tracking history. Events are the only thing that has a definite record. Nodes are created to interpret the records, and links between nodes are considered somewhat ephemeral. Nodes can change types over the course of events, and links can be created or destroyed by events.

One's "snapshot" of the database is a replay of the entire journal which stopped where you either ran out of history, or elected to stop parsing. Each event spawns, destroys, intertwines, or splits apart nodes in the imagination of the reader. While recreating historical events until the present will require an expert and a few eyewitnesses, with luck moving forward this database will serve not only as a package database, but also as a historical research tool.

There is also no single authoritative source for events. As such, the database includes a measure of skepticism and uncertainty for every event and the facts to be learned from them. A certain source can be considered more trustworthy than others (and even so, only on certain matters), but there is no designated authority save the user him or her self. Sobyk, as a distribution, is a means for the user to outsource the decision on what to trust and what not to trust. Users can also elect to accept its judgement on select matters, but apply their own knowledge in specific areas. A cooperative/competitive project is perfectly free to take the same data source, and using their own sense of trust, develop different conclusions.

Projects have life cycles, and the events recorded in the Sobyk Bazaar are structured to capture those. Here is a summary of events captured:

---

[7] See: *Introducing TOOL The Tcl Object Oriented Library*, https://tcl.tk/community/tcl2015/assets/talk6/Tool_paper.pdf

[8] See: http://www.benfranklin300.org/frankliniana/tos.php

Project Events

Event	Data Captured	Description
project	Date, Developer, Progenitor, Aegis, name	Date project first created. If this is a fork of another project, we add the field *progenitor*. If this project is a standalone effort under at the auspices of a larger project (for instance a Tcl Improvement Project (TIP)), we add the *aegis* field.
alternate	Date, Project	Lists an alternate implementation (not a fork) that performs a similar role.
cancel	Date, Event	Acts as a catch-all means of negating the effect or a previous event.
distribution	Date, Developer, URL, Profile, Type, Version	A record that a distribution of the released form of this project is available from a person or organization, and is available for download at a stated URL
fission	Date, Project, Developer, Name	Date when a modules of a project fissions off to form a new project in its own right
fork	Date, Project, Developer, Name	Date when a different developer put forward an alternative version of a project. This new project will be
fusion	Project, Date	Date when a formerly distinct project can be considered a component of this project
join	Developer, Role, Date	Date when a new developer assumes a new role in the project
merge	Project, Date	Date when this project merged with another project that was previously considered a fork, or, if developing is being taken over by a larger project (for instance an extension becomes part of the core)
orphaned	Date	Date at which the last developer has given up the last role for this project. (This marks the project as up for grabs for any interested party to take on development or maintenance.)
release	Developer, Date, Version	Date of a public release, and which person or organization released it
rename	Date, name	Date when a project changes names
repository	Date, URL	Date when a mirror or SCM repository is published for the project
retired	Date	Date at which the project has technically ceased to be technologically relevant
sever	Developer, Role, Date	Date when a developer ceases interacting with a project in a given role.
successor	Project, Date	Links this project which is presumed to be retired with one or more projects that perform a similar role

In addition to projects, the database tracks "porgs", short for People and Organizations. These aren't users so much as names of parties who can impart changes on other entities in the database.

Event	Data Captured	Description
human		Date an individual developer begins involvement with projects and organizations within the Sobyk Bazaar
organization		Date an organization begins involvement with projects and organizations with in the Sobyk Bazaar
cancel	Developer, Event	Acts as a catch-all means of negating the effect or a previous event. The developer issuing the cancel is recorded, and thus parties can evaluate the veracity
fork	Developer, Name	Date when a developer (presumably an organization) splits into a second entity. That new entity gets a new identifier and name.
join	Developer, organization, role	Date one developer (presumably human) joins with another developer (presumably organization) and takes on a given role
kudos	Developer	Date when a developer offers praise to another developer.
merge	Developer	Date when two developers (presumably organizations) merge to form a singular developer. Also makes a handy cleanup if one accidentally creates two different accounts and builds a history with both.
nongratis	Developer	Date when one developer pronounces a different developer to be untrustworthy.
rename	Developer, name	Date when an individual or organization changes their name
retired	witness	Date when an individual developer or organization ceases all involvement with projects and organizations in the Sobyk Bazaar
sever	developer, organization, role	Date one developer (presumably human) cease performing a role for another developer (presumably organization).
successor	Developer, Date	Date when a developer indicates that all authority and trust granted to them should be passed to another individual or organization
user	Project, Date	Date when a developer indicates that it/he/she utilizes a project

Database Schema

The core schema for the Sobyk Bazaar is quite simple. We record one table with a UUID for each event, and another with a key/value list for all data we know about an entity. The eventid field is an optimization for the local database. It is generated by sqlite as an ever-increasing 64 bit integer. When records are exported, they are only ever referred to by UUID.

```
create database chronology (
  eventid integer primary key,
  uuid   guuid,
  entity guuid,
  source guuid,
  event  string,
  date   julian
);

create database chronology_info (
  eventid integer REFERENCES chronology(eventid) ON UPDATE CASCADE,
  field   string,
  value   string,
  PRIMARY KEY(eventid,field) ON CONFLICT REPLACE
);
```

As the stream of events is interpreted, Sobyk Bazaar will populate the following ephemeral tables:

```
create table entity (
  entityid integer primary key,
  uuid guuid,
  type string,
  name string,
  status string,
  mtime julian
);
create database entity_info (
  entityid integer REFERENCES entity(entityid) ON UPDATE CASCADE,
  field   string,
  value   string,
  veracity double,
  source  integer REFERENCES chronology(eventid) ON UPDATE CASCADE
);
create table link (
  linkid integer primary key,
  e1 integer REFERENCES entity(entityid) ON UPDATE CASCADE,
  e2 integer REFERENCES entity(entityid) ON UPDATE CASCADE,
  type string,
  veracity double,
  source  integer REFERENCES chronology(eventid) ON UPDATE CASCADE
);
create database link_info (
  linked integer REFERENCES entity(entityid) ON UPDATE CASCADE,
  field   string,
  value   string,
  veracity double,
  source  integer REFERENCES chronology(eventid) ON UPDATE CASCADE
);
create table distribution (
  entityid integer REFERENCES entity(entityid) ON UPDATE CASCADE,
  organization integer REFERENCES entity(entityid) ON UPDATE CASCADE,
  type string,
  date julian,
  name string,
  version string,
  profile string,
  url string,
  source  integer REFERENCES chronology(eventid) ON UPDATE CASCADE
);
```

As technology advances and the implementation matures, we will undoubtedly find those ephemeral tables needs to be restructured, or additional tables added. We will probably find other interactions that need to be added to the journal. But the core of our database, the chronology, will remain unchanged.

Conclusion

The Sobyk project is an attempt to create a project specific enough for the TCT to identify where core internal changes will affect downstream users, but generic enough that downstream users can customize it to suit their own ends. The actual build process for the system is finished and cutting production binaries for the Author's day job. Future plans for Sobyk will include a marketplace of ideas where the history of the projects and the people who develop, distribute, and use those projects can be recorded for posterity.

The Sobyk project's main website is: http://sobyk.com
The code is distributed via fossil at: http://sobyk.com/fossil/sobyk
Or on chisel app at:
https://chiselapp.com/user/hypnotoad/repository/sobyk/index